University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Level 4 Project Report

Euan McGrevey

March 20, 2020

Abstract

Two keys challenges have arisen since the advent of highly parallel computer architectures: Programmability and Performance Portability. This project used Lift(Steuwer et al. 2015a) as a template to explore how these issues could be addressed. Specifically it explores the Lift rewrite rule model for optimising programs applied to encode mathematical relations on expressions, and how efficiently rewriting of these expressions could be done. The developed interface for rewriting has proven to useful.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Euan McGrevey Date: 30 January 2020

Contents

1 Introduction

Lift is a research project that strives to ease the problems of Performance Portability (Writing optimised code for different computer architectures) and Programmability (How hard it is to write said optimised code).

Performance Portability is a challenge for programmers. It is inescapable due to the fact that hardware vendors design their processors differently from one another. Extensive performance guides are published by each, detailing the best ways to write code which takes full advantage of their processors. Ultimately this leads to programmers writing low level code that works really well on one specific architecture, and works worse or may not even compile on other architectures.

Programmability is another such issue that comes with writing optimised code. In current industry standard parallel programming models such as OpenCL, optimisations are typically ad hoc and unstructured, which results in unintuitive, hard to read code. Not only does this put a significant burden on the programmer, as writing optimised code is often time consuming, but it results in the code being inherently unmaintainable.

Compilers present the opportunity to solve or relieve both of these issues.

This project started out by aiming to investigate how the application of Lift's rewrite rules could improve the performance of common image processing techniques, such as blurring and up-scaling. However, over the course of the first semester of project work, it became apparent that this was much harder than first anticipated, and so the scope of the project was changed to "An Interface for exploring the application of rewrite rules in Lift".

The transformation of expressions using rewrite rules in Lift is of particular interest for Performance Portability and Programmability, as rewrite rules encode optimisation and implementation choices. High level Lift code that a programmer writes does not express to the compiler *how* the code will be executed, just the abstract solution.

For example, the absolute summation of values in an array can be written in high level Lift as:

```
1 asum = reduce (+) 0 map abs
```

Listing 1.1: *High level Description of Absolute Summation (Steuwer et al. 2015c)*

Note the map and reduce functional primitives that are exposed to the programmer. This allows for a concise and readable definition of the function, that does not describe how it is actually applied.

After running the high level code in Listing 1.1 through Lift's compiler, we can get several low level implementations for different hardware architectures, such as Listing 1.2, which is targeted for an Nvidia GPU.

```
1 λx.(reduceSeq join join mapWorkgroup (
```

```
    toGlobal mapLocal (reduceSeq ((a, b). a + (abs b)) 0) reorderStride
        2048
) split 128 split 2048) x
```

Listing 1.2: *Low Level implementation of Listing 1.1, targeting an Nvidia GPU (Steuwer et al. 2015b)*

Note the 'reduce' and 'abs' functions are still present, but the majority of the function is spent encoding optimisations with hard-coded numbers to denote how to lay out data in memory and how to split work between compute units on the GPU. As these numbers are specific to a particular Nvidia GPU, the optimisations do not carry over to other architectures, rendering it unportable.

Still, why might it be useful to decouple what code does from how it does it, as Lift does? Suppose we have a high level algorithm as in Listing 1.1. We could also write a low level optimized implementation separately. As the latter will tend to implement hardware specific and ad hoc techniques to gain maximal performance, there is often a lot of 'noise' and obfuscation in the code that can make it difficult to see that it still is in fact a valid implementation of the original high level algorithm, as with Listing 1.2.

This is where Lift uses rewrite rules, which encode these specific implementation optimisations, to tackle Programmability. The programmer need only specify a high level algorithm such as Listing 1.1, and decide which target architecture they would like, and the optimised implementation code is generated. This allows the opportunity for the programmer to write simple, easy to reason about programs, whose low level implementations, automatically generated by Lift, can be guaranteed to be correct, and thus save much time and effort on the programmers part. By way of design, this process can be done multiple times to generate many low level implementations that are optimised for different architectures, aiding the Performance Portability challenge.

This project aims to be the first stepping stone to this goal, by developing an interface through which we can explore the application of Lift-style rewrite rules.

2 Background

In this section we take a look at Halide, an image processing language that tries to tackle performance portability and programmability in an alternative way to Lift. We then compare this model to Lift, a more general language that aims to optimise programs automatically.

2.1 Halide

Halide is a language embedded in C++ designed to target image processing pipelines. As image processing techniques often require applying the same function to many different pixels of an image, exploiting data parallelism using a GPU or multi-core CPU is a natural way to enhance performance.

Halide attempts to tackle Performance Portability by decoupling 'what' the program does from 'how' it does it. A Halide program is split into two parts. The 'Algorithm', which specifies what the program will do, and the 'Schedule', which states how the algorithm will be executed.

The Schedule, written by the programmer, explains to the compiler things like how to allocate memory and how to order the instructions to achieve whatever is optimal for the programmer. In this way the programmer can minimise memory usage or runtime, for example.

