Program Transformations in Magnolia

Kristoffer Haugsbakk

Master Thesis
June 2017

Bergen
Language
Design
Laboratory

Department of Informatics
University of Bergen

Supervisors
Magne Haveraeen
Anya Helene Bagge

To my dear parents and Gautama Buddha

Contents

Abstract

We explore program transformations in the context of the Magnolia programming language. We discuss research and implementations of transformation techniques, scenarios to put them to use in Magnolia, interfacing with transformations, and potential workflows and tooling that this approach to programming enables.

Acknowledgments

I would like to thank my two supervisors, Magne and Anya, for their continual support through this long time. I would like to thank my loving parents for their tireless patience and support. I would like to thank Tero Hasu for all of our thought-provoking conversations about programming and programming languages. I would like to thank Eivind Jahren, Anna Maria Eilertsen and others for their moral support which turned out to be crucial for this process.

1 Introduction

We describe the goal and motivation for this thesis. Then we give an outline of the thesis.

1.1 OVERVIEW

This thesis explores ideas and concepts from the field of program transformations. The ideas are put in the context of the Magnolia programming language, a general-purpose research language focused on specifications and the domain of high-performance computing (HPC). Magnolia is a fruitful place to explore program transformations. The reason for that is that it is purposefully designed to be easy to analyze. Mainstream languages—like Java, C, C++—have semantics that complicates analysis. Magnolia avoids this by—in simple terms—restricting or avoiding certain features and capabilities.

Program transformations is a broad term for manipulation of programs. Some of the uses of program transformations are:

1. Optimisation
2. Code generation
3. Reverse engineering

We will explore both the utility of program transformations in general, and how Magnolia is amendable to transformations.

1.2 OUTLINE

The thesis is structured as follows:

1. Introduction to program transformations (chapter 2).
2. Introduction to the Magnolia programming language (chapter 3).
3. Explorations of two program transformation techniques (chapters 4 and 5).

4. Two case studies in program transformations applied to Magnolia programs (chapters 6 and 7).

5. An interface for program transformation (chapter 8).

6. Workflow and tooling for working with program transformations (chapter 9).

We start by introducing and motivating program transformations. Since program transformations is a big field, we explain our own approach within the discipline, and contrast it with other approaches, as seen in programming languages and the research literature. Then we give a short introduction to Magnolia; its purpose and the important constructs that it offers. We only describe the things that set it apart from other languages; we do not go into the details of syntax and semantics which are similar to other languages. Then we drill down into two specific program transformation techniques; partial evaluation and *slicing*. These techniques, among others, are then put to use in two case studies of program transformations applied to Magnolia. First we consider working with varying implementations of a Modulus Group. We discuss and go into detail on how one can, given one implementation of this concept, transform that implementation into different implementations. Second we consider how transformations can be used to derive a data type from another data type. Then we change gears and consider the bigger picture of workflow and tooling in the context of program transformations. We explain how the more structured approach that we propose for programming can be leveraged to derive more structured artifacts of programming, using tools like version control systems (VCSs). Finally, we discuss how the programmer can interact with transformations. We envision many interfaces, but we focus on an interface using *directives*.

2

Program Transformation

We introduce and motivate program transformation *as a general technique and as it applies to Magnolia. We discuss the potential it has to support the craft of programming and the process of software engineering.*

2.1 INTRODUCTION

> The time is clearly ripe for program-manipulation systems (Donald Knuth, 43 years ago[Knuth, 1974].)

What is meant by *program transformation* is straightforward. It is a transformation between programs. We do not need to say more about this before we get into specific kinds of transformations. In this chapter we will not care about dry definitions. Here I will make the case for program transformations and why you should care about them. Both in general and in the case of Magnolia.

You might not trust me enough to take my word for it. But as it happens I am not alone. I can make the first case for this technique right now, namely that it is an old and unoriginal idea that neither me nor my supervisor came up with. And we all know that all great ideas are not original.

2.2 EXAMPLES

Manually renaming a variable at every point of use is an example of a program transformation, specifically a refactoring. Automatically doing the same with the help of an integrated development environment (IDE) is an example of a program transformation. Extracting a procedure or a method from a sub-block[1] of statements—manually or with tool help—is another example. In order to deal with some specific examples right away, we can consider some semantics-preserving transformations that simplify *expressions*.[2] The arrows indicate the transformation steps.

[1] An example of a block is a for-loop.

[2] An expression is a combination of explicit values, variables, operators, and functions that can be evaluated to a value. This is to be contrasted with a *statement* that is evaluated for its *side effect* (like printing some text), and does not result in a value. Expressions may or may not have side effects; in Magnolia they do not.

$$0 + \texttt{a} \Rightarrow \texttt{a}$$
$$5 + 5 + \texttt{a} \Rightarrow 10 + \texttt{a}$$
$$\texttt{power}(\texttt{x}, 3) \Rightarrow \texttt{x} * \texttt{x} * \texttt{x}$$
$$\texttt{x} * \texttt{y} == \texttt{y} * \texttt{x} \Rightarrow \texttt{x} * \texttt{y} == \texttt{x} * \texttt{y} \Rightarrow \texttt{true}$$

These kinds of transformations essentially boil down to doing algebraic manipulation on the program.

Your compiler transforming the program you wrote into efficient machine code is yet another example of a program transformation. The rest of this section goes into a concrete example of some techniques that a compiler might use to optimize a program. As the description does not assume that the reader is familiar with the theory, design and/or implementation of programming languages, readers who are might want to skip ahead to the next section.

A compiler optimizes a program through many transformations, and the transformations are typically done on some intermediate representation (IR) and not the original source code. But you do not need to learn to slay dragons[3] in order to understand the essential concepts. A compiler might be built to do two kinds of transformations for the purpose of optimisation:

1. Simplifying the program (normalization).
2. Optimising the simplified program.

Point 1 consists of expressing certain constructs in the program using simpler building blocks. It is like when you help your friend with a writing assignment, and you notice that he has become too good at English for his own good. He has written about a simple and relatable everyday event, but using complex and obscure synonyms in order to show off his talent for language. If you help him simplify his style, his article can tell the same story but in a clearer and simpler way. Now, this is not a perfect analogy. A programming language does not have constructs that strictly speaking could be expressed using simpler constructs just to be fanciful. Such constructs are meant to be a convenience for the programmer. But for the compiler, it is easier to analyse a simplified language. And a compiler does not have eyes that begin to glaze over once it starts seeing the same basic constructs repeated over and over again in a section of the code.

So simplifying a program consists of transforming a program into an equivalent, simpler one. An example of a construct that can be simplified in Java[4] are the increment operators ++. A statement `i++;` can be expressed using

[3] In reference to "the dragon book", which might refer to both Aho et al. [1986] and Aho and Ullman [1977]. Slaying the dragon is a metaphor for conquering complexity.

[4] Or most languages with a C-like syntax.

the more general operator += as `i += 1`. That statement can in turn be expressed using an even more general construct, the assignment operator =, as `i = i + 1;`. So once your Java compiler has simplified your program to not use the ++ and += operators,[5] the next program transformations it will do will be simpler to perform. This is because there are two less constructs to consider. So the compiler will not have to worry about encountering statements like this:

```
array[i++] = a++;
```

Which after simplification of the program would actually consist of multiple statements, not just one.

Point 2, optimisation of the program, involves transforming the program in such a way that it runs faster, uses less memory, and so on[6]. What is an *optimisation* depends on the needs of the user, but since a compiler has to serve *all* of its users it has to settle for these lowest common denominators of *being less slow* and *hogging less memory*. As an example of an optimisation, say you have a program that both finds the sum of an array and the maximum value in it. A sort of multitasking program. You have coded this program in a clean and elegant way, separating the two concerns of summing the array and finding the maximum value into two methods. In Java:

```
public class Main {
   public static void main(String[] args) {
      int[] array = {1,2,3,4,5};
      System.out.println(sum(array));
      System.out.println(maximum(array));
   }
   private static int sum(int[] array) {
      int s = 0;
      for (int e: array) {
         s += e;
      }
      return s;
   }
   private static int maximum(int[] array) {
      int max = Integer.MIN_VALUE;
      for (int e: array) {
         if (e > max) { max = e; }
      }
      return max;
   }
}
```

[5]In the case of Java such things might be done using *Just in Time Compilation* in the virtual machine at runtime, but such distinctions are not important for our purposes.

[6]Using less memory and running faster might both be achieved, one of them achieved, or one of them at the cost of the other. The last strategy is called a *space-time tradeoff*.

In the "real world" the array would not be given in the program but instead given as an argument to the program, for example. But it is just a simplified example. Now you would naturally be very pleased with this program, as it perfectly implements your desired "multitasking" program. But for all its elegance, it could probably be faster. The first observation we will make is that we have defined two methods that are each only called once. From the machine's viewpoint, methods can help reduce the resulting size of the program, since methods can be called from multiple places. Instead of having to duplicate the code of the method in each place that it is called. But the drawback of methods—again from the machine's viewpoint—is that calling methods induces *call overhead*. And in our example, since these functions are only called once each, we get the drawbacks of using methods without getting any of the benefits. To make up for this, the compiler can choose to *inline* the code of the methods each place that it is called. This means to replace the method call with the whole body of the method. After such a transformation the code might look something like this:

```
public class Main {
   public static void main(String[] args) {
      int[] array = {1,2,3,4,5};
      int s = 0;
      for (int e: array) {
         s += e;
      }
      int max = Integer.MIN_VALUE;
      for (int e: array) {
         if (e > max) { max = e; }
      }
      System.out.println(s);
      System.out.println(max);
   }
}
```

What we did to derive this program was to

1. move the calls `sum(array)` and `maximum(array)` into separate variable declarations (*extract variable*);

2. replace the calls in those variable declarations with the bodies of the methods (inline); and then

3. remove the two methods that are no longer needed.

There is something that might strike you with this new, transformed code. It is ugly! But that is what compilers do. They transform your elegant, not-quite-efficient code into ugly, more efficient code.

There is another optimisation transformation that we can do, and it would probably have more impact (hopefully for the better) than the first optimisation. This transformation will be done on the inlined program. We make the observation that both summing and finding the maximum of the array are solved using the same strategy, namely by iterating over the array. We are iterating over the array twice, even though we could get away with only doing it once. Now instead of having two separate loops for looping over the array, we can choose to truly "multitask" in only one loop by applying a technique called *loop fusion*. The start of the loop is the same for both methods:

```
for (int e: array) ...
```

So that stays the same. The only thing we do is to 1. make sure that both variables are declared before the loop, 2. add together the code from both loops, and 3. make sure that the print statements come after the loop. The resulting—even *uglier*—program looks like this:

```
public class Main {
   public static void main(String[] args) {
      int[] array = {1,2,3,4,5};
      int s = 0;
      int max = Integer.MIN_VALUE;
      for (int e: array) {
         s += e;
         if (e > max) { max = e; }
      }
      System.out.println(s);
      System.out.println(max);
   }
}
```

You might think that if this program is faster than the previous one, then it is because there is half as much overhead due to the looping itself. Although this is true, it would probably not be the main motivation for doing this kind of optimisation, at least these days. A bigger concern with regards to efficiency is how the program accesses memory. By combining two loops[7] which accesses the same array in the same order, we try to play nice with the memory caching in the machine. If we go through an array from beginning to end and only once, we have done all that we can to make sure that data caching is on our side[8].

[7] This technique is called *loop fusion*. Yes, fusion technology *is* real.

[8] To complicate matters, most modern machines also have an *instruction cache* for the instructions to the CPU. It might very well be that, by having more instructions in this loop, we have been less nice to *this* cache. This is why programmers who are not hardware experts, or who do not feel confident in their ability to simulate x86 machines in their heads, should measure to see if an "optimisation" was really that for a given program.

2.3 motivation

We will further motivate the utility of program transformation by discussing optimising compilers and *interactivity*.

Program transformation and optimising compilers

In the last section we went through examples of program transformations that a compiler might do. They were:

1. Simplification of certain constructs.
2. *Inlining* of the example program.
3. *Loop fusion* on the inlined program.

You might have noticed that there were some ifs and buts that went along with my description of the optimization in the previous examples. Indeed, an *optimisation* is not always that, and can in certain cases actually make the program slower. Now you might think that this weakens my case for motivating program transformation techniques. But on the contrary, I would say that it *strengthens* it. What it does show is a shallow glimpse of the limitations of general purpose optimising compilers. Now, recall that this chapter is about *program transformations*, and that compilers are just an example of that. Optimising compilers are in widespread use today, and programmers that use them do not need to learn how they work in order to get good code back from them (most of the time). You simply feed it a high-level program, and it churns out a fast executable. But what if your compiler cannot make your specific program fast? What if you need to make a slight bug fix, and an unintended consequence of that is that your program becomes slow to the point of being useless?

> At this point many readers will say, "But he should only write P, and an optimizing compiler will produce Q". To this I say, "No, the optimizing compiler would have to be so complicated (much more so than anything we have now) that it will in fact be *un*reliable." (Knuth [1974])

There are perhaps two primary ways that slow compiler-produced code is dealt with:

1. Try to change the code to be more "compiler-friendly", i.e. easier to optimize. This might involve consulting some resource on how the compiler optimizes the program, and looking at the target code (e.g. assembly).
2. Rewrite the code in a "faster" language (easier to write efficient code in).

The downside with the first approach is that the programmer now has to care about how the compiler works, specifically the optimizer. The point of an optimising compiler is to, to the best of its ability, do the optimising on behalf of the programmer. Another problem is that these changes might not work between different compilers and compiler versions. There is a more insidious problem associated with this, other than the obvious problem of not always being able to do the job of optimisation to a satisfactory degree (we might forgive the program for that). That problem has to do with the uncertainty that always comes with adding a *layer of abstraction*. This is an uncertainty for the programmer about whether the layer at which the programmer is working can be trusted, or not. (And if he is *too trusting*, he might be in for a rude awakening some day.) In this case he might for good reason trust the correctness of the generated code. But he might not have enough reason to trust the execution speed or memory use of the generated code. The insidious part is that there are no automated alarm bells that go off when the optimizer fails to do its job. Instead, the programmer has to pay a sort of price for using this convenience of an optimising compiler. That price is constant vigilance that the program might not be fast enough, or use only a reasonable amount of memory. Contrast this with using a language that is easier to optimize. This language will probably be lower level, and so will demand more work or skill of the programmer. But what the programmer has to pay in the form of extra implementation effort, he might in some cases more than make up for in the form of having to be less vigilant of external tools failing to live up to his expectations.

Note 1. This is one of the reasons for why some programmers prefer to work in lower-level languages rather than higher level ones.