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}
```

Listing 2.1: *A Halide program for blurring an image(Hal)*

We can see in Listing 2.1 Halide's approach to solving the programmability problem of parallel systems. The actual algorithm itself is entirely decoupled from any notion of optimisation, requiring the programmer only to state what has to be done.

This enables the programmer to write different schedules which optimise the same program for different hardware architectures of their choosing, with each schedule taking advantage of the

chosen architecture's design. This makes Halide a suitable choice when there are only a handful of target architectures for a product, as only a small number of schedules have to be written.

However, there are some major drawbacks. Firstly, Halide is only intended to be used for image processing pipelines. This intentional limit on the problem space is what allows it to be a practical platform to program with. Its problem scope is small so that it can still be a practical language for developers.

Secondly, it does require the programmer to write the schedule, which is often far more difficult than writing the actual program, when aiming for maximal performance. And if you want your program to be used on many different architectures, then you have to write a separate schedule for each. This makes it unsuitable for use in mainstream end-user products, since they will inevitably use myriad different types of computer architectures.

Lift aims to go for even more ambitious heights, and addresses both of these issues.

2.2 Lift

Lift aims to be a more ambitious research project. The language allows more than just image processing applications to be written. Similarly to Halide, Lift has a high level DSL provided for programmer to write their code. However, it differs in that the optimisation process is completely transparent in Lift. That is, there is no 'schedule', the programmer simply chooses a target architecture and Lift produces the low-level optimised code.

But how are particular hardware optimisations and implementation choices encoded? This is done through what are called rewrite rules. The Lift compiler takes program source code written by a developer and automatically generates several low level implementations, for example it could generate both C code and OpenCL code that do the same thing. The compiler may also produce different implementations of the code in the same language, where there might be implementations which are optimised for running on an AMD GPU or a multi-core Intel CPU, for instance.

Rewrite rules encode optimisations and implementation choices, and by being selective about which rewrite rules are applied, the Lift compiler can generated low level code on par with hand-optimised code for any particular architecture.

The Lift high level language exposes to the programmer common functional primitives with which to write programs.

```
1 dot = λ xs ys. (reduce (+) 0 ∘ map (*)) (zip xs ys)
```

Listing 2.2: *Lift code fragment defining the dot product(Steuwer et al. 2015b)*

Listing 2.2 shows a high level Dot Product, which uses the functional primitives 'zip', 'map' and 'reduce'. As with Listing 1.1, the definition is concise and easy to reason about. When Lift generates the low level implementation, we want to keep the same behaviour that this high level specification has.

3 Requirements

3.1 Requirements from motivation

Our basic question is this: "How can we find out if we can transform one expression into another?". In this section we will establish what exactly is required in order to answer this question, as well as the constrained version of this question that this project aims to answer.

Based on our question, we might want to establish a framework or suite of functions that manage transformations automatically, to save much time and effort.

For such an interface, we can draw up the following requirements. We want to know:

- Which rules are used in the transformation of expressions.
- What order they are applied in.
- Where in the program/expression AST they are applied.
- That certain rule applications don't change the meaning of the program (semantic preservation).
- That it is arbitrarily precise (that is, if the transformation is possible, the interface should be able to tell you definitively, with no false negatives).
- It is efficient.

This is a lot for a single project, so we will constrain our requirements to make it more manageable. Specifically, we will throw away efficiency for the sake of ease of programming. And we will not be rigorous when it comes to proving the application of certain rewrite rules do not alter the meaning of the program. Such proofs may be suitable for further work.

3.2 Success Criteria

In order to know how well we have satisfied our constrained requirements, let us specify some more concrete goals:

- We would like to develop a series of functions which transform expressions.
- For our functions, we would like them to tell us precisely which rules are used in an expression transformation, as well as the order of the rule applications.
- Furthermore, the functions should output the location where the rule applications happen, in such a way that the transformations can be replicated exactly.
- Finally, our functions must be able to take arbitrary start and goal expressions, and any number of rewrite rules to work with.

If we can achieve this, then we will have a primitive version of the universal expression transformer that satisfies requirements 1,2, and 3. To satisfy the requirement that our functions are arbitrarily precise, we can allow our functions a stop parameter which will only permit the function to allow so many rule applications before giving up (that is, if a transformation can't be done in X rule applications or less, it will be treated as though it is impossible).

In chapter 4, we design a series of functions that handle the traversal of a more abstract data type, the tree.

These functions are then implemented in chapter 5.

Chapter 6 uses our traversal functions with a set of rewrite rules modeling Arithmetic properties such as Commutativity for a case study evaluating how well we have fulfilled our success criteria.

4 Design

4.1 Data Representation

As Lift is a DSL built in Scala, this project uses Scala for ease of extensibility.

4.1.1 Abstract Syntax Trees

One of the steps early on during the compilation of a program is the reduction of the source code to an Abstract Syntax Tree (AST). An AST is a representation of a program's source code with all unnecessary information expunged. Thus we use trees as a basic data structure to represent expressions since they can model the design of ASTs and may be extended in future work to accommodate more programming language features. Eventually we may be able to model entire programs, which is needed in order to answer our question presented in Chapter 1, "How can we be sure that a low level optimised implementation of a high level algorithm is correct?"

4.1.2 Expressions

Our trees are modeled as a trait, with two possible values, either an empty node, or a node with a value and two children, which may be empty nodes themselves.

```
1 sealed trait Tree {}
2
3 case class Node(var left: Tree, var value: String, var right: Tree)
      extends Tree {}
4
5 case object EmptyNode extends Tree {}
```

Listing 4.1: *Definition of a Tree*

For the purposes of making implementation easier, our trees are strictly binary. Each node has a String value. This string can be anything, and is used to encode information about the node, and can be used to pattern match in rewrite rules or elsewhere. These trees do not need to encode a full program AST. For the purposes of this project, we can use them to represent a smaller set of arbitrary ASTs of computations.

Since our goal is to have a universal expression transformer, we need some way of traversing a given tree as well as a way of deciding when a rule has been successfully applied to a node in the tree.

4.2 Rewrite Rules

Also referred to as a Strategy (this is what they are called in Lift), a rewrite rule somehow transforms an expression whilst maintaining its meaningfulness. While semantic preservation

may not be true for all rewrite rules you could write, it is desirable for the purposes of this project to only use such rules. Proofs that certain rewrite rules do indeed preserve semantic meaning are beyond the scope of this project.

More concretely, rewrite rules are functions, parameterised over some type P, which take an object of type P as input and return a RewriteResult, which is also parameterised over P.

This make sense, the rewrite rule interface is type-agnostic, and each individual rule only works on whatever data type it specifies. This allows a small and simple interface. Listing 4.2 lays out this definition.

```
type Strategy[P] = P => RewriteResult[P]
```

Listing 4.2: *Definition of a Rewrite Rule as a type alias*

4.2.1 Rewrite Result

Listing 4.3 provides the interface for the output of rewrite rules, RewriteResult, which allows us the ability to decide whether a rule application was a success or failure.

We use a Scala trait to represent the result of applying rewrite rules to a node, with case classes for Success and Failure. This trait is primarily just a wrapper for the result value and is used for pattern matching purposes, as we can decide what to do when rule application on a particular node does not work via pattern matching over these two case classes.

```
// omitted are functions for retrieving the result of the rule
    application
sealed trait RewriteResult[P] {...}

case class Success[P](p: P) extends RewriteResult[P] {...}

case class Failure[P](p: P) extends RewriteResult[P] {...}
```

Listing 4.3: *The RewriteResult trait definition taken directly from Lift (omitted are functions for retrieving value from the wrapper object, not used in this project)*

4.2.2 Rule Application

Listing 4.4 provides an example rewrite rule, which takes an empty node and turns it into a tree with a left child.