Note 2. Put another way: in some cases it might better to wrangle with the intricacies of C code, than to wrangle with the intricacies of some Java Virtual Machine. Perhaps especially if you only have to develop for one architecture.

The downside with the second approach—rewriting the code in another language—is that you have to use another language in your project. This is not only a learning- and knowledge-burden for the programmers on the project, but also a burden on the project itself, since another language has to be integrated to work with everything else. Moreover, you might have a fairly limited range of languages to choose for your secondary lower-level language. This has to do with integration as well, more specifically integrating with running alongside or in concert with the higher-level language. This can be harder to achieve than it might sound like, because you have to deal with the following issues, among others:

- How to call functions written in the first language from the second language, or the other way around.

- If a garbage collector is involved, one must make sure that both languages let the garbage collector own and manage the memory that it is responsible for, instead of for example freeing it prematurely.
- How one should deal with exceptional conditions that might happen during the execution of one of the languages, especially if they have different mechanisms for handling exceptional conditions.

In short, in order for a language to call code written in another language, it needs some kind of *foreign function interface* (FFI). And given the problems that an FFI needs to deal with, the range of languages provided for you might be limited. A language that many other languages can call into is C.[9] C might be a good programming language to use for some cases and domains. But if you are mostly writing your application in a high-level language, there are several downsides with using C. One of them is that C has little in the way of abstraction facilities, which might hurt ones productivity. Another problem is that C is very error-prone, in the sense that it is easy for the programmer to make mistakes in it compared to many other languages. Not only does this cause bugs which cannot happen or are harder to cause in higher-level languages, but it has security implications as well. If you are not careful, you might leave your application open to security exploits because of for example improperly handling memory. All of the above might be a steep price to pay if you originally planned to just program in a very high level language, like for example Python or some Lisp language.

To reiterate, the two answers to the problem are:

1. Make more compiler-friendly code.
2. Rewrite the code in another language.

The improvements that program transformations provide to the above are, respectively:

1. Use program transformations directly, instead of indirectly through compiler-friendly code.
2. Use program transformations for the same language, instead of a whole new language.

And these two approaches provide *directness* and *uniformity*, respectively. *Directness* means that you manipulate the code as you want it to end up more directly. So instead of *indirectly* producing the code that you want by

[9] The reason for that is basically that C is the systems language of the widely-used Unix family of operating systems, and implementations for it exists for a very wide range of architectures. It has established itself as the *lingua franca* of the programming world, in the sense that it is widely used and can be integrated and called from a lot of other programming languages.

trying to coax the compiler into producing the correct code, you deal with the program transformations more directly yourself. After all, if you intend for your high-level code to be optimized in a certain way by the compiler, you are trying to force a certain optimisation transformation to happen. And if you actively have to consider that transformation and make the compiler produce it instead of delegating the whole task to the compiler, it might be better to forego the compiler for that task and just deal with the transformation directly yourself. *Uniformity* means that you just deal with one language, instead of having to switch between a high-level and a low-level one. The relevant insight here is that if you are switching to a lower-level language because the compiler failed to do some optimisation transformation, then the problem is with the compiler, not with the high-level language not being low-level enough. Thus you can avoid the low-level complications of a language like C, which was discussed above.

> I've tried to study the various techniques that a hand-coder like myself uses, and to fit them into some systematic and automatic system. [...] We always found ourselves running up against the same problem: the compiler needs to be in dialog with the programmer; it needs to know properties of the data, and whether certain cases can arise, etc. And we couldn't think of a good language in which to have such a dialog.
>
> For some reason we all (especially me) had a mental block about optimization, namely that we always regarded it as a behind-the-scenes activity, to be done in the machine language, which the programmer isn't supposed to know. This veil was first lifted from my eyes in the fall of 1973, when I ran across a remark by Hoare [42] that, ideally, a language should be designed so that an optimising compiler can describe its optimizations in the *source* language. Of course! Why hadn't I ever thought of it? (Knuth [1974])

Feedback

Program transformations provide more feedback when programming. In a way, that is not very hard to do, since a lot of modern programming leaves a lot to be desired on that front. This statement might be a bit controversial, so let's make an analogy with something else; text editors.

Some decades ago, computers weren't very interactive, and feedback was not instantaneous. This had to do with computer speed and limited peripherals; you can imagine that it is hard to interact with a computer if the output has to be physically printed. Things that we now take for granted, like typing stuff on a keyboard and immediately seeing them on the screen, were not a reality to people who used teletypes. Ed (or stylized `ed`) was a text editor for the first version of Unix. In fact, it is still installed on many Unix-like systems to this day, probably to the delight of the most survivalist-inclined

system administrators. Due to it being the standard editor for an old version of Unix, it got the following tagline, which comes across as cheeky when read in the modern day:

> Ed is the standard text editor.

Ed is what you might call a command-driven editor. You issue commands in order to view or change the file. Yes, that's right; you have to issue commands to view the file, separate from changing it. Recall the limited peripherals at the time; being able to both view and manipulate a file was not really feasible. When you open a file with Ed you are greeted with this view (the bar indicates the cursor):

```
0
|
```

From here you give commands like *move* (`m`), *delete* (`d`), and *append* input to file (`a`). If you fail to give a command that Ed understands, it will give you this obtuse response:

```
?
```

My guess is that verbose error messages were not suited to the typical output devices at the time.

In the mid-seventies, the text editor Vi (or stylized `vi`) was released. "Vi" is an abbreviation for "visual". Can you guess what it does compared to the original version of Ed? Exactly; it allows you to view the text that you're editing (it's a so-called *screen-oriented* editor).[10]

Nowadays, being able to edit files while you are also seeing them is a given. Of course, there are always exceptions—maybe there are people who would like to pursue some kind of sublime minimalism to the point that viewing the rest of the file while writing the next paragraph becomes too distracting.[11] But it wouldn't be an exaggeration to say that interactive—i.e., with immediate and whole-file feedback—editing is what the vast majority of us expect. Now let's compare this to how we interact with programming languages. Some languages afford more feedback than others. Dynamically typed languages certainly provide a good level of interactivity, in that they don't take that long to compile (if they compile to some target code at all). Statically typed languages tend to be more *unwieldy*, in that they often have a more involved compilation pipeline, and often don't have interactive tools like *read eval print loop (REPL)* programs. The workflow that compiled languages often promote is sometimes called *edit-compile-run*. This bears some similarity with the workflow that you

[10] Vi's modern heritage is that it pioneered the still-popular style of *modal editing*, but the visual aspect of it is more relevant to us here.

[11] The text editor *Ghostwriter* comes close to this with its *focus mode*. In that mode only the current line, sentence, or paragraph is highlighted, which goes some of the way towards (or back to) the style of Ed. It also has a *Hemingway Mode* where you cannot use the backspace or delete keys; in case simulating teleprinters is too new-school and you would like to simulate typewriters instead.

can imagine Ed promoted. If we compare the *editing code* step to the step of *editing text* in Ed, and the *running code* step to the *display text* step in Ed, we can begin to see the relevant similarities. Although Ed was of course used to edit code, if we for now just focus on the resulting text as the artifact that one is after when using an editor, the artifact of the *edit-compile-run* workflow is the (running) code. The problematic part is not the fact that you have to compile the code. Rather, the problematic part is the gap between sending the code to the compiler, and then running the code. Just like *line-oriented* editors made the contents of the file kind of opaque to the the user, modern tooling around compilers leaves something to be desired when it comes to figuring out what it is doing.

The issue is not with peripherals, or with computer speed. A compilation can take a fraction of a second and yet yield results that are opaque to the programmer. Nor is it with slow iteration on code changes, per se. The issue is the same as we have touched on before; a compiler is often used as a black box, and might only be understood indirectly through things like documentation.

We have already discussed the *black box*-nature of the traditional compiler; the program transformations that it does internally are completely opaque from the perspective of the user, i.e. the programmer. This is similar to how Ed makes the contents of the file opaque. But with Ed, you can at least ask it to show you some of the file. With traditional compilers, there is no such mode of interrogation. Instead, you have to reverse engineer what happened by looking at the output target code of the compiler. This is certainly a form of feedback, although it is an indirect kind of feedback. Worse, though, is when you have to look up some reference and use your judgment to guess what the compiler will produce. There is no feedback with the language tooling involved here; only with your own mind. But you can of course recover some feedback by trying to challenge your own guesses, by looking at a profile of the running target code, or looking at the output target code.

Now let's compare this approach to using program transformations more directly. In one sense nothing change, since you are still dealing with transformations just like the compiler. In another sense, this is a profound change. Instead of having to guess what some opaque tool does, then second and third-guess, you deal with what you are after directly; a series of transformations on the code.

2.4 COMPARISON WITH OTHER APPROACHES

Program transformation is an old subject. It goes back to the 1970s, at the least. Already then, a lot of great ideas about this topic was being proposed. In fact, even though we have faith in our own approach to this subject, we will probably not manage to propose anything novel compared to this time period[12]

[12]Recall what was said in the introduction: "we all know that all great ideas are not original".

There is a certain range of goals and focus in these papers. Some focus more on automated transformations while others focus on user-guided transformations, for example. Based on our own needs we have come up with what we need from a transformation system. We are going to contrast our own needs with what some of these papers focused on. We will also look at transformation concepts which are implemented in some programming languages, and also compare these to our own needs.

Programming Languages

The premiere example of transformations implemented in programming languages are the Lisp family of languages. These languages use transformations in the form of *macros*, which are functions on code that effectively expand to regular code before the program is run. But it is not macros alone that make Lisp languages into successful transformational languages. What is most-widely credited for their usefulness on this front is homoiconicity. Homoiconicity is a property of the concrete syntax of a language which makes the code easy to transparently treat as data. In program transformations, what is being transformed is code, and so code is treated as data. To the uninitiated, Lisp just looks like a mass of parentheses (here Scheme):[13]

```
(define (factorial n)
  (if (<= n 0)
      1
      (* n (factorial (- n 1)))))
```

But it is exactly this uniformity that makes it so suitable for transformations. Scheme's concrete syntax is just a thin veneer over its abstract syntax. Very briefly, a Scheme program conists of *s-expressions*, and s-expressions are a notation for nested lists. Since nested lists can contain different types (Lisps are typically dynamically typed), this nested list is a tree structure, which among other things can store Lisp code itself.

Scheme and the wider Lisp family has a very different approach compared to Magnolia. While Scheme is homoiconic, Magnolia has a syntax in the lineage of C, and also the Pascal lineage. The approach to between these groups of languages goes deeper than technical details. Lisp seems to be relatively alone in its approach to uniformity of syntax. Some programmers coming from other traditions and families might think that Lisp is too minimalistic when it comes to syntax, and that it makes programs harder to read. So there is a cultural element to these different programming worlds. Another difference is that Scheme is dynamically typed, while Magnolia is statically typed. Moreover, Magnolia goes further than many statically typed languages and avoids features that makes the code harder to statically analyse, such as higher-order functions and pointers. What these differences amount to is that

[13] Code is taken from Rosetta code [sch], licensed under GNU Free Documentation License 1.2.

Scheme's motto of code as data is inappropriate for Magnolia. We cannot be as nimble as Scheme and intermingle regular code with transformation code seamlessly. Instead, we will have to use interfaces and languages that are separate from Magnolia to operate on Magnolia code; more on this in a later chapter.

The research effort behind Magnolia has some background in C++, and it is partly inspired by that language. C++ in turn came about as an extension of C in 1980s. C has a very crude facility for program transformations in the form of the *C preprocessor*. The C preprocessor is what is run first on the C source code, and does things like inclusion of header files, macro expansion, and conditional compilation. Macro expansion is in principle similar to how Scheme's macros work, but are much more primitive. They only operate on text (strings) and not on the structure of C programs, so the output they give can lead to errors in the resulting code if the programmer is not careful. An example of a macro is defining constants:

```
#define PI 3.14159
```

All occurrences of the string `PI` in the program will be expanded to `3.14159` after preprocessing. Thus, these macros allow for the most basic form of constant folding, in effect.

C++ also uses the C preprocessor. But C++ provides a more powerful facility for program transformations in the form of *template metaprogramming*. This is a style of programming which uses the template facility of the language to write code that will be executed at compile-time. What makes this different from preprocessor macros, and similar to Scheme's macros, is that templates operate on the structure of C++ programs. As previously stated, preprocessor macros just operate on strings and know nothing about the semantics of C or C++. What makes C++ templates different from Scheme macros is that the template language inside C++ is very different[14] from what you might call the *value language* of C++ (code that is to be executed at runtime). While Scheme code is simply manipulated using Scheme code, compile-time programming to generate C++ code has to use the C++ subset of C++. Some claim that this templating language is hard to use.Sheard and Jones [2002]

Automatic vs. Manual

The degree to which the program transformation is automated varies. Some approaches strive to fully automate transformations. Other approaches use a semi-automatic approach where regular, manual programming is interspersed with automated program transformation. These automated program transformations might be what me might call deterministic transformations, in the sense that the transformations are functions that give the same output when

[14] As noted in Robison [2001]: "The list above makes obvious to functional programmers what the committee did not realize until later: templates are a functional language, evaluated at compilation time."

given the same input. The use of macros in Lisp languages might be an example of this. In contrast, non-deterministic transformations use internal heuristics that are opaque from the point of the user, and so are not well-defined functions. Our proposed program transformation system uses the semi-automatic approach.

Specification and correctness

There is a lot of focus on specification and proving the correctness of code in the programming language research literature. If we were to sum up the ideal which was pursued:

Definition 1 (Correctness ideal)**.** Software should be meticulously specified, implemented, and then proven to be correct according to the specification.

Many of the program transformation papers from the 1970s dealt with specification and correctness in some way, or assumed it as a part of the software engineering process. We choose to not emphasize correctness or verification of transformations. This is an indirect consequence of the way we use transformations, which is as a complement to manual programming. Transformations are intermingled with manual coding. And so if there are no obligations on the programmer to prove or verify his manually programmed code, there should be no additional obligations for transformations. Magnolia is in part a specification language, so specification and verification is available to the programmer—irrespective of whether he uses transformations or not.

In the introduction to Bauer [1976]:

> More precisely: Programming as a process can be done in an orderly way, step by step, applying rules safely - under control of intuition. This has been demonstrated by the discipleship [...] of 'Proving program correctness'.

Bauer [1976] describes an "evolutionary" process of program construction; start with a high-level, "mathematical" approach, and then work your way towards an efficient implementation. This is a theme in many program transformation papers. The paper doesn't emphasize proofs of correctness as the top priority, but it thinks it is of great value:

> *It is more important to be able to derive a correct program than to prove its correctness, although the moral value of a direct correctness proof should not be neglected.*

Balzer et al. [1976] implicitly embraces specifications and proofs by stating that their proposed system would deal with proofs of *equivalence preservation*[15]:

[15] If a transformation is *equivalence-preserving* it doesn't change the semantics of the code. An example of this is an optimization.