```
case class Section4Example() extends Strategy[Tree] = {
   def apply(e: Tree): RewriteResult[Tree] = e match {
      case EmptyNode => Success(Node(Node(EmptyNode, "x", EmptyNode),
          "y", EmptyNode))
      case _ => Failure(Section4Example())
   }
}
```

Listing 4.4: *A Rewrite rule which takes an Empty node, and replaces it with a Node with one left child*

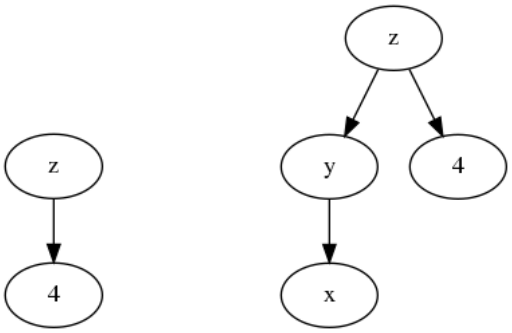

Figure 4.1: *Before and after application of the rule in Listing 4.4 to the empty left child of the left expression*

Note the return values are wrapped in either Success() or Failure() to indicate whether the rule application worked.

To help visualise the effect of applying rewrite rules, the application of the rule presented in Listing 4.4 can be visualised as in Figure 4.1

Now that we have talked about our tree representation and how rewrite rules work, we can discuss the traversal functions that will make up our interface for exploring the application of rewrite rules on expressions.

4.3 Traversals

As our trees are binary, and because trees themselves are naturally recursive, our traversal functions become much more straightforward, since we only need to decide what happens to a single node and both its children.

We now go through each of our success criteria as in chapter 3, and design a way in which we can meet them.

From the definition of our rewrite rules, they are functions which return a rewrite result. Thus we can answer whether a rule application was successful by pattern matching on the return value, which will either be a Success or a Failure. Therefore, we can design a simple function which takes a start and goal expression and one rewrite rule, and does a preorder traversal of the start expression from the root, trying to apply the rule to each node, and on a Success, we can check whether the result is the goal. If it is then we know the transformation is possible. If not, then we can simply continue the traversal until we run out of nodes, in which case it is also not possible.

This is the simplest version of our goal, as it only takes one rewrite rule, does not tell you where the rule application happened if successful, and only has an exploration depth of one. That is, if it were possible to get to the goal expression, but it took two rule applications, then the function would still return false.

For keeping track of which rules are used in the process of transforming one expression into another, as well as the order in which they are applied, we can maintain an ordered sequence of rules to represent the order of rule application.

Returning the location of where in an expression a rule is applied is a little tougher, since in our design we do not have a unique identifier associated with each node in an expression. Thus, we must develop another way. Our solution is to design two functions.

- applyOnce, which traverses an expression, trying to apply a given rule to each node, stopping on the first successful application, or failing after all nodes fail.

- And applyOnceWithSkip, which does the same as applyOnce, but is provided with a number denoting how many successful rule applications to skip over during its traversal.

If we choose to do a preorder traversal for both of these functions, then the order of rule application attempts will always be the same. Thus, when there are many potential nodes to which a given rule can be applied, this 'skip' number will allow us to recreate the result of a particular rule application, by doing a preorder traversal over the expression and skipping that many successful applications, after which point we will be at the desired location in the expression.

Finally, we do not need to do any work to allow our functions to take arbitrary start and goal expressions, and in order for them to take any number of rewrite rules, we can include a Set of rewrite rules as a parameter for the functions.

5 Implementation

In the implementation for these functions, each node in an expression can be either an operation or an operand. The precedence of operators corresponds to depth in the tree. Each operation node has two children, which are its operands, and these may be operations themselves.

Note that we do not place any restriction on what an operand may be, which allows the flexibility of representing Algebraic terms, where operands may be letters such as "A" or "B". This comes at the cost of not checking correctness of expressions, to ensure they are well formatted or meaningful. For example, you could have a node with value "4" that has two children "3" and "5". This may not make any sense, but it is not checked for.

5.1 Traversal Function Implementations

Here is a presentation of the function implementations whose design is laid out in chapter 4. Each one builds on the previous until we reach our goal of having an arbitrarily precise function that outputs whether it is possible to go from one expression to another, and provides a means to replicate the steps it found to do so.

5.1.1 applyOnce and applyOnceWithSkip

We implement two of the functions from our design that will enable us to replicate the application of a rewrite rule at a particular position in an expression.

applyOnce, referred to in chapter 4, is the most constrained version of the problem scope, where we want to answer whether we can apply *one* rule, *exactly once*.

Listing 5.1 provides the implementation for this function.

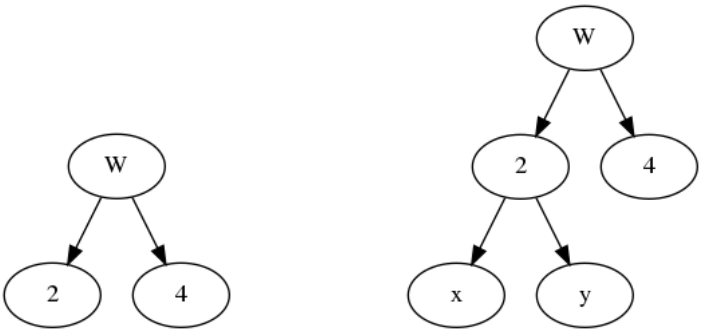

Figure 5.1: *The result of calling applyOnce in Listing 5.1 on the Left diagram with a rule that takes a leaf node and gives it two children. Note as the traversal is done preorder, the left child node of the root is checked first thus the result of applyOnce gives this node two children.*