> The TI [Transformation Implementation] approach would eliminate the implementation proofs required in the Levels of Abstraction approach, which are difficult to specify and construct. In TI, the functional equivalence of each transformation would be proved once and for all before it is entered in the catalog, and thus would be of no concern to the programmers using the transformation.

Manna and Waldinger [1979] is about using program transformations in the context of verification systems. They propose a way to develop program and its correctness proof "hand in hand". So this is another example of a paper focusing on specification and correctness.

Feather [1982] makes note of "assisting program description and verification" as a potential application of program transformations, in addition to the application of assisting software development. The paper adopts Burstall and Darlingtons's transformation method [Burstall and Darlington, 1977]. Burstall and Darlingtons's transformation method was a semiautomatic program transformation system for transforming first order recursion programs.

2.5 THE FLUX OF CODE

Everything is in a state of flux. It is obvious enough that night follows day. It is obvious enough that spring follows winter. It can be less obvious that even mountains are not permanent fixtures. That they too will disappear in time. That something else will take their place. In the same way, code has a more *pliable* or *mouldable* quality than might be obvious at first sight.

Any programming involves sending around values and variables. These are the "flexible artifacts" that the programmer first encounters. Most everything else feels like scaffolding, structure, bricklaying—carefully and tediously erected fixtures. But if the programmer looks with a keen eye at the rest of the code, he might eventually pierce through the illusion of the fixedness and rigidness of code.

- Functions don't have to be fixtures; they can be regular values. (*Higher order functions.*)
- Types don't have to be fixtures; they can be regular values. (*Dependent types.*)
- A compiler doesn't in principle have to be written manually; it can be derived from an interpreter using *partial evaluation.* (*The second Futamura projection [Futamura, 1999].*)

It is not just the *data* that code manipulates that is pliable. The code itself is mouldable and pliable. As program transformation shows us.

3

Magnolia

We give an introduction to the Magnolia programming language.

3.1 OVERVIEW

Magnolia is a general-purpose research programming language. The quickest way to describe it might be to compare it to most mainstream imperative programming languages. First of all, *specifications* is important in Magnolia, and has first-class support. Second of all, the semantics of the language is designed to achieve two goals:

1. To be easy to analyse.
2. To not put restrictions on the underlying implementation or architecture.

One of the most important examples of the first point is that Magnolia has no concept of a *pointer* or *reference*. This simplifies analysis, since the possible presence of pointers in programs complicates analysis. To the second point, although Magnolia is a general purpose programming language, it has been put to most use on problems in the domain of HPC. Unlike domains like Web programming or mobile development, in HPC a lot of different hardware and architectures are used in order to solve computationally demanding problems. As an example, the array data structure can often be assumed to be represented as a contiguous chunk in memory when doing most programming tasks. But in HPC, the array might in fact be distributed among several memory representations, and be physically separated from each other to the point that a computation across the array can be called a distributed computing. We can contrast this approach to the C programming language. The C specification under-specifies a lot of behavior in order to allow compiler writers flexibility, and allow C to be implemented on a wide range of architectures.[1] Despite that, contemporary implementations of C have converged on a memory model similar to PDP-11 [Chisnall et al., 2015, p. 1], which was the original target architecture for C. The assumptions and limitations of C—and

[1]This is done by explicitly labelling certain cases as undefined behavior. The unpredictability born out of that has lead to a lot of surprises for C and C++ programmers. Magnolia does not use the concept of undefined behavior in its semantics.

also C++—makes it sub-par for the domain of HPC. In order to target different architectures, one should be able to express code that can translate well to those different architectures, and not just machines that behave similarily to the C abstract machine. If you cannot do that, you might be forced to implement things directly for each architecture.

Another limitation of languages like C and C++ is their imperative nature. Although imperative programming is often viewed as the paradigm that leads to more efficient programs very reliably, this is too simplistic. First of all, even if imperative programming might lead to more efficient programs given enough time, time is always limited. In some cases, using a different paradigm to solve a programming problem might lead to code that is more maintainable and amendable to change, which might in turn mean that the code can be more easily optimized than would be the imperative solution to the same problem. Second of all, an imperative program is simply not a universally efficient implementation for all execution environments. Imperative programming lends itself well to serial problems, but less so to parallel problems. Moreover, an imperative solution to a problem might easily over-specify the solution to the problem. What this means is that the implementation is too specific, in the sense that it leaves too little room for things like architecture-specific optimizations. For example, the problem of transforming a one-dimensional matrix by adding some number to each element can be solved iteratively by looping through an array that represents the matrix and for each element adding the given number. The potential problem with this is that it over-specifies the order of iteration. It demands that the addition operation is to be done on each element in turn. But such operations can often be done more fruitfully in parallel. This might not seem like too much of a problem for one mapping over a matrix, but for a large composition of mappings and on sufficiently large matrices, it can definitely be a problem. Unlike C—but similar to modern versions of C++—Magnolia is designed to accomodate both imperative a declarative programming [Bagge and Haveraaen, 2010].

3.2 interface declarations

What is in the literature is called *interfaces* or *signatures* is at the core of Magnolia. A signature is a collection of operations (function, procedure, and predicate declarations), along with a collection of types. Magnolia has concepts, which are signatures extended with axioms. Concepts are similar to things like Java's interfaces and Haskell's typeclasses. Axioms are a distinguishing feature, which allows the programmer to declare invariants. Axioms consist of *assertions* which have to hold for any implementation of the concept. Axioms act like declarations of algebraic properties. More relatable to most programming practice might be to regard them as parameterised unit tests; unit tests that have to hold for any instantiation of the input. In any case, axioms is a specification mechanism for Magnolia, and is used to test implementations of concepts.

As an example, this concept describes a linear abstract data type:

```
package Collections.Linear;

concept Linear = {
  type Data;
  type Linear;

  function peek ( l:Linear ) : Data;
  function pop ( l:Linear ) : Linear;
  function push ( l:Linear, d:Data ) : Linear;
  function clear () : Linear;
  predicate isEmpty ( l:Linear );
  predicate isFull ( l:Linear );

  axiom pushNotIsEmpty ( l:Linear, d:Data ) {
    assert !isEmpty(push(l,d));
  };
  axiom clearIsEmpty () {
    assert isEmpty(clear());
  };
  axiom clearPushPeek ( d:Data ) {
    assert peek(push(clear(),d)) == d;
  };
  axiom clearPushPop ( d:Data ) {
    assert pop(push(clear(),d)) == clear();
  };
};
```

The functions provide ways to add (`push(...)`), remove (`pop(...)`), examine (`peek(...)`), and clear (`clear(...)`) the data structure. Based solely on the names, parameters, and return types of the operations, it should be fairly easy for a programmer to make a guess about how this abstract data type should behave. But for the compiler, given just the types and operations of a concept, the compiler can only catch relatively simple mistakes like returning or passing in a value of the wrong type. This is where axioms enter.

The four axioms for the `Linear` concept goes a long way to capture the requirements that a programmer might have already guessed. If an element is added (*pushed*) to the data structure, it should not be empty:

```
  ...
  axiom pushNotIsEmpty ( l:Linear, d:Data ) {
    assert !isEmpty(push(l,d));
  };
  ...
```

If the collection has been cleared, it should then be empty:

```
...
axiom clearIsEmpty () {
  assert isEmpty(clear());
};
...
```

If a cleared collection has then had an element added to it, `peek(...)` should return that element:

```
...
axiom clearPushPeek ( d:Data ) {
  assert peek(push(clear(),d)) == d;
};
...
```

Clearing, then pushing, then popping a collection, is the same as just clearing:

```
...
axiom clearPushPop ( d:Data ) {
  assert pop(push(clear(),d)) == clear();
};
...
```

None of the axioms for `Linear` specifies the order in which elements are extracted. It could be first in last out (FIFO) or last in first out (LIFO); both would be equally valid implementations. We can use `Linear` to make a queue (FIFO) and to make a stack (LIFO). This is how a stack could be defined:

```
concept Stack = {
  use Linear[ Linear => Stack ];

  axiom pushPop( l:Stack, d:Data ) {
    assert pop(push(l,d)) == l;
  };

  axiom pushPeek( l:Stack, d:Data ) {
    assert peek(push(l,d)) == d;
  };

};
```

The axioms `pushPop(...)` and `pushPeek(...)` specifies the desired LIFO behavior. Notice the renaming of the type `Linear` to `Stack`. This is how a queue could be defined:

```
concept Queue = {
  use Linear[ Linear => Queue ];

  axiom pushPop( l:Queue, d:Data ) {
    assert isEmpty(l) => pop(push(l,d)) == l;
```

```
    assert !isEmpty(l) => pop(push(l,d)) == push(pop(l),d);
  };

  axiom pushPeek( l:Queue, d:Data ) {
    assert isEmpty(l) => peek(push(l,d)) == d;
    assert !isEmpty(l) => peek(push(l,d)) == peek(l);
  };

};
```

Like for `Stack`, the axioms define the extraction behavior.

3.3 MAGNOLIA OPERATIONS

Magnolia uses three kinds of operations to implement code; procedures, functions, and predicates. Procedures are subroutines which pass both input and output through its parameters. Parameters are distinguished through three different *modes*, according to their purpose and limitations:

1. **out**, which passes output. Cannot be read from inside the procedure.
2. **obs**, which passes read-only input.
3. **upd**, which can pass in data as well as be updated in the procedure, which affects the calling code.

Magnolia functions are pure, so they do not have side-effects. Together, procedures and functions allow for both imperative as well as functional-style programming. Recall that both imperative and expression-oriented (functional) styles are useful for the domain of HPC, since one needs to be able to use different levels of abstraction.

Magnolia predicates are functions that either return a boolean value. But due to the underlying theory of the language and the role of predicates in it, predicates have a special status in the language.

4

Partial Evaluation

We describe what the transformation technique partial evaluation is and how it works.

4.1 INTRODUCTION

Partial evaluation is a source-to-source program transformation technique for specializing programs with respect to parts of their input [Consel and Danvy, 1993, p. 1]. It is based on and leverages the fact that programs some times have input arguments whose values are known before executing the program. In other words, parts of the input to the program are *statically known*. The simplest case is when a program, or a program-fragment, only has static input and the program is side-effect free; an example of this is an arithmetic expression involving only constants. In that case, the whole program can be evaluated *statically*, i.e., at compile-time. This special case of partial evaluation is known as constant folding, and the output is whatever the program evaluates to, such as a number in the case of an arithmetic expression. When only parts of the input to the program are statically known, the output of partial evaluation is a specialized program, hence why partial evaluation is considered a form of program specialization [Jones et al., 1993, p. 2]. The motivation behind partial evaluation is program optimization [Jones et al., 1993, p. 5].

The term *program specialization* is by some authors used interchangeably with "partial evaluation"[1] [Jones et al., 1993, p. 1]. According to Jones et al. [1993, p. 367], Andrei Ershov introduced the term *mixed computation* to mean roughly the same as partial evaluation [Ershov, 1977]. Also according to Jones et al. [1993, p. 369], Komorowski suggested the term *partial deduction* for partial evaluation of pure logic programming languages [Komorowski, 1989].

[1]Though using these two terms as synonyms in a wider context might be unwarranted. Reps and Turnidge [1996] shows that the program specialization done by program slicing can in some cases not be achieved by partial evaluation or other techniques.

What

Consider a program p and its input in, which might consist of one or more variables. Evaluating p applied to in yields an output out, which we might denote as:

$$[\![p]\!]in = out$$

Note that, since program specialization involves treating programs as both programs and data, we adopt the convention from Jones et al. [1993] of denoting the execution of programs by $[\![\_]\!]$.

It might be the case that parts of in are statically known, which means that we know their values ahead of executing p. The rest of in is considered *dynamic*. Call the static part of in s and the dynamic part d. Given that we have a partial evaluator—which we will call mix—we can partially evaluate p with s as its static input:

$$[\![mix]\!][p, s] = p'$$

p' is a program, and in particular a *residual program* [Jones et al., 1993, p. 71]. Evaluating p' applied to d—the remaining inputs—yields out, i.e., $[\![p']\!]d = out$. In other words, evaluating p by first partially evaluating it with the static input, and then evaluating the resulting residual program with the rest of the input gives the same output as evaluating p directly.

$$\begin{aligned} out &= [\![p]\!]in && (4.1)\\ &= [\![p]\!](s \cup d) && (4.2)\\ &= [\![[\![mix]\!][p, s]]\!]d && (4.3) \end{aligned}$$

Notice that evaluating p via partial evaluation involved two *stages*, namely first partially evaluating p and then evaluating p'. This is in contrast to evaluating p directly, which only consists of one stage of computation. This is called *multi-stage programming* [Taha and Sheard, 1997] or *multi-stage computation* [Jones et al., 1993, p. 7]. With that in mind, we might more succintly refer to evaluating p via mix and p' as *two-stage evaluation of p via partial evaluation*.

Examples

The following example is adapted from Jones et al. [1993, p. 3].

Below is an implementation of the *power* function.

```
f(x,n) = if n == 0 then 0
       else if even(n) then f(x,n/2)^2
       else x * f(x,n-1)
```

We might have some code in which one or more of the arguments are static. Let us first consider the case in which the exponent is statically known, say like `f(x,5)`. The first observation we will make is that all conditionals in the definition of the function only depend on knowing the value of `n`. In particular, we have **`if`** `n == 0`, and **`if`** `even(n)`. This is what we get if we substitute all occurrences of `n` with `5`:

```
f(x,5) = if 5 == 0 then 0
       else if even(5) then f(x,5/2)^2
       else x * f(x,5-1)
```

As we can see, the two conditionals get specialised to **`if`** `5 == 0` and **`if`** `even(5)`, both statically computable. The second observation is that we can use our first observation to eliminate all of the recursive calls in the function definition. This can be achieved by applying techniques like *symbolic computation* on the conditional expressions, and *unfolding* the functional calls `f(x,n/2)`, and `f(x,n-1)`. What we end up with is a specialized function of arity one, which we will call `f5(x)`:

```
f5(x) = x * ((x^2)^2)
```

Now that we have considered the case in which the exponent is the statically known value, we will next consider the case in which the exponent is dynamically known while the base is statically known. As an example we will use the function call `f(3,n)`. Unfortunately, there are less opportunities for partial evaluation in this case. There are few operations on `x` which can be done by the partial evaluator, since all expressions involving `x` also depend on knowing what the value of `n` is. It seems the best and only thing we can do in this case is to inline the value of `x` into the function definition and remove the `x` parameter:

```
f_base_3(n) = if n == 0 then 0
       else if even(n) then f(3,n/2)^2
       else 3 * f(3,n-1)
```

But now we can not seem to specialise the code any further. This is an example of how just knowing some static parameters might not be enough to get any meaningful optimization out of partial evaluation techniques. In fact, programs might need to be written or structured in certain ways in order to be able to be easily specialized [Jones et al., 1993, p. 16].

The next example is adapted from Consel and Danvy [1993, p. 1].

Many programming languages have functions for producing formatted text. Examples include `format` in Lisp, and `printf` in C. The parameters of such formatting function consists of a control string and a varying number of values. The function outputs text by interpreting the control string in order to determine how to format the list of values. What makes this suitable for partial evaluation is that the value of the control string is, in most cases, statically known. This means that we can try to eliminate the interpretive overhead of reading the control string at runtime.

Let us consider a formatting function in Scheme This function only handles three formatting directives: `~N`, `~S` and `~%`. The first two directives specify that the corresponding element in the list of values must be printed as a value or as a string, respectively. The last directive is to be interpreted as an end-of-line character.

```
;; vs is a list containing two values
(format.1 vs) = (format "~N is not ~S~%" vs)
```

```
;; vs is a list containing two values
(define format.1
  (lambda (values)
    (write-newline
      (write-string
        (write-string
          (write-number-port (car values))
          " is not ")
        (car (cdr values))))))
```

Above is some Scheme code that uses the `format` function, and a specialization of the invocation of the `format` function. In this case, all the operations manipulating the control string have been performed at compile-time. No references to the control string are left in the residual program. The specialized function only consists of operations manipulating the run-time argument, namely the list of values to be formatted.

How

Conceptually, partial evaluation encompasses both compilation and interpretation of the target program; partially evaluating a program with respect to all of its inputs amounts to running this program and producing a constant residual program. This means that a partial evaluator must include a interpreter to construct that residual program. And partially evaluating a program with respect to none of its input amounts to producing a (possibly simplified) version of this program. So a partial evaluator must also include a compiler to construct the residual program [Consel and Danvy, 1993, p. 3].

We can describe how a partial evaluator works by listing how it deals with different kinds of levels of *dynamicness*, so to speak [Consel and Danvy, 1993, p. 3]:

- Expressions only depending on static data: evaluate the expression (constant folding).
- Propagate constant values, including the ones that the partial evaluator manages to generate.
- Function calls (that can't be fully evaluated statically): specialize function definition for that input. A *monovariant* specializer produces at

most one specialized function for every source function. A *polyvariant* specializer can produce many specialized versions of a source function. The specializer may choose to unfold—i.e. inline—specialized function definitions.

There are two categories, or approaches, to making partial evaluators: online and offline partial evaluators [Consel and Danvy, 1993, p. 3]. Both have been actively researched, and each of them provide their own benefits and downsides.

An online partial evaluator can be viewed as a one-stage, or one-pass, partial evaluator, since the treatment of each expression is determined on the fly. Online partial evaluators are in general very accurate, but they tend to have a considerable interpretive overhead [Consel and Danvy, 1993, p. 3].

In contrast to the one-stage partial evaluators, an offline partial evaluator is multi-staged in the sense that it does its work through several distinct stages. An offline partial evaluator can be divided into two stages, namely a preprocessing stage and a processing stage.

The preprocessing stage often includes a binding-time analysis (BTA). BTA consists of analyzing which parts of the program can be evaluated statically, and which parts that have to be evaluated dynamically [Consel and Danvy, 1993, p. 3]. It starts by looking at which parts of the input to the program are statically known, and then uses that information to propagate this information through the program, determining for each expression whether it can be deduced that it can be evaluated statically or not. The processing stage then uses the information gathered from the preprocessing stage, like the BTA, to guide it in performing the specialization of the program.

We can think of BTA as annotating every expression in the program with one of two values; S if the BTA can determine that the expression can be evaluated statically, D otherwise. For example, in the expression $x + y$, if x is annotated as D and y is annotated as S, we will have to annotate the whole expression as being D.

We have mentioned that monovariant specializers produce at most one specialized function for every source function, while a polyvariant specializer can produce many specialized versions of a source function. As an example, recall the specialization of the power function `f`; we created a new specialized function definition called `f5`. We can imagine that we had a program with multiple calls to the `f` function, each one with a distinct and static power-argument. Then we might end up with a specialized program for each of these function calls, for example called `f6` (specialization of `f(x, 6)`), `f9`, `f13`, and so on. This is an example of polyvariant specialization. In contrast, a monovariant specializer only creates one specialized function definition for each function definition in the source program. Of course, a partial evaluator can choose to use both mono- and polyvariant specialization for different parts of the program [Hatcliff et al., 1998, p. 3].

Polyvariant specialization is conceptually similar to *monomorphisation* from generic programming. Monomorphisation is a technique for implementing generic functions. If a function `g` takes as argument a generic type `T`, then a compiler which uses monomorphisation will generate a specialized function for each invocation of `g` with a distinct type `T` in the source program. For example, if we have a function `head` which extracts the first element of a list, then we can make it polymorphic in the type `T` of the elements of the list. Then if we have a source program which uses this function with lists of type `Int`, `String`, and `Float`, the compiler will make three functions specialized for each of these types. This implementation of generic functions can be viewed as a special case of partial evaluation; consider the input type parameter of a generic function as arguments to the function that are guaranteed to be statically known.

Online and offline partial evaluation can be combined. For instance, whenever the exact binding-time property of an expression can be determined, offline partial evaluation is used. Otherwise, the treatment for this expression is postponed until specialization-time, when concrete values are available [Consel and Danvy, 1993, p. 3].

4.2 background

A mathematical one-argument function can be obtained from one with two arguments by *specialization*, i.e., by setting one input to a fixed value [Jones et al., 1993, p. 1]. In mathematical analysis this is called "restriction" or "projection", and in mathematical logic it is called "currying". Closer to computer science, partial evaluation was shown to be computable by Stephen C. Kleene in 1943, through the s_{mn} theorem [Jones et al., 1993, p. 1] [Kleene et al., 1952]. However, Kleene's results are not directly useful for applying partial evaluation for program optimization, since Kleene was not concerned with program efficiency. Indeed, his construction gave specialized programs that were *slower* than the original program [Jones et al., 1993, p. 1].

Yoshihiko Futamura investigated the relationship between interpreters and compilers in the context of partial evaluation in 1971 [Futamura, 1999]. He found that partial evaluation can in principle be used to generate target programs,[2] compilers and even compiler generators. Generation of compilers involved self-application of the partial evaluator, while generation of compiler generators involved double self-application. These results have later become known as *the Futamura projections* [Jones et al., 1993, p. 13].

Ershov independently discovered the principle of compiler generation via self-application later in the 70s [Ershov, 1977]. Ershov also gave two comprehensive overviews of the activities in the field of partial evaluation and mixed computation, including overviews of the literature up until that time [Jones et al., 1993, p. 367] [Ershov, 1978] [Ershov, 1982]. Futamura gave another overview of the literature [Jones et al., 1993, p. 367] [Futamura, 1983].

[2] In a broad sense; not just programs written in machine code or other low-level code.

As has been mentioned, it was discovered in the 70s that partial evaluators could in principle be applied to themselves. However, it was not until 1984 when the the first self-applicable partial evaluator was implemented. It was written in a language of first-order recursion equations (or first-order statically scoped pure Lisp), and was used to generate toy compilers and compiler generators [Jones et al., 1993, p. 367] [Jones et al., 1985] [Jones et al., 1989] [Sestoft, 1986]. Due to an increased interest in partial evaluation at the time, the first Workshop on Partial Evaluation and Mixed Computation (PEMC) was held on October 1987 in Denmark. The workshop was the first event to bring together a substantial number of partial evaluation researchers from all over the world [Jones et al., 1993, p. 367].

5

Slicing

We describe what the transformation technique slicing *is and how it works.*

5.1 INTRODUCTION

Program slicing is a program transformation technique that tries to eliminate (*slice*) irrelevant code from a program. Slicing was first presented by Mark Weiser in Weiser [1981]. The technique as he presented it was a formalization of how he observed students approached debugging [Weiser, 1981, p. 439]. The original slicing technique dealt with imperative programs, and so slicing was implemented by deleting those statements that were found to not be needed according to some criterion. Since Weiser's original paper, a large number of papers have been published on different forms of program slicing, algorithms to compute them, and applications to software engineering [De Lucia, 2001, p. 144].

Weiser classified the technique as being useful in maintenance rather than design, since the technique is applied after the program is written. Keep in mind that slicing as originally presented by Weiser dealt with executable programs, and moreover it was demanded that a successful slicing produces an executable program as well. But later formulations and implementations for slicing have since been proposed that relax the *executable* criteria [De Lucia, 2001, p. 145]. In particular, definitions of slicing that allow slicing of subsets of the program that might not themselves be executable. An example of this might be a procedure in the program that is not the designated main procedure of the program. This more general view of this program transformation technique might allow this technique to be used in program design as well as in the originally envisioned maintenance phase of the software life-cycle.

As mentioned, the original motivation for slicing was debugging. Since then, a number of other applications have been proposed; parallelization, program differencing and integration, software maintenance, testing, reverse engineering, and compiler tuning [Tip, 1995, p. 124].

5.2 what

A program slice consists of the parts of a program that potentially [1] affect the values computed at some point of interest [Tip, 1995, p. 121]. The *point of interest* is typically specified by a pair; a *program point*, and a set of variables. The parts of a program that have a direct or indirect effect on the values computed at slicing criterion C constitute the *program slice with respect to criterion C* [Tip, 1995, p. 121].

Note that the pair *(program point, set of variables)* is *static* data, i.e., we do not need to know anything about the dynamic behavior of the program to slice the program. Consequently, this definition is called *static slicing* when it needs to be differentiated with or contrasted with *dynamic slicing*. Dynamic slicing extends the pair from static slicing with another point of interest, namely some kind of dynamic information about the program. A *dynamic slicing criterion* is typically a triple *(input, occurrence of a statement, variable)* [Tip, 1995, p. 124]. So the difference between static and dynamic slicing is that static slicing assumes nothing about the input, while dynamic slicing assumes some *fixed* input. There have been developed hybrids of these two slicing techniques, among them a technique called *quasi static slicing* [De Lucia, 2001, p. 146]. The motivation behind such hybrids techniques are applications in which some input variables are static while the behavior of the program must be analyzed when other input values vary. The concept of quasi static slicing is closely related to the partial evaluation program transformation technique [De Lucia, 2001, p. 146]. From the standpoint of applying program transformation techniques for the purpose of program comprehension, combining partial evaluation with program slicing allows to restrict the focus of the specialized program with respect to a subset of program variables and a program point [De Lucia, 2001, p. 146].

Example

We present the canonical static slicing example, first introduced by Weiser in his original paper.[2] The program takes a number `n` from the user and computes the sum of the numbers `1` to `n`, the factorial of `n`, and outputs these numbers to the user.

```
read(n)
i = 1
sum = 0
product := 1
while <= n:
```

[1] As argued by Weiser in his original paper, the problem of finding a minimal slice is undecidable.

[2] You might recognize this as a slight variation on the example from the Program Transformation chapter (2.2). This time we use pseudo code, and we find the sum and the product instead of the sum and the maximum value.

```
  sum = sum + i
  product = product * i
  i = i + 1
write(sum)
write(product)
```

```
read(n)
i = 1

product := 1
while <= n:

  product = product * i
  i = i + 1

write(product)
```

The next program shows a slice of the first program with respect to the slicing criterion `(10, product)`, where `10` is the line containing the statement `write(product)`. All statements which are not relevant to the variable `product` have been *sliced away*. In particular, note that none of the statements which referenced or updated the `sum` variable could affect the value of the `product` variable. So none of those statements were relevant with regards to this slicing criterion.

How

There are three major approaches to program slicing; dataflow equations, information-flow relations and dependence graphs [Xu et al., 2005, p. 2].

DATAFLOW EQUATIONS Weiser's approach used iteration of dataflow equations. More concretely, his approach found static slices by computing consecutive sets of transitively relevant statements, according to data flow and control flow dependences [Tip, 1995, p. 122]. A *data dependence* is in this case a particular kind of relationship between two statements. In particular, a statement j is dependent on another statement i if a value computed at i is used at j in some program execution [Tip, 1995, p. 126]. *Control dependence* is more involved, but an example of control dependence are the statements inside the branches of an **`if`** or **`while`** statement; those statements are control dependent on the control predicate of the **`if`** or **`while`** statement. With a basic understanding of what data and control dependences are in mind, we can describe the iterative algorithm for program slicing introduced by Weiser. This algorithm uses two distinct "layers" of iteration, which can be characterized as follows [Tip, 1995, p. 128]:

1. Tracing transitive data dependences. This requires iteration in the presence of loops.

2. Tracing control dependences, causing the inclusion in the slice of certain control predicates. For each such predicate, step 1 is repeated to include the statements it is dependent upon.

INFORMATION-FLOW RELATIONS Another approach to slicing is the use of *information-flow relations*. In this approach, several types of relations are defined and computed in a syntax-oriented, bottom-up manner [Xu et al., 2005, p. 2]. Slices can then be easily obtained by relational calculus. As an example, let S be a statement, which itself might be a sequence of statements. Let v and v' be variables, and $(v, v') \in \rho_S$ iff the value of v on entry to S may affect the value of v' on exit from S.

DEPENDENCE GRAPHS Yet another approach to slicing uses dependence graphs. In this approach, slicing is solved in two steps; constructing a suitable dependence graph, and then doing the slicing by performing reachability analysis on the graph. The simplest dependence graph is a Program Dependence Graph (PDG). A PDG is a program representation where the nodes of the graph represent statements and predicates, while edges carry information about control and data dependences [De Lucia, 2001, p. 144]. According to Tip [1995, p. 131], Ottenstein and Ottenstein [1984] found that PDGs can be used for slicing of single-procedure programs.

For slicing of multi-procedure programs, Horwitz et al. [1988] introduced the *System Dependence Graph* (SDG). An SDG is an extension of a PDG. The problem that the SDG solves is to represent the new dependences that occur in an interprocedural setting. For example, to slice a procedure `main(...)`, one needs to determine the dependences that variables defined in `main(...)` have across procedure calls. If there is a statement `call p(a, b);` in `main(...)`, `p(upd a: Integer, upd b: Integer)` needs to be analyzed to determine any dependences that might be between `a` and `b`. `a` and `b` might not affect each other directly in `main(...)`, but one of them might affect the other through the call to `p(...)`.