```
def applyOnce(t: Tree, r: Strategy[Tree]): RewriteResult[Tree] = {
  r(t) match { // try the current node
    case Success(newT) => Success(newT)
    case Failure(_) =>
      t match {
        case EmptyNode => Failure(r)
        case Node(ln, s, rn) =>
          // consider left child...
          applyOnce(ln, r) match {
            case Success(newlT) => Success(Node(newlT, s, rn))
            case Failure(_) =>
              // consider right child...
              applyOnce(rn, r) match {
                case Success(newrT) => Success(Node(ln, s, newrT))
                case Failure(_) => Failure[Tree](r : Strategy[Tree])
              }
          }
      }
  }
}
```

Listing 5.1: *Function which applies a given rule to only one point in a given tree, returning the transformed tree on a Success or the original on a Failure*

In order to ensure that the supplied rewrite rule is only ever successfully applied once, we must consider the left child node before the right child node, or vice versa.

In this case, we prioritise the left child over the right so that the traversal is done preorder. The same is true of all the functions we will develop. To make it easier to visualise what this function actually does, Figure 5.1 provides an example.

As applyOnce does a preorder traversal, the leftmost leaf node in the left expression is the one that first matches in the function, and so of the two leaf nodes that our supplied rule could work on, the left one is what applyOnce finds first and returns the right expression. The left leaf node value is not changed by the rule application, but it is given two child nodes. Note that in this example the node values have no meaning yet, they are just dummy data.

We now examine applyOnceWithSkip, which builds on applyOnce to deliver more functionality. This is our key function from chapter 4 for solving the problem of replicating rule application in a particular place in an expression.

This function is very similar to the previous one. We add an input parameter 'skip', which is an int representing how many successful rule applications to skip during the expression traversal. If our rule application is successful, but skip is still positive, then we decrement skip and continue the traversal.

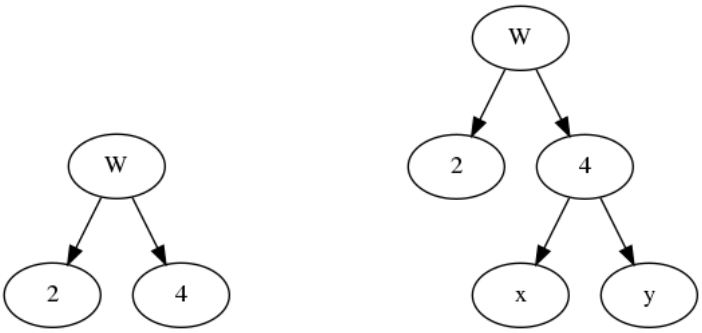

Figure 5.2: *An example of calling the function in Listing 5.2 with skip value of 1 in on the Left expression, with the same rewrite rule as in Figure 5.1*

```
def applyOnceWithSkip(t: Tree, skip: Int, r: Strategy[Tree]): (Int,
    RewriteResult[Tree]) = {

  def recurse(skip: Int): (Int, RewriteResult[Tree]) = {/* Handle the
      recursive case */}

  r(t) match {
    case Success(newT) =>
      if (skip == 0) {
        (0, Success(newT))
      } else {
        recurse(skip - 1)
      }
    case Failure(_) =>
      recurse(skip)
  }
}
```

Listing 5.2: *The same as Listing 5.1, but with an additional skip parameter n which will 'skip' n successful rule applications before returning the result of a successful one*

Naturally, if you pass a skip value of 0, then this function works exactly the same as applyOnce. However, this algorithm is inefficient, as the same computations are repeated n times for an initial skip value of n and enough possible successful rule applications. It is not true in general that the number of successful rule applications is known, and so we will need to develop functions later on to figure this out. For the purposes of this function, it is assumed to be known.

Figure 5.2 shows the result of applyOnceWithSkip with the same input expression and rewrite rule as Figure 5.1, but with a skip value of 1. Note how during the preorder traversal, we visit the leftmost leaf node first, but as we passed a skip value of 1, return the successful rule application here. We continue to the right leaf node, and with our decremented skip value of 0, we return the result of the successful rule application on the right leaf node.

5.1.2 applyOnceNTimes and applyOnceNTimesReturnSkip

We now move to develop functions that can return multiple trees from all possible successful rule applications on an expression. applyOnceNTimes in Listing 5.3 returns a set of trees representing

the result of all possible successful rule applications of a single rule on a single expression.

```
def applyOnceNTimes(t: Tree, r: Strategy[Tree], set: Set[Tree]):
    Set[Tree] = {
  applyOnceWithSkip(t, set.size, r) match {
    case (_,Failure(_)) => set // couldn't find a rule application that
        resulted in something new
    case (_,Success(newT)) =>
      t match {
        case EmptyNode => set + newT // we succeeded in rule application,
            but can't recurse anymore
        case Node(lt, v, rt) =>
          applyOnceNTimes(t, r, set + newT) // keep recursing until
              applyOnceWithSkip fails.
        // Each success should add an element to the set,
      }
  }
}
```

Listing 5.3: *Returns the set of all possible expressions that could be obtained from the successful application of a rule on a given expression*

We utilise applyOnceWithSkip from Listing 5.2 to progressively build up a set of the possible resultant trees. By continually calling applyOnceWithSkip, increasing the skip parameter each time until it fails, we ensure that every node is tried, and all nodes that the rule is successfully applied to have their result added to our set. Thus, we now have a function that can tell us where one particular rule can get us to in one step. Recall that we are building to multiple rules across multiple steps.

To solve our earlier problem with applyOnceWithSkip where we don't know in advance how many skips we will need, Listing 5.4 presents a version of Listing 5.3 that also returns how many skips were required to get each resultant tree in the set.