An SDG consists of [Horwitz et al., 1988, p. 3]:

1. A PDG, representing the program's main procedure.

2. *Procedure dependence graphs*, representing the rest of the program's procedures.

3. Some additional edges of two sorts: a) edges that represent direct dependencies between a call site and the called procedure, and b) edges that represent transitive dependencies due to calls.

Horwitz et al. claimed that the chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure. Weiser's original paper on slicing did provide a way to deal with interprocedural slices, but this approach suffered from what Horwitz et al. call the *calling context*

problem. This problem is due to the fact that Weiser's algorithm, after having *descended* into a called procedure, *ascends* to all calls of that procedure—instead of only ascending to the call-site at which it descended.

And due to considering procedure calls that might be irrelevant to a given slice, this means that Weiser's algorithm produces larger slices than necessary; using the phrasing of Horwitz et al., Weiser's slices are *less precise*.

5.3 MAGNOLIA

In this section we discuss how approaches to slicing in the literature can be applied to Magnolia, and what accommodations need to be made for our needs.

Weiser [1981] considered statement-level slicing on imperative programs with simple types,[3] structured control flow, loops and recursion. They call the parameter passing mechanism for *value-result*; the arguments are both copied on entry to the procedure, as well as copied back on exit from the procedure. Horwitz et al. [1988] used a similarly simple language with simple types. Although their slicing criterion is less general than Weiser's, they argue that his slicing criterion is more general than what is often needed [Horwitz et al., 1988, p. 1]. The survey paper Tip [1995] is mostly—if not wholly—about slicing of imperative programs.

Magnolia has structured control flow, no looping nor recursion, and no pointers nor references. Having structured control flow simplifies the flow analysis of the programs. The absence of pointers simplifies analysis, since aliasing is not a concern; aliases make determining flow dependences very hard, even in the limited case of intraprocedural slicing [Tip, 1995, p. 23]. Magnolia has three modes for procedure parameters: **`obs`**, for parameters that can only be read (*observed*); **`out`**, for parameters that can only be written to; and **`upd`**, for parameters that can both be read and written to, and that will have its value copied back on exit from the procedure. **`upd`** corresponds to the value-result parameter passing mechanism in Horwitz et al. [1988]. We are interested in slicing composite types and arrays, and so considerations need to be made that go beyond the case of simple types [Tip, 1995, p. 23].

Although we are interested in slicing imperative programs, this by itself is too limited for our purposes. Papers like Weiser [1981] and Horwitz et al. [1988] consider non-abstract code—code that leaves no generic parameters to fill in, such as e.g. types. In contrast, one of the strengths of Magnolia is the facilities it has for code abstraction. In addition to imperative code, we need to be able to slice concepts and other abstraction facilities. For example, one might want to slice away one of the types of a concept. Clearly, statement-level slicing is not what you want in this case. Slicing things like concepts can also involve slicing other declarations like requirements and uses. And since requirements and uses can come from other concepts, one needs to slice

[3] I.e., types that are not composite (not made up of other types) and are not references. Papers like Tip [1995] call this "scalar variables".

and rewrite across concepts; borrowing the terminology from interprocedural slicing, one might call this *inter-module slicing*.

6

Modulus Group Implementations and Transformations

We go through a hypothetical workflow of applying program transformations in order to derive different implementations of Modulus Group.

6.1 INTRODUCTION

We discuss the Magnolia source file `ModulusGroupCxx.mg`; see Appendix A.

We have three different implementations—mgArgument, mgConstant, mg7—that implement Modulus Group. The Modulus Group implementations provide the type `MG` to do modular arithmetic upon, and functions for correctly performing modular arithmetic. When talking about the base number associated with in the abstract—as opposed to as a concrete type or variable in the implementations—we will refer to it as *base*. The differences between the implementations are:

- mgArgument takes the *base* number as a dynamic argument to construct the type `MG`. Thus one can do arithmetic modulo *base* for any positive value of *base* with this implementation.
- In mgConstant, *base* is a constant. This means that that any **`program`** that builds on this implementation needs to fix *base* to a certain value, and cannot do any other arithmetic except modulo this *base*.
- mg7 fixes $base = 7$.

We are interested in how to translate automatically between these three different implementations of Modulus Group.

Preliminaries

Modulus Group is an algebra which describes *modular arithmetic*. Modular arithmetic is a system of arithmetic on integers where numbers "wrap around" upon reaching a certain value; the *modulus*. An example of modular arithmetic is the *12-hour clock*, the convention for time which divides the 24 hours of a

day into two 12 hour segments. Since the number of the hour does not exceed 12, the time "wraps around" to 1 after 12. For example, 8 hours after 6 in the morning would be 2 in the afternoon. This is called *arithmetic modulo 12*.

Let MG consist of the following:

$$base \in \mathbb{N} - \{0\} \tag{6.1}$$
$$S = \{0, \ldots, base - 1\} \subset \mathbb{N} \tag{6.2}$$
$$plus : S \times S \to S \tag{6.3}$$
$$uminus : S \to S \tag{6.4}$$

Note 3. "uminus" stands for "unary minus".

And let *plus* and *uminus* be defined as:

$$plus(x, y) = (x + y) \mod base \tag{6.5}$$
$$uminus(x) = \begin{cases} 0 & \text{if } x = 0 \\ base - x & \text{otherwise} \end{cases} \tag{6.6}$$

Where $\mod : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is an infix operator that returns the remainder of dividing the first operand with the second operand.

Motivation

We will here describe a workflow which leverages being able to automatically translate between the three different Modulus Group implementations.

We have a programmer that wants to implement this algebra. He chooses to implement the equivalent of mgConstant. But after having done that, he realizes that he needs to be able to use several different Modulus Groups, i.e., Modulus Groups with different base numbers. But mgConstant does not permit this, since *base* is implemented as the constant function `bn()`. On the other hand, mgArgument takes *base* as a dynamic variable, so it would permit having several Modulus Groups. Now, since we assume that it is possible to automatically translate between the three different implementations that we have introduced here, he chooses to translate his mgConstant implementation to the mgArgument implementation. Now he is able to use many different Modulus Groups.

Later, he finds out that he wants to use his Modulus Group in the context of some other code. Say that he wants to interface his implementation with some algebraic concept. But this concept cannot accomodate the mgArgument implementation, since the dynamic *base* number does not fit into it. It effectively requires *base* to be abstracted away, or to at least not be exposed like it is in mgArgument, namely as a dynamic argument to all of the relevant functions. The solution to this is to, for all values of *base* that the programmer cares

about, specialise mgArgument to an implementation with *base* specialised to that constant number. The mg7 implementation is one example, but of course the constant number could be any positive number that the programmer is interested in.

To sum up the workflow:

1. Implement mgConstant manually.
2. Automatically transform mgConstant to mgArgument.
3. For each desired *base*, specialise mgArgument to an implementation with this specific and constant *base*, like mg7.

6.2 SPECIALISATION OF FUNCTIONS

The implementation mgArgument takes *base* as an argument to the various functions, while mgConstant has *base* defined as a constant in the code. In order to derive mgConstant from mgArgument, we need to *specialise* the arguments corresponding to *base*. This is an application of partial evaluation.

More concretely, we are interested in fixing one value `bn` which is a parameter to all the functions of implementations as a member of the `MG` type. Since we are fixing one value to a static value, this requires a monovariant specialisation.[1]

The `bn` value is embedded in the `MG` type. Let's assume that we are able to divide `MG` into the types `MG_1` and `MG_2`. Once this has happened, we have a type `MG_2` with a single member `bn` that is sent into each function in the implementation (we also assume that duplicates of bn has been removed from the parameter list). We can simplify this so that each function has a parameter `bn: Integer`. Now we can mark all these parameters to signify that each `bn: Integer` parameter is statically known, and that they have the same value. Now the resulting program does not need to have a parameter `bn` and can instead incorporate it as a constant in the function. Furthermore, since it known that all these `bn` constants are the same value across the different functions, its value can be stored in a mutually callable place; like how the constant value is stored and called from the `bn()` function in the mgConstant implementation.

Note 4 (Constant functions). Magnolia functions are referentially transparent. As a consequence *constant functions*—functions with arity zero—must necessarily return the same value each time they are called. In turn this means that constant functions can be used to represent constant values.

[1] Monovariant specialisation was discussed in section 4.1.

6.3 lifting and lowering

We here introduce the notions of *lifting* and *lowering*,[2] which will be useful when discussing the differences between the different Modulus Group implementations.

Lowering is a general term which we will use to refer to making some *item* available *sooner*. *Lifting* is then, in contrast, the opposite of lowering. "Item" here is a loose collection of whatever we find relevant in the program, and for which the term makes sense. The word "sooner" refers collectively to some "time" occurring either at the stage of writing the program, during what one might call the compilation pipeline, or during execution of the program. So "time" in this sense refers to some discrete point in the stage of writing the program, compiling the program, or running the program.

We will get into the details of exactly what these terms mean for some different kinds of constructs. But let us first give some concrete examples of things for which *lifting* and *lowering* mean. The first example is program expressions; an expression is *lifted* if it is taken into a more dynamic context. An expression is *lowered* if it is taken into a more static context. The second example is requirements and programs; a program is *lifted* if it is stripped of its implementation, yielding only its signature. This can be achieved with constructs like `signature(_)`. In contrast, a requirement is *lowered* if an implementation—for e.g. the type, function, etc. as appropriate—is supplied.

To elaborate on program expressions, consider the application of techniques like partial evaluation. In partial evaluation, whether an expression is statically computable is important. If it is statically computable, the result of the expression is in principle available at compile time. In general, the results of expressions are only available at runtime. So the result of a static expression is available *earlier*—in our conception of "time"—than a dynamic expression. Applying the term "lowering" to static and dynamic expressions, *lowering an expression* then means to transform the program in such a way that the expression is part of some construct which is guaranteed to be available statically. *Lifting an expression* then means the opposite, namely to take an expression out of some construct that is guaranteed to be statically available, and putting it in a context which might only be dynamically available. Note that *lowering an expression* is meant to be a simple transformation that does not change the expression itself. So *lowering* is not supposed to include heavy-duty program transformation techniques like partial evaluation, which attempts to evaluate as much of a program statically as it can; *lowering* uses the expressions as-is but simply *moves them around*, loosely speaking. Also note that in order to *lower an expression*, the expression must already be able to be lowered—it needs to be statically computable. There is no restriction on *lifting an expression*; an expression, or any other item, can always be put in a context which

[2] Which are terms of our own making. But which aims to usefully categorise (and perhaps unify) already established concepts in the literature.

effectively makes it available *later*.

Let us consider how this might look like in a programming language. The running example is an expression that can in principle be computed statically, but is in such a context that does not *guarantee* that the expression will be computed statically. In the case of Magnolia, we have already seen that static expressions can be represented as constant functions. So *lowering* here would involve taking the expression *out* of a construct or context which is in principle dynamic—like a variable—and *into* a static construct, like a (constant) function.[3] Consider the C preprocessor as another example, which we discussed in section 2.4. In the case of a value like an integer, we can represent this value as a string-based macro. Now we are guaranteed that the value itself—as opposed to some indirect access to it—will be inlined at every place which the macro is written, effectively making it a static value. As another example, consider C++11 and its *constant expressions*. Constant expressions are guaranteed to be evaluated statically. Thus *lowering an expression* can in C++ be achieved by moving the expression out of a variable or function into a constant expression. Yet another example is Lisp macros. If the expression which is in principle dynamic is represented as a function, then we can instead define it as a macro. Thus we are guaranteed static evaluation.

Static Granularity

One might often regard a program solely in terms of its dynamic semantics. That might be the only part in which something of interest happens, while the static semantics like type checking is just viewed as necessary scaffolding. Compile-time metaprogramming shows that it is sometimes fruitful to not only consider the runtime of the program as something "executable", but also a stage prior to running the program as an executable; what we call "static" or "compile-time". But there is also runtime metaprogramming, such as in the form of programs being modifiable at runtime. In that case, we do not view the dynamic semantics as as one monolithic execution, but rather as many different executions. And in the same way, it might be fruitful to regard the static semantics of a program in a more fine-grained manner. And not necessarily in the sense of explicit metaprogramming, but in terms of *lifting* and *lowering*.

As an example, consider signed or unsigned integers. In case that the integer is *word sized*—i.e., processor dependent—we cannot say what the size in bytes the integer will be, at the time of writing the program. In C, the type `int` was historically likely to be 2 bytes in size. On modern computers it is more likely to be 4 or 8 bytes in size. Still, the size of a type such as `int` is nonetheless statically known. So we have an example of a type whose size is statically known, but which we do not know at the time of writing

[3]Note that Magnolia does not have higher-order functions, so all functions are statically available.

the program. Viewing this through the lens of lifting and lowering, we could regard a generic integer type `Int` as a *lifted* version of a concrete integer type, such as for example a 4 byte integer. A 4 byte integer type would be a *lowered* version of `Int`. Hence we have an example of lifting and lowering between items which are statically known, though still conceptually "concretised" at different times. Magnolia allows the programmer to under-specify the size of an integer type, and then to define its size at a later point. The size may of course have no relation to the word size of the underlying machine, but might for example instead be chosen based on the guaranteed upper bound which values of the type will have.

Not all programming languages have integers or similar constructs of undetermined size. But nonetheless there are plenty of examples to be found of things to view through the lifting/lowering lens. The facilities that a language provides for abstractions is a category to draw examples from.

Definition 2 (Abstraction)**.** A representation of something with some details of the original thing removed. Alternatively, a conceptual interface that simplifies another, so-called *underlying*, interface.

Note 5 (Expressiveness). A more abstract interface is less expressive than the interface that it *abstracts over*, as a consequence of being simpler than it.

In the context of Magnolia, a central abstraction facility is *concepts*. A concept is more abstract than whatever *implementation* or *program* implements it, since it lacks any code for implementing the function signatures and types. A program on the other hand has the function signatures, types and in addition to that implementations for the types and functions. The only thing it lacks compared to the implementation is the axioms. But this is not relevant for the purposes of concrete executable code, since the axioms are not supposed to be part of any final, executable program. So a program is definitely less abstract than the concept that it models. As an example, consider the `Stack` abstract data type.

```
concept Stack = {
  type Data;
  type Stack;

  function peek ( l:Stack ) : Data;
  function pop ( l:Stack ) : Stack;
  function push ( l:Stack, d:Data ) : Stack;
  function clear () : Stack;

  predicate isEmpty ( l:Stack );
  predicate isFull ( l:Stack );
};
```

This concept only tells us the types and functions that a program that models this concept should have.[4] But there are different ways of implementing this concept; we could use a continuous array as a backing storage, a linked list, and so on. So the more abstract notion of a *concept* both frees us from having to care about certain details, and *disallows* us from controlling them even if we may want to—these are simply two sides of the same coin.

Let us see how abstractions relate to the *lifting* and *lowering* notions. Though there might often be subtle ways to make things more abstract, the other end of the extreme can be demarcated as something that is not abstract at all, or alternatively something which is *fully concrete*. In the context of programming, this would be executable code. For a language whose implementation is a compiler with native code as its target, this would be machine code. One might also want to consider the finished program as a fully concrete program, since it is the code that will be fed to the compiler. But there might be some abstractions left in the finished program, which will be absent in the executable. We have already seen an example of this in the form of the `Word` pointer type.

There is also another factor which can be considered when it comes to demarcating a lower bound for *lowering*. Even code that is *fully concrete* from the perspective of the programmer, might still be subject to further transformations. And in that sense it is not *fully lowered*, since it is not the final program. To reiterate, in the context of *lifting* and *lowering* we consider the final program as being *fully lowered*, so to speak. Perhaps the most prominent example of code that is subject to further transformations is non-optimized code. Non-optimized code might be fully concrete at the source code level, in the sense that there are no further abstractions at this representation level. But this code might be written naively when it comes to efficiency, requiring optimisation transformations in order to derive an acceptable final program. Optimisations in the form of those made by compiler are vital; any contemporary, practical compiled language needs to have an optimising compiler. Not merely a compiler that transforms the source code to target code verbatim. In light of this we will not consider the finished program as the *most concrete* program.

Invariant Lifting and Lowering

When lifting or lowering an item, we have to mindful to maintain any invariants associated with it. If we do not, we risk changing the semantics of the program. In particular, we do not wish to lose any checks of correctness. In order to achieve this, we might have to add guards and assertions at certain places in order to preserve the same invariants as before.

[4] We could have also included invariants in the form of *axioms*, but omitted such things for brevity.

As an example, consider lifting a constant value `c: Integer` to a variable. This constant is also associated with a type, which we represent as a wrapper type over `Integer`.

Definition 3 (Wrapper type)**.** A composite type that only consists of a single type. In Magnolia, this is a struct consisting of only one member.

```
type MG = struct{ var v:Integer; };
```

`c` is associated with `MG` in the sense that all values of type `MG` will be associated with the same value, as opposed to each being associated with an arbitrary value of the same type. For example, `c` might be part of an invariant on `MG`.

Now when we lift this constant to a variable, the value becomes a dynamic, variable value at each use-site. This might manifest itself as it being added as a parameter to each function in which it is used. So if the constant `c` was used in three different functions, it would be added as a parameter to each of these functions. But now that this value is a variable—in fact, three different variables in our example—we are no longer guaranteed the same value across the program. This in turn means that we cannot rely on all values of type `MG` to be associated with the same value, because the associated value is a variable and not a constant. At any point in which we have two values `x` and `y` of type `MG`, we need to dynamically ensure that they are associated with the same value. This is in order to preserve an invariant that was inherent in the original program, where the associated value was a constant. This means that we need to add guards or assertions at each *interface boundary* for values of type `MG`. An *interface boundary* is a junction of programming interfaces, such as functions or modules. For example, a function provides the interface of its parameters (input) and return value (output). A function call serves as the interface boundary between the calling code and the body of the function.

So if there are two variables of type `MG` in a scope, they might each be associated with their own variable. If this scope is a function, then we might guard the function:

```
guard v1 == v2;
```

Where `v1` and `v2` are the associated values.

In general, invariant lifting involves adding guards and assertions at appropriate interface boundaries to replace more static checks that would otherwise be lost in the process of lifting.

While we must be mindful to maintain invariants when lifting, we should try and remove redundant invariant checks when lowering. For the purposes of maintaining invariants, we can choose to do nothing to any assertions or guards when lowering. This is because these checks are just redundant; their presence does not make the program less correct. But for concerns of runtime efficiency and maintainability, one should ideally remove all redundant dynamic checks in the process of lowering some item. This aids maintainability in the case of a

lowering if we assume that the resulting code is going to be read and possibly altered by a programmer. Things like redundant dynamic checks are simply noise, which one should avoid leaving around in a code base. So while the concern of invariant lifting is maintaining correctness, the concern of invariant lowering is removing redundant correctness checks.

6.4 TRANSLATION FROM MGARGUMENT TO MGCONSTANT

The implementation mgConstant can be regarded as a specialisation of mgArgument. mgConstant has a constant function `bn()` while mgArgument takes the corresponding value as an argument to the constructor for its type `MG`. Specialising code to more concrete values can be accomplished through partial evaluation. So we inline values corresponding to *base*, if we are looking at the problem through the lens of the partial evaluation technique. What if we assume that we choose to use `MG.bn` as a constant in mgArgument? First consider `MG` in mgArgument.

```
type MG = struct{ var v:Integer; var bn:Integer; };
```

Even if we assume that `MG.bn` is static, `MG` is still dynamic since the whole structure is not static. This is a so-called "partially static data structure" [Jones et al., 1993, p. 223]. But if we decompose the structure we get a dynamic and a static data type:

```
type MG_1 = struct{ var v:Integer; }; //dynamic
type MG_2 = struct{ var bn:Integer; }; //static
```

This technique is called *arity raising* [Jones et al., 1993, p. 371]. As we will see, this technique might raise the arity of the functions it is applied to, whence the name.

The next obvious step is to change the function signatures to use these two types instead of the composite `MG` type. This is straightforward for the parameters, but return values are harder since a function can only return a single value.

We can try to solve this by duplicating the function implementations, with the only difference being that we only return the relevant value. So in the case of `plus(...)` from mgArgument we get two signatures:

```
function plus_1 (
   a_1: MG_1, a_2: MG_2, b_1: MG_1, MG_2: b_2
   ): MG_1
function plus_2 (
   a_1: MG_1, a_2: MG_2, b_1: MG_1, MG_2: b_2
   ): MG_2
```

The next step is to try to specialise these functions such that we end up with these signatures:

```
function plus_1( a_1: MG_1, MG_1: b_1 ): MG_1
function plus_2( a_2: MG_2, MG_2: b_2 ): MG_2
```

With these changes, the implementation for `plus_1(...)` ends up looking like this:

```
function plus_1 (
   a_1: MG_1, a_2: MG_2, b_1: MG_1, MG_2: b_2
   ): MG_1 guard a_2.bn == b_2.bn =
 MG_1{ v = ( a_1.v + b_1.v ) % a_2.bn };
```

Notice that we have removed the expression `bn = a.bn` in the return position since `MG_1` does not have a `bn` member.

The first thing to note is that in the original program, the function requires the `bn` member of both `a_2` and `b_2` to be equal. This is expressed in the guard of the function. Recall our discussion in section 6.3 of why this guarding is necessary. In the new signature that uses `MG_1` and `MG_2`, we are forced to pass in two variables that are required to be equal. The first problem with this is that sending in two equal variables is a clear redundancy. The second problem is that the guard becomes obsolete when we are ultimately only going to send in one, static value. So while moving from mgConstant to mgArgument would require an invariant lift of the `bn` value associated with `MG`, we now want to reverse this operation in order to avoid redundant dynamic checks.

To deal with this we observe the immutable property of function arguments in Magnolia; arguments to functions are not allowed to be changed in the body of the function. This means that any two arguments that are equal are interchangeable at any point in the function.

Property 1 (Substitution of equal arguments)**.** Arguments to a function that are equal may be substituted for each other freely.

So if we assume that we are going to assign the arguments `a_2.bn` and `b_2.bn` to the same value, we can replace all occurrences of `b_2.bn` with `a_2.bn` in the body of the function:

```
function plus_1 (
   a_1: MG_1, a_2: MG_2,
   b_1: MG_1, MG_2: b_2
   ): MG_1 guard a_2.bn == a_2.bn =
 MG_1{ v = ( a_1.v + b_1.v ) % a_2.bn };
```

The first thing to note is that, since we no longer make use of `a_2.bn`, we can remove it as a parameter from the signature:

```
function plus_1 (
   a_1: MG_1, a_2: MG_2, b_1: MG_1
   ): MG_1
```

Property 2 (Elimination of unused parameters)**.** A parameter to a function that does not occur in the body of the function may be removed. This is due to the fact that functions are referentially transparent, so evaluating the arguments to a function has no side effects. These parameters are thus guaranteed to have no effect on the program.

Specialisation 1 (Elimination of parameters with corresponding equal arguments)**.** In the case of equal arguments, one may replace all occurrences of these variables with only one of them by property 1. Then one can remove all the corresponding parameters except for the variable that is used in the body of the function, by property 2.

The second thing to note is that the guard of the function now trivially holds:

```
guard a_2.bn == a_2.bn
```

The predicate `a_2.bn == a_2.bn` is a straightforward tautology in Magnolia. Rules to eliminate such obviously redundant guards are simple to make, and do not require any actual computation of static values such as constant folding. So we assume that this guard gets eliminated, leaving us with this function:

```
function plus_1 (
   a_1: MG_1, a_2: MG_2, b_1: MG_1
   ): MG_1 =
  MG_1{ v = ( a_1.v + b_1.v ) % a_2.bn };
```

6.5 TRANSLATION FROM MGCONSTANT TO MGARGUMENT

As in the translation from mgArgument to mgConstant, we will use the `plus()` function as a running example. Other functions will be brought up in so far as they present new challenges for the translation.

The first thing we need to do is to parameterise `bn()`. This means to replace all occurrences of `bn()` with a locally defined variable. We do this by introducing a fresh parameter, which we will call `bn_new` for reference.

Definition 4 (Fresh variable)**.** A fresh variable is a variable that is introduced into a scope which does not conflict with or shadow any of the already-defined variables in that scope.

`bn_new` is of type `Integer`, like `bn()`. We then replace all occurrences of the function call `bn()` with `bn_new`. This is the resulting function:

```
function plus ( a:MG, b:MG, bn_new: Integer ) : MG =
   MG{ v = ( a.v + b.v ) % bn_new };
```

Generalisation 1 (Parameterising constant values)**.** For a named constant $c : T$ in a function $f(\ldots)$, introduce a fresh parameter $x : T$ and replace all occurrences of c with x in $f(\ldots)$.

This transformation works in the same way for all functions that call `bn()`, which is all functions except `bn()` and `zeroMG()`.

As discussed in section 6.2, when specialising many different variables across an implementation that serve the same purpose, it is important to

remember that they indeed are placeholders for the same thing. In the case of specialising mgArgument to mgConstant, this meant that we were able to factor the constant `bn` into a function `bn()` callable by the whole implementation. In this case we encounter the same problem, but the other way around. During the overall translation between these two implementations, we want to keep in mind that all these new parameters serve the same purpose. This will be important in order to treat the bn values uniformly. This is an interface issue when it comes to program transformation.

Note that `plus(...)` belonging to mgArgument is guarded from being given two arguments with different *base* values. We have already discussed this scenario in section 6.3; if we consider mgArgument as a lifted version of mgConstant, the guard is a result of the resulting invariant lift of *base* being associated with `MG`. One difference from the discussion in the aforementioned section is that none of the functions have parameters corresponding to *base*. This is because *base* is embedded in the type `MG`, so it is given indirectly as an parameter through this type. To reiterate, mgConstant does not have to concern itself with the possibility that numbers associated with different *base* values can be added together; since there is only one *base*, all numbers constructed from mgConstant's `MG` type can be operated on freely. But since each `MG` takes *base* as a dynamic argument, we need to guard from mixing the wrong numbers together. We will now discuss how to embed *base* in `MG`, and then how to do the invariant lift.

We want to embed *base* in `MG`, and to remove the parameters representing *base* from the parameter lists. We do this by extending `MG` with an extra member with a fresh name, which we for reference will call `bn`. Then we need to replace all occurrences of variables representing *base* with this member of `MG`. In this **`implementation`**, all `MG` and *base* variables come from the parameters of the functions. Thus we only have to concern ourselves with changing the parameter declaration of variables, and their occurrences in the bodies of the functions.

So we do these transformations on the implementation:

1. For each function containing a parameter `base: Integer` representing *base*,[5] replace all occurrences in the body of the function with `a.bn`, where `a: MG` is a parameter. If there is no parameter of type `MG`, this transformation does not work.

2. In the case of multiple parameters of type `MG`, add a guard to the function which makes sure that all values of `a.bn` are equal to each other. Do this by first choosing the first parameter of type `MG`, which we will call `a`. Then for each additional parameter of type `MG`, make an expression `a.bn == b.bn`, where `b` is a parameter of type `MG` distinct

[5]Recall how we have previously stated that it is important to keep track of which variables in the program represent *base* as we do consecutive transformations. We take it as a given that we know which variables represent *base* for this transformation.

from `a`. Finally, join all of these boolean expressions into one by applying the boolean operator `&&` in between them, resulting in the expression `a.bn == b.bn && ... && a.bn == k.bn`, where `k` is the last parameter of type `MG` in the parameter list.

3. replace all occurrences of `base` in the body of the function with `a.bn`.

4. Finally, remove the `base` parameter, since it no longer occurs in the body of the function.

Item 2 is exemplified in the `plus(...)` function of mgArgument. This function is guarded by the expression `a.bn == b.bn` to make sure that two invalid numbers are not added together.

Member projection

The next thing to do is to consider functions that return `MG`. Since `MG` in mgArgument contains one more member (named `bn`) than `MG` in mgConstant, we need to make sure that this extra member gets initialized in the constructor of `MG`. A problem with this is that we need to insert a default value for `bn`. Other than relying on predefined values for types like `Integer`, there is no way for a program to know what such a value such be. But in this case, we should use mgArgument as a guide for how to derive such a default value. In mgArgument, functions with return type `MG` assign the same value to the member `bn` as one of the parameters of type `MG`. In the case of multiple parameters of type `MG`, we choose the first parameter of this type in the parameter list. Though all such functions are guarded from these parameters having different values for member `bn`, so this choice is immaterial. This choice of value for `bn` makes sense when we consider the previous discussion about how only numbers associated with the same *base* should be combined; The function `plus(...)`, for example, adds two such numbers which have the same *base*. It is only natural that it returns a number associated with the same *base*.

Definition 5 (Member projection[6])**.** For a composite type T, a *member projection* for member k belonging to this type is a mapping over T that preserves the value of k.

In the `plus(...)` function, we construct the `bn` member by assigning to it directly from `a: MG`, the first parameter of this type. This is a member projection on `MG`, with `a.bn` serving as the value for member `bn`.

```
function plus ( a:MG, b:MG ) : MG guard a.bn == b.bn =
   MG{ v = ( a.v + b.v ) % a.bn, bn = a.bn };
```

[6] *Projection* is a borrowed term from the mathematical concept of a projection mapping.