```
def applyOnceNTimesReturnSkip(t: Tree, r: Strategy[Tree], set:
    Set[(Tree, Int)]): Set[(Tree, Int)] = {
  applyOnceWithSkip(t, set.size, r) match {
    case (_,Failure(_)) => set //
    case (_,Success(newT)) =>
      t match {
        case EmptyNode => set + Tuple2(newT, set.size)
        case Node(lt, v, rt) =>
          applyOnceNTimesReturnSkip(t, r, set + Tuple2(newT, set.size))
      }
  }
}
```

Listing 5.4: *The same as Listing 5.3, except that we return how many skips each resultant tree took to achieve*

Now we have a complete set of functions that allow us to find out what possible trees we can get as a result of applying a single rewrite rule once, as well as a method for recreating all of these transformations. We now begin answering our question of whether we can turn one expression

into another.

5.2 Expression Transformers

At this point, we have all the components necessary to begin answering the question of "Provided with a set of rewrite rules to use, can we get from one expression to another?"

5.2.1 naiveExpressionTransformer and naiveExpressionTransformerReturnOrder

This expression transformer takes a start and goal expression, a set of rules with which to achieve the transformation, and an integer 'depth' representing the maximum number of rule applications before giving up.

Listing 5.5 lays out the implementation for our first expression transformer. Because we are ignoring efficiency and performance, there is an exponential time complexity which increases with the number of possible successful rule applications. This is because after collating the set of all resultant trees by calling applyOnceNTimes for each rule, we recurse on all of these different trees. This gives order O(n^m) time complexity in the worst case, where n is the number of nodes and m is the number of rules used, but this is extremely unlikely in practice, as it requires every rule to be applicable at every node in an expression.

```
def naiveExpressionTransformer(begin: Tree, goal: Tree, rules:
    Set[Strategy[Tree]], depth: Int): Boolean = {
   if (depth == 0) return false

   var candidates : Set[Tree] = Set()
   for (rule <- rules) {
      val rulecans = applyOnceNTimes(begin, rule, Set())
      candidates ++= rulecans
   }
   // candidates should now hold all possible expressions we could get
       from successfully applying one of the rules in the provided set
       once.

   for (can <- candidates) {
      if (can == goal) return true
   }
   for (can <- candidates) if (naiveExpressionTransformer(can, goal,
       rules, depth - 1)) return true
   return false // if all else fails
}
```

Listing 5.5: *The first and simplest expression transformer*

This transformer only outputs whether it found a possible transformation, and not how to recreate it. To do that, we must return the rules used, in order, and the number of skips in a preorder traversal before applying each rule.

Our next transformer will tell us not only if the transformation was possible, but also the rules used to do the transformation, in order of application.

```
def naiveExpressionTransformerReturnOrder(begin: Tree, goal: Tree,
    rules: Set[Strategy[Tree]], depth: Int, appOrder:
    Seq[Strategy[Tree]]): (Boolean, Seq[Strategy[Tree]]) = {
  if (depth == 0) return (false, Seq())

  var candidates : Set[(Strategy[Tree],Set[Tree])] = Set()
  for (rule <- rules) {
    val rulecans = applyOnceNTimes(begin, rule, Set()) // rulecans is the
        set of trees possible from successful applications of rule
    candidates += Tuple2(rule , rulecans) // keep track of which trees
        came from which rule
  }


  for ((rule, rulecans) <- candidates) {
    for (can <- rulecans) if (can == goal) return (true, appOrder :+ rule)
  }

  for ((rule, rulecans) <- candidates) {
    for (can <- rulecans) {
      naiveExpressionTransformerReturnOrder(can, goal, rules, depth-1,
          appOrder :+ rule) match {
        case (true, newAppOrder) => return (true, newAppOrder)
        case (false, _) => (false, Seq())
      }
    }
  }

  return (false, Seq())
}
```

Listing 5.6: *The same as Listing 5.5, except that it returns the order in which rules were applied if successful*

Things get interesting here as there may be more than one solution. That is, more than one way to achieve the transformation from start to goal expression using the same rules. Some methods may be more desirable than others. However, as there is no priority or preference given to any particular rule, this means that we might find a solution that takes much longer than is necessary.

The good news is that if the transformation is possible within the provided depth, all of our transformers will still find the way to achieve it. This is due to the fact that we simply try every possible combination of rules against every node in our starting tree, as well as all intermediate trees. We can stop if at any point one of our resultant trees is in fact the goal expression.

5.2.2 universalExpressionTransformer

The difference between our previous two transformers and this one is that it returns a way for the solution found to be recreated by using applyOnceWithSkip from Listing 5.2 iteratively with each returned rule and corresponding skip value. Listing 5.7 presents this final function which completes our interface.