6.6 the other translations

The other translations between the three implementations follow similar patterns as we have already discussed.

Let us consider the translation from mgConstant to mg7. mg7 can be viewed as a specialisation of mgConstant since mgConstant **`require`**s a constant function `bn(): Integer`, while mg7 provides such a concrete function in the form of `seven(): Integer`. Since this is the only difference between these two implementations, one can translate from mgConstant to mg7 simply by replacing `bn()` with `seven()` in mgConstant. This is a *lowering* of this value, since we replace a requirement for a constant function with an implementation of a constant function. We are guaranteed a specific, constant number, instead of just being guaranteed an arbitrary constant number.

6.7 summary

We have gone through a hypothetical implementation of three different Modulus Group implementations. We have considered how one can move between these implementations by first writing an initial implementation, and then using program transformation to move between them as needed. We have seen how partial evaluation can be applied to this problem, specifically when going from a more dynamic implementation to a more static one. We have introduced and discussed concepts like *lifting* and *lowering* which have allowed us to discuss the different aspects of the transformations.

7

Deriving Set from Dictionary

We discuss how a Dictionary concept and implementations can be used to derive a Set concept and implementations through program transformations.

7.1 INTRODUCTION

We will discuss `Dictionary.mg` and related files; see Appendix B.

Consider an abstract data type Dictionary which represents a collection of $(key, value)$ pairs, such that each distinct key occurs only once in the collection. This abstract data type is also known as Associative Array, Map, and Symbol Table, somewhat depending on what the intended use for it is. We will only refer to it as a Dictionary. Further consider a *Set* abstract data type, taken from set theory. A Set consists of only unique values—no duplicates. The similarity between these two collection types is the uniqueness property. While a Set consists of only unique values, a Dictionary also consists of unique values (keys), but also of values of possibly another type that are associated with each key. (Hereinafter we will refer to the unique values of the Dictionary as *keys* and the values associated with the keys as *values*, or associated values when it could be confused for the generic word *value*.)

Thus a Set can be seen as a degenerative case of a Dictionary where we do not care about the values associated with the keys, only the keys. So one way to effectively derive a Set from a Dictionary is to just insert dummy data in place for the values of the Dictionary, and not expose the interface to insert or inspect these dummy values to the programmer who uses the Set. But of course this leads to needless overhead in the form of inserting and storing dummy data, not to mention keeping around the code to manage this now useless data. So we should rather aspire to eliminate all code associated with the values of the Dictionary. We will discuss how this can be achieved by using program transformations, specifically program slicing.

This problem can be solved through slicing since we are after a subset of the behavior of Dictionary. In particular, we want to remove the `Data` type—the type that represents the associated values—and all other items that this removal would entail. Some items are removed by necessity, such as parameters of type `Data` and functions which return type is `Data`. Other items that we should remove might not be so obvious, such as functions which

become noops after the removal of `Data`. Another example are axioms that test some invariant related to this type.

7.2 Slicing the Dictionary concept

We will go through some of the function signatures and axioms that we get when we slice Dictionary on this criterion:

```
slice -on Dictionary -remove Data
```

Note 6. We use a custom language for expressing the transformation; we will discuss this in chapter 8.

That is, we want the `Dictionary` concept except the type `Data`.

The sliced code will be commented out.

When slicing away `Data`, we of course have to slice away this parameter from any function signatures; such as in `insertNew(...)` and `replaceData(...)`.

```
function insertNew( d:Dictionary, k:Key /*, e:Data*/ )
  : Dictionary;
function replaceData ( d:Dictionary, k:Key /*, e:Data*/ )
  : Dictionary;
```

The slice of `insertNew(...)` makes sense for the purposes of a Set. `replaceData(...)`, however, does not make sense for a Set. There is no associated value to replace, so the function itself should ideally be sliced away entirely. As it is we will probably be left with a function that simply returns the same Dictionary as it got as its input.

We remove the functions which return type is the sliced type.

```
// function find ( d:Dictionary, k:Key ) : Data;
```

In this concept, slicing the axioms just consists of eliminating any `_: Data` parameters and removing any statements which contain a function call to a function that has been removed.

Lastly, we have to deal with a *satisfaction*. In Dictionary, the satisfaction states that Dictionary models `PartialIndexable`. But in the slice SetSlice, this satisfaction does not work. We have to slice PartialIndexable.

```
satisfaction Dictionary_models_PartialIndexableSlice =
SetSlice models
PartialIndexableSlice[ A => Dictionary, I => Key,
// E => Data, get => find,
accessible => isPresent ];
```

It turns out that this slice is useless; the resulting PartialIndexableSlice concept contains no functions or procedures. This is not surprising when we consider that the original satisfaction was about accessing the associated values with the keys used as indices. So we would not expect such a satisfaction to be useful in the derived SetSlice.

Axioms in SetSlice

We will start with the axioms that are not affected by the slicing, i.e., that do not contain any `_:Data` parameers. These axioms turn out to still be useful to this Set concept. `createEmptyNotPresent` tests that the empty value constructor `empty()` does indeed create an empty Set. This is still meaningful and useful in `SetSlice`. `removeNotPresent` also tests something useful for Set, namely that after removing an element from a Set with `remove(...)` the element is not present any more. `removeOtherPresent` tests that removing an element does not also remove an unrelated element. This is also useful in `SetSlice`.

7.3 ANOTHER APPROACH

The Rust programming language has an interesting approach to the problem of deriving a Set implementation from a Dictionary. Rust has *zero-sized types*, which are types which values do not occupy any space. They are defined as empty structures:

```
struct Empty;
```

Rust has a built-in type `()` which is isomorphic to the above struct. It can contain only one value, and as a consequence it can store no information.

Note 7. `()` corresponds to the *unit type* in type theory.

Recall that one way to easily implement Set from Dictionary is to set the type `Data` to something useless. But this incurs overhead since the Dictionary still has to load the useless associated value each time. But the Rust compiler knows how to take advantage of zero-sized types. In order to get a Set it is sufficient to set the parameterised type `Data` to `Empty`:

```
Set<Key> = Map<Key, Empty>
```

Where `Map` is a hash map and `Set` becomes a customized hash set implementation, due to monomorphisation.

Rust uses algebraic data types. For such data structures, concepts like *the sum of types* and *the product of types* is natural to express. In turn, singleton value types like `()` are also treated naturally; when `()` is added to a struct, it does not increase the size that the struct takes. This is because a struct is a product type, `()` is isomorphic to the value 1 in the algebra of numbers, and $x \times 1 = x$. Using algebraic data types to easily implement Set works great for Rust, but it is not suitable to Magnolia. The reason for that is that Magnolia treats type as opaque, abstract types. This allows for more freedom on the implementation side, as Magnolia cannot impose any restrictions on how data is to be structured; all it cares about is the behavior as reflected through the operations on the data. As a result, Magnolia can use code that is made for many different types of memory architectures, without imposing restrictions on things like the layout of memory.

8

Interfacing Transformations

We discuss ways to interface with a program transformation system. We consider directives *as one of the ways to interface with the system.*

8.1 INTRODUCTION

How the programmer is to interface with transformations is where the rubber hits the road for a transformation system. It does not matter how good it is under the hood if the programmer cannot easily *drive it*, so to speak. More structured forms of programming using program transformation systems have not taken hold in any programming communities, perhaps except for the Lisp communities. This might partly be because of interface problems. After all, the field of programming has made many tools centered around text-based interfaces and artifacts. But interfacing with tree-like structures, like abstract syntax trees, seems to remain largely unsolved; consider for example how structured editors have failed to make an inroad into programmer's editing habits.

We propose that there are two primary aspects to the task of interfacing with a program transformation system:

1. selecting the part of the program to work on; and
2. expressing the transformation.

We will mostly focus on point 2, by considering the necessary features for a domain-specific language (DSL) for program transformations.

8.2 MOTIVATION

Creating any kind of programming language is an undertaking. In principle, any interface can be expressed as a custom DSL, optimized for that specific purpose. But this would take a lot of work, and there is an opportunity cost to having to learn a DSL for every application that one would use. If you only use something once in a while, it is not worth it to learn a very peculiar interface, perhaps least of all a DSL.

Note 8. We will only consider textual DSLs, like the vast majority of programming languages.

But it seems fruitful to make a DSL for a transformation system. First of all, we will restrict the domain of expressing the transformations to such a point that the language ends up being relatively easy to implement and to use. Second of all, the DSL will not be the only way to interface with the system, so all users of the system will not necessarily have to learn the DSL. Third of all, having a transformation language allows the programmer to make her own transformation scripts, and to use tools like a VCS to manage the transformations (we will explore this in chapter 9). Moreover, the DSL will have the potential to complement and support alternative interfaces. The reason for that is that a menu-based graphical interface, say, can be made to also output the equivalent transformation commands in the DSL. This has been implemented before in graphical interfaces for languages, such as the statistics language R.[1]

Having a DSL for a transformation system also allows us to make interfaces and ways to develop Magnolia code that is not possible without such a custom language. Later in this chapter we will explore one such possibility in the form of using the DSL in directives. This allows us to interpolate transformation commands and expressions with Magnolia code.

We mentioned old editors like Ed in section 2.3. The limited peripherals of the time made interactive editing of text quite limited. Instead, you had to think up the edits that you wanted to do to the text, execute them, and then view the results. But necessity is the mother of all invention; the computer-limitations of the time forced the developers of such editors to make languages for editing text: *command languages*. These languages allow the user to programmatically edit and extract text. This meant that when Vi was invented, which is a visual, full-screen editor, one could leverage the underlying command language Ex to drive the editor. And today users of editors like Vim (one of the modern implementations of Vi) can easily program text editing through things like macros. Although we do not seem to have much in the way of limited computer capabilities these days, we believe that such a scheme can be fruitfully be applied to more domains than text editing—including program transformations.

8.3 needs of the interface

When designing a language we need to know what it needs to express, and in turn the features that it requires. We will limit the needs put on this language, and in turn make something that should be relatively easy to implement and to learn for users.

We consider for our purposes transformations to be functions on programs. Each of these functions is a transformation in a global catalog that the programmer can choose from. Each transformation takes some input and returns

[1] See for example the program R Commander [rco] which returns the R code for menu selections.

a single output in the form of a *program fragment*. What we mean by *program fragment* is a part of a program that might not compile on its own, but should be put in such a context that it makes for a program that does. The inputs are also program fragments. This is a non-exhaustive list of the values that the transformation system should be able to handle as input and output:

- Variables
- Operations
- Concepts
- Implementations
- Axioms

In short, anything that can be named in Magnolia should be able to be handled by this language.

We make a separation between the implementation of transformations and the use of them—we do not intend to be able to implement fundamental transformations in the language. Complex transformations that involve intricate knowledge and manipulation of the program are to be implemented in the compiler, and also through some extension mechanism in case there is a need for third-party transformations. Still, it might be practical to allow for some limited composition of existing transformations. If the user is to be able to use this language to script transformations, she should be able to store transformation sequences as transformation procedures and call them from other procedures. Not having any form of modularisation would probably be too limiting. Further, there could be a need for basic composition of transformations, as opposed to just being able to express sequences of transformations. We will address this later.

8.4 THE TRANSFORMATION LANGUAGE

The primary element of this language are *commands*. A command is a transformation function, which is either one of the transformations in the global transformation catalog, or a user-defined command. The syntax is similar to Haskell's function call syntax, or shells like Bash' command syntax.

As a running example we will use `renaming`, a transformation that takes a module item and a list of renamings as input and outputs a module item with the relevant renamings. This is an existing, built-in transformation in Magnolia. Our transformation `renaming` will be slightly different from the built-in `renaming`, but will be functionally the same. The following renaming renames the functions of a dequeue (*double-ended queue*)to more domain-appropriate names for a stack:

```
renaming dequeue_as_frontStack = [
  Dequeue => Stack,
  pushFront => push,
  peekFront => peek,
  popFront => pop
];
```

The motivation for this renaming is that the operations of a dequeue subsumes the operations of a stack; the front or back of the dequeue can be operated like a stack. In our case, we use the front operations.

renaming is a function that needs to take the following arguments as input:

1. A module-level Magnolia item to be renamed.
2. The renamings.
3. The name of the new module with the renamings.

We will use named parameters. The motivation for this is that we think it will be easier to deal with for the domain of transformations than having to remember the position of parameters Another reason is that transformation expressions will probably in practice be less nested than normal expressions, which mitigates the downside of named parameters being more verbose than unnamed parameters. The named parameters are, respectively:

1. `-module`
2. `-renamings`
3. `-name`

We indicate the named parameters by the sigil hyphen (-). The values for each parameter follows directly after the name, like how many command line programs are parsed. Thus the renaming ends up looking like this:

```
renaming -name dequeue_as_frontStack
   -module Dequeue
   -renamings [
     Dequeue => Stack,
     pushFront => push,
     peekFront => peek,
     popFront => pop
   ];
```

The language uses these data types:

1. Names, e.g. concept names, function names.
2. Pair of names.

3. List of names or pairs.

Note 9. The pair syntax is taken from Magnolia's current **renaming** construct.

In the renaming above –`module` and –**name** are names while –`renamings` is a list of pairs.

The **renaming** transformation above is a command. Commands are terminated with a semicolon. Commands have the effect that they affect existing Magnolia code or add new code. This code can be thought to exist right after the command has been evaluated. The effect of the **renaming** above is that a new concept `dequeue_as_frontStack` is added:

```
concept dequeue_as_frontStack = {
  ...
}
```

Commands that follow the **renaming** are able to use Magnolia items that preceding commands have produced. In a wider sense, commands operate on the state of the Magnolia code that it is implicitly operating on, whether that may be just one module or several modules. Thus, this language can be considered an imperative language which implicit state is the code it is operating on.

Note 10. Recall from section 8.1 that we mostly concern ourselves with the task of expressing transformations, and not so much with the task of selecting the part of the program to work on; and. This is why we do not concern ourselves that much with the question of what kind of selection of the code base the transformation commands are operating on.

The **renaming** transformation above is a command. More specifically, it is a command consisting of a *transformation expression*. Transformation expressions have in our discussion have just consisted of transformation function calls. It might be useful to have some primitive operations that operate on transformation expressions. One example might be an infix combinator `|` that passes the output of one transformation expression to the next transformation expression:

```
renaming ... | slice ...
```

It might also be useful to have some primitives that operate on data types like names and lists. An example might be an infix ++ operator which appends lists.

As mentioned previously, it is most probably necessary to have some facility to define some kind of procedure in order to modularise code. But as we will not be needing such things in our discussion and examples, we omit this discussion.