```

def universalExpressionTransformer(begin: Tree, goal: Tree, rules:
    Set[Strategy[Tree]], depth: Int, appOrder: Seq[Tuple2[Strategy[Tree],
```

```
    Int]]) : (Boolean, Seq[Tuple2[Strategy[Tree], Int]]) = {
  if (depth == 0) return (false, Seq())

  // we track which rule led to which resultant trees, as well as the
      number of skips for each tree-rule combination
  var candidates : Set[( Strategy[Tree] , Set[(Tree, Int)] )] = Set()
  for (rule <- rules) {
     val rulecans = applyOnceNTimesReturnSkip(begin, rule, Set()) //
         rulecans is the set of trees and their skip values
     candidates += Tuple2(rule , rulecans)
  }

  // Check for the goal expression
  for ((rule, rulecans) <- candidates) {
     for ((can, skips) <- rulecans) {
        if (can == goal) return (true, appOrder :+ (rule, skips))
     }
  }

  // recurse
  for ((rule, rulecans) <- candidates) {
     for ((can, skips) <- rulecans) {
        universalExpressionTransformer(can, goal, rules, depth-1,
            appOrder :+ (rule, skips)) match {
           case (true, newAppOrder) => return (true, newAppOrder)
           case (false, _) => (false, Seq()) // just keep going
           case _ => ???
        }
     }
  }

  return (false, Seq())
  }
```

Listing 5.7: *Our final traversal function, which returns the information needed to replicate the transformation from start to goal expression it found.*

With this, we have developed a way of transforming expressions, and a sure way of testing whether it is possible to use particular rewrite rules to get from some starting expression to some goal expression.

We now move on to evaluate the effectiveness of the functions shown in this chapter.

6 Evaluation

In this chapter, we will take a look at a concrete implementation of the ideas discussed in the previous chapter, through a case study. We have encoded actual transformation rules that correspond to mathematical properties and our trees that rules are applied to represent mathematical expressions (such as 3 * (4 + 5)).

First we show the implementations for some of the properties we have encoded as rewrite rules, and then we evaluate the performance of our transformers from chapter 5.

6.1 Case Study: Arithmetic Expressions

It is important to note that the mathematical properties we encode for Arithmetic (Associativity, Commutativity, and Distributivity) do not change the value of the expression, just the way they are written.

6.1.1 Associativity as a rewrite rule

Associativity is the property that both Addition and Multiplication share, where the order you evaluate multiple operations in does not affect the result. For example, (3 + 4) + 5 is the same as 3 + (4 + 5).

For the sake of modularity we define left and right Associativity (swapping the left argument and right argument respectively) separately, but put together they offer all the same capabilities that we need for the Full Associativity of Addition and Multiplication.

To illustrate our implementation, Figure 6.1 illustrates how it interacts with the aforementioned (3 + 4) + 5.

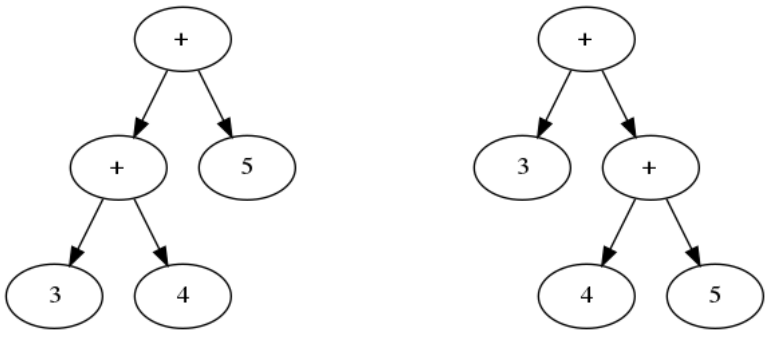

Figure 6.1: *Left: The expression representing (3 + 4) + 5; Right: The result of applying our Associativity rule to the left expression*

```
// Associativity - Order of Operations irrelevant - Order of two or more
    of "+" or "*" in same precedence doesn't matter
case class LeftAssociativity() extends Strategy[Tree] {
   def apply(e: Tree): RewriteResult[Tree] = e match {
      case Node(lr, v, c) if v == "+" || v == "*" =>
      lr match {
         case Node(a, v1, b) if v1 == v =>
            Success(Node(a, v, Node(b, v, c))) // (a + b) + c == a + (b
                + c)
         case _ => Failure(LeftAssociativity())
      }
     case _ => Failure(LeftAssociativity())
   }
}

// Similarly for Right Associativity...
```

6.1.2 Commutativity as a rewrite rule

Commutativity is the simplest of the three properties presented; It simply means you can swap the arguments in an Addition or Multiplication freely.

```
case class Commutativity() extends Strategy[Tree] {
   def apply(e: Tree): RewriteResult[Tree] = e match {
     case Node(lr, v, rt) if v == "+" || v == "*" =>
       Success(Node(rt, v, lr)) // a + b => b + a
     case _ => Failure(Commutativity())
   }
}
```

6.1.3 Distributivity as a rewrite rule

Distributivity is a bit more involved than the previous two, since it requires two functions in it's definition. Simply put, if Multiplication distributes over Addition, then any multiplication that has an Addition as one of its operators can be rewritten into two separate multiplications.

This is best illustrated with another diagram, as in Figure 6.2.

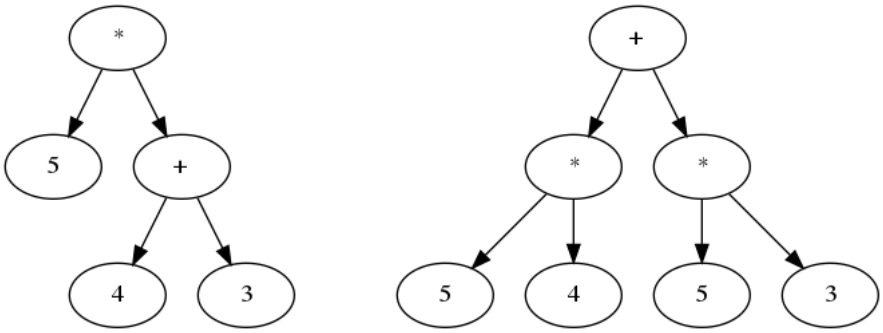

Figure 6.2: *Before and after applying Distributivity to split the left expression into two separate multiplications*

```
// Distributivity - Multiplication distributes over Addition i.e. a * (b
    + c) == (a * b) + (a * c)
 case class Distributivity() extends Strategy[Tree] {
  def apply(e: Tree): RewriteResult[Tree] = e match {
   case Node(a, v, rt) if v == "*" =>
    rt match {
     case Node(b, v1, c) if v1 == "+" =>
      Success(Node(Node(a, "*", b), "+", Node(a, "*", c))) // a * (b
          + c) => a*b + a*c
     case Node(b, v1, c) if v1 == "-" =>
      Success(Node(Node(a, "*", b), "-", Node(a, "*", c))) // a * (b
          - c) => a*b - a*c
     case _ => Failure(Distributivity())
    }
   case _ => Failure(Distributivity())
  }
 }
```

6.2 Examples

Now that we have shown some of the properties we have encoded, we can evaluate how our expression transformers work with some examples.

6.2.1 Simple Example

Let's first take a look at a simple example.

If we create trees which represent the expressions (3 + 4) * 5 and (5 * 3) + (5 * 4) as in Figure 6.3, we can see that we can go from the left expression to right expression by first applying the Commutativity rule on the multiplication, and then splitting the multiplication in two using the Distributivity rule.

We now examine what happens when we call our universalExpressionTransformer from Listing 5.7 with these start and goal expressions. As expected, we learn that the transformation is possible, but the method provided by our function to do so is far from the optimal two stage rule application we desire.

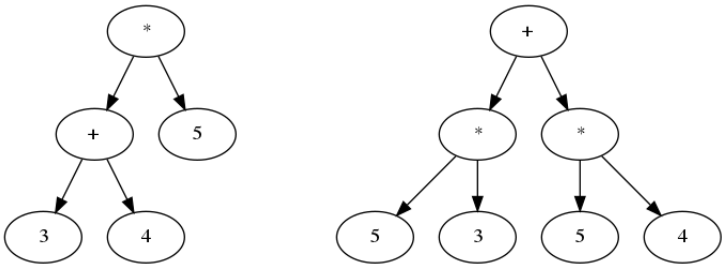

Figure 6.3: *Start and Goal expressions*

Interestingly, the function applies Commutativity twice on the same operation (to get back to where we started). The solution it presents is double the length of the optimal one, but it does indeed find a solution.

The same surplus of steps can be seen in various other examples, and our expression transformer almost never finds the optimal solution first. There are ways to aid this problem, such as giving priority to each rule, as certain rules tend to be more niche but better than other solutions when they can be applied. However, there was no time left to add more to the project.

7 Conclusion

7.1 Research Outcomes

As it turned out for me during my first semester working on this project, research projects are terribly complicated and hard to get stuck into if you've never worked on one before, and your also not familiar with the key topics that it is based around. Also, the fact that research is not for public use and so is does not have extensive documentation that you can simply look up online made it hard to make progress when stuck on a problem.

That being said, my supervisor did a great job supporting me and helping pivot when things weren't going well and changing the scope of the project to be something that I would have an easier time with.

In the end, this meant that I didn't work with actual Lift expressions, and had to have a pseudo representation of much simpler expressions. Still, I hope that this project may prove even slightly useful in the future.

7.2 Future Work

As this project aimed to develop a prototype for a more general Lift program transformer, it succeeded in providing a basis from which more complicated language constructs could be represented, and along with the proofs that certain rewrite rules do no alter the semantic meaning of a program, could become the first stepping stone in proving the correctness of low level code implementations generated by Lift from high level user code.

7.3 Conclusion

In conclusion, we developed a function interface for exploring the application of Lift-style rewrite rules. Specifically, we answered whether or not certain rewrite rules could transform one expression into another. The resulting interface does indeed solve this problem, but not as efficiently as possible, often times providing a solution that is more than twice as long as optimal.

A Appendices

7 Bibliography

Halide website. URL https://halide-lang.org/.

Steuwer et al. Improving programmability and performance portability on many-core processors. page 277, 2015a.

M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance opencl code. In K. Fisher and J. H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 205–217. ACM, 2015b. doi: 10.1145/2784731.2784754. URL https://doi.org/10.1145/2784731.2784754.

M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules. page 42, 2015c.