8.5 Transformation Languages as Glue Languages

A transformation language like we have described can be regarded as a high-level language for transforming Magnolia programs. Such a language might

have some similarities with glue languages, in the sense that it is meant to be a high-level language for combining transformations (although less so in the sense of connecting different heterogeneous components, since we just deal with Magnolia code). There have been many languages that might have some point been intended to serve as glue languages:

- Shell languages (Bourne shell, Bash, Zsh, Fish, and more)
- Tcl (originally intended as an extension language)
- Perl (also a general-purpose language)

Shell languages are often used to glue different programs together in order to make new functionality or programs. The following script takes a number n as input, and lists the n most used words sorted by their frequencies:[2]

```
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1}q
```

Such *pipelines* (using `|`) can solve certain tasks succinctly.

The long history of glue and scripting languages should provide lessons when designing a DSL for transformations. We propose that a DSL should be fit-for-task in that it sticks to being good for solving problems in its domain, and does not try to overextend its role. Some glue languages have arguably overextended themselves by incorporating more features, perhaps particularly features intended for more general-purpose programming. For example, Bash has a peculiar syntax compared to non-shell scripting languages, which makes it so that even basic things like conditionals and properly looping over values can be hard to master or remember if you do not use it that regularly.

Not all languages have to use separate meta-languages or scripting languages in order to achieve things like combining transformations. Lisps are homoiconic and built for metaprogramming, and in turn it is more natural to metaprogram a Lisp program in Lisp than to interface with a separate language for metaprogramming. In contrast, Magnolia is designed to be a programming language which is easy to analyse, and so in turn it does not have any mechanisms for runtime or compile-time reflection or metaprogramming. And as we discussed in section 2.4, a language designed like Magnolia—including those in its lineage—are hard to metaprogram in itself, and in turn what is essentially separate languages in style and use are used (like C++' templates). Given the design goals of Magnolia compared to more dynamic and metaprogrammable languages like Lisps, having a separate language for metaprogramming seems appropriate.

[2]This script was written in 1986 by Douglas McIlroy in response to a program written by Donald Knuth to solve the same task [Bentley et al., 1986].

8.6 DIRECTIVES

We will now discuss *directives* and how our transformation language can be used in this role.

A directive is a language construct that operates somewhat separately from the rest of the language. In fact, it might not be part of the language's grammar—we discussed the C Preprocessor in section 2.4, and they are expanded before the language is processed further. Directives can be used for things like specifying or giving hints to the compiler. For example, the C Preprocessor has directives which can be used to specify which parts of a program should be compiled based on some criteria; this is called *conditional compilation*. We will use discuss directives as a front-end to transformations which the programmer can use to interpolate transformations with Magnolia code.

Motivation

In our view, directives serve to solve problems that have to do with cross-cutting concerns when dealing with code and crucial language tooling like compilers. A compiler for a language might be invoked by some commands that take parameters that let you configure things like optimisation levels. The compiler program has no way to communicate with the language, and vice versa. This separation of concerns is often proper and sufficient, but it is too restrictive in some cases. The crux of the issue is that the writer of the code has no way to communicate certain concerns and intents behind parts of the code. This is because the language probably is not expressive enough, or have the mechanisms to express, things like *inline this code*, *pack this structure*, *do not compile this code for that architecture*, and so on. Directives allow instructions to third-party tools like compilers to live side-by-side with code, without the language necessarily having to understand them.

Design Considerations

We propose that it is important for directives to not interfere with the language. Some languages don't have a facility for directives *per se* but instead introduce them through some more informal means. An example of this can be seen in the Go programming language. Go version 1.4 [go1] introduced the `go generate` tool which uses comments starting with `go:` to generate code. One can use these comments as directives to invoke external programs like Yacc. The problem with this more informal approach is that now all readers of the code have to be vigilant of whether any given comment has some program-affecting meaning or not. Another example is the string `+build` , which a library called *Build* uses to declare constraints on whether the file should be included in a package. For example:

```
// +build linux,386 darwin,!cgo
```

This isn't only a problem for semantic comments, but also for comments that *look* like they should be semantic but are not. All it takes is a simple typo in what was intended to be the introduction of the directive (misspelling `buil`):

```
// +buil linux,386 darwin,!cgo
```

And now that comment is just a comment, not a directive. Of course one could make error messages for such cases. But as you provide more and more error-checking, you intrude more and more on what was supposed to be the domain of free-form data for human consumption; not commands for external tools.

Magnolia Plus Directives

We will now discuss directives in Magnolia as an extension of the language. This does not mean that Magnolia itself has knowledge of the language, but rather that it allows for directives to occur in Magnolia code. All an implementation of this language needs to do is to ignore at parse-time the parts of the program that are marked as directives. Directives are lexically distinguished from Magnolia by two sigils: `#` and `@`. These are to be immediately followed by a set of parentheses that contain the directives:

```
@(...)
...
#(...)
```

It is useful to explicitly name this new language, since we will sometimes want to refer to "Magnolia with directives", and also "Magnolia without directives". We will name this new language Magnolia PD, which stands for *Magnolia plus directives*. Note that we consider this a separate language from Magnolia. The purpose of this *layering* of languages is to maintain a clear separation between directives-extended Magnolia PD and plain Magnolia. We propose that this separation will make it easier for tooling to deal with Magnolia code, as they can choose to concern themselves with Magnolia instead of Magnolia PD if that is more appropriate. Thus we should avoid the problem that languages like C has with the C Preprocessor complicating the analysis of the language (although our directives should be much less invasive than C's macros).

Magnolia PD has two different kinds of directives; *block directives* and *inline directives*. These differ in where they can occur. Block directives are associated with items like functions and concepts, so they are put next to them, specifically right before them. Thus they are similar in style to Java's annotations, Rust's attributes, and others. Inline directives occur in positions where Magnolia constructs are expected. So the aforementioned Magnolia renaming:

```
renaming dequeue_as_frontStack = [
  Dequeue => Stack,
```

```
  pushFront => push,
  peekFront => peek,
  popFront => pop
];
```

Could be replaced with an inline directive expression like this:

```
@(renaming -name dequeue_as_frontStack
   -module Dequeue
   -renamings [
     Dequeue => Stack,
     pushFront => push,
     peekFront => peek,
     popFront => pop
   ]);
```

Here we have taken the previously described transformation command `renaming` and put the transformation command inside in an inline directive `@(_)`. The semantics of directives is that they are reduced to Magnolia code before compilation, and thus directives impose a preprocessing phase. In turn, Magnolia PD can be regarded as an extension of Magnolia that adds a new phase to the compilation pipeline.

Note 11. We use the term *reduce*; it might be more familiar to think of the directive as *expanding* to an appropriate construct (renaming in this case). We use the term *reduce* in order to stay consistent with how we talk about evaluating code.

Expansion and User Interaction

It should be possible to inline directives in the code. We will call this *inline reduction*, or *inline expansion*. There are two uses for this:

1. Allow the programmer to see what the directives will reduce to; providing a different *view* of the code.
2. Use a directive to generate code, and then to manually change that code.

Note 12. Point № 2 is unidirectional; we do not consider mechanisms for *collapsing* Magnolia code to a directive, in order to avoid complications that bidirectional transformations bring.

This is another example of how our transformation language can be used to provide the programmer with different interfaces to the code.

8.7 SUMMARY

We have discussed some aspects of interfacing with a transformation system. We have considered this topic through the lens of a textual language—a DSL

for transformations. We have argued that such a language is worth the effort to implement and to learn, since its domain of use can be sufficiently restricted, and because it can be used to complement and support other ways of interfacing with the transformation system. Finally we applied our transformation language to directives in an extension of Magnolia; Magnolia PD. Transformations can be used through this mechanism to intermingle Magnolia code and transformation code, something that we think would be an asset to Magnolia programmers.

9

Bookkeeping

We discuss how a bookkeeping *system can be used in order to complement a workflow which is in part based around program transformations.*

9.1 INTRODUCTION

The transformation approach to programming is a more structured way of programming. A more structured process allows for more structured information to be gleaned from the process. As a consequence there is potential for capturing that information in some kind of *bookkeeping* system, for later review and maybe even for manipulation. We will discuss how a potential bookkeeping system could be designed to assist a more transformation-driven approach to programming. This bookkeeping system would capture each step in the process of development, which could then be reviewed and manipulated.

For a discussion about bookkeeping systems in early program transformation systems, see Partsch and Steinbrüggen [1983, p. 3].

Terminology

In our discussion, we will need two to use these two terms:

- A transformation session is a user-initiated and user-stopped period of time where the user works on the programs, using both transformations and manual coding, or *hand-coding*.
- A step consists of either a transformation, or normal coding (hand-coding). Conceptually one can consider a transformation step to be stored as the sequence of transformation commands that were done in that step, while a coding step can be stored as a set of patches.

9.2 VERSION CONTROL INTEGRATION

Managing the change of code over time is the domain of VCSs. A transformation system could fruitfully use Git as a backend for its bookkeeping system. In such a system, each step in a session corresponds to a commit. Since Git has very low-level semantics we do not think it is fruitful to use Git directly

(specifically the Git(1) command line interface). Instead, it should be more practical to use a higher-level tool such as Stacked Git (StGit), a system for managing a stack of patches stored in Git. This because it provides higher-level features like *undo*, *redo*, and more.

As mentioned, there are two different steps to store. A hand-coding step is simple to store, since it is just a regular commit without any additional meta-data. For a transformation step meta-data should be stored in addition, namely the sequence of transformations that produced the change to the code. Since Git has poor built-in support for adding meta-data to a commit,[1] meta-data could be stored in a dedicated directory at the root of the directory tree, like how Git uses the `.git` directory. When a session is committed, the commits would be rebased and the meta-data for each commit put in a Git note instead.

The integration with StGit would abstract over the commands for Stg(1), which is the CLI for StGit. This is in order to be sure that the history, from the bookkeeping system's viewpoint, is not ruined by manual intervention from the user. So this integration demands that the user goes through our limited interface when it comes to her version control needs, for the duration of a session. The bookkeeping system could provide some commands for manipulating the history, such as combining steps into one step. Moreover, the final history can be changed in Git once a session has been committed; to commit a session means to yield control to Git. What this means is that Git(1) commands can be run inside the directory tree of the project. In order to abstract over Git, we would need to *disable* it, somehow. This can be achieved by renaming the Git directory `.git` to something like `.transformation`. So when a session is committed, `.transformation` is renamed back to `.git`.

9.3 demands of the user

The process of programming is iterative and not at all straightforward. And just like VCSs demand some discipline from the workflow of the programmer, as does this bookkeeping system. First of all, the user needs to decide when to start and end each session. She also needs to not use the underlying VCS while doing a session, as stated in the previous section.

Are these demands worth it? One of the goals behind the ideas in this thesis is to find out more about more structured approaches to programming. A bookkeeping system would provide rich information about user experiences with such approaches. In turn, we think that one would learn more about what works and what does not work in practice. In this light, we propose that such a bookkeeping system would not slow down or hinder the programming process, but would on the contrary *improve* the pragmatics of the system in the long run.

[1] Git-notes(1) could have been used, if not for the fact that StGit does not handle them correctly; the notes are not carried over on so-called rewrites of commits.

9.4 REPRESENTATIONAL LEVELS

In proposing an approach to program in a structured way and using Git as a VCS backend, one might think that we have set ourselves up for a contradiction. If we want to manipulate code in a more structured way, why would we want to use a tool that stores source code as unstructured plain text. Furthermore, even when it comes to source code management tools, there are more *intelligent* tools to choose than Git.[2] This might seem like a sub-par arrangement, but focusing on a snapshot-based VCS is very much a deliberate choice. We deliberately choose to take advantage of multiple conceptual *representational levels*.

A representational level can be thought of as how an item is represented at a certain level, often a conceptual one. For instance, a wooden table can be thought of as a collection of atoms, as that is one representational level. But in everyday settings it is more useful to regard it as a table that is made from wood. Programming languages often have a text-based grammar, in the sense that all valid programs are strings. A program as a collection of text is one representational level. One can also regard a valid program as an *abstract syntax tree*, and that is a more useful representation in certain circumstances. Another representational level could be to regard the program as a flowchart. Note that it is not important whether the programmer can transform the program into a flowchart; if she can merely conceptualizes the program as such, then it can be thought of as a representational level.

The version control format is a representational level. For versioning source code, there are several factors that may important:

- How fast two histories can be merged together.
- How fast to check out a point in history.
- Whether a repository can be partially checked out, or the whole history needs to be checked out
- How fast to clone a repository
- How fast to *bisect*[3] the history.

All of these concerns are unrelated to our high-level goal of more structured programming. If we were to add the constraint that the version control should operate on more structured data, it might interfere with factors like the ones listed above. Moreover, one might have to solve those problems all over again,

[2] See for example Mimram and Di Giusto [2013] which describes a categorical theory of patches, which is clearly a more involved approach than Git's snapshot-based model.

[3] *Bisect* is a term from Git which means to use binary search to search backwards through the history of a branch in order to find some point of interest. Often the point of interest is the commit in which some bug was introduced.

instead of relying on the work put into existing tools. VCSs have centered around plaintext files for decades, and there a lot of tools to deal with such systems.

Higher levels

So we have established that there is value in leaning on the work done on plaintext VCSs. But how are we going to have structure at a higher level? The answer is to represent that on a higher representational level. The first step is to store higher-level data. We have already touched on this, namely when we discussed that we store transformation-specific data in the VCS for each step. Let's say that this transformation-specific data is in the form of a sequence of transformation commands. By itself, it's just a representation of how to get from the previous step to the current step, which should give the same code as output as a unified diff to represent the change as line changes.

Cohesion between levels

We can take advantage of the aforementioned definition of sequences of transformation commands:

Invariant 1. The transformation commands for a step should result in the same code as the code stored in the commit for that step.

The key to complete this higher-level representational level is to provide mechanisms to enforce these invariants. This is how we enforce cohesion between representational levels. So if we for example have completed six transformational steps, this mechanism should check that for each step the aforementioned invariant holds.

Another way to look at this is to consider the storage at each level as simple data at rest. Other than trusting the processes that generate the data, we have no guarantees that they are cohesive. But if we add dynamic checks on the passive data, we can enforce that invariants like the one above are maintained. So dynamically-checked invariants on data at rest enforce cohesion between representational levels. This means that, when dealing with these levels, there should be automated dynamic checks which regularly check for cohesion. One could for example run these checks for each step that is done, undone, redone, etc. This in turn means that our approach requires more than simple data storage, but also continuous dynamic checks on the data.

9.5 breadcrumbs

When you may want to retrace your steps, it can be useful to leave behind clues about where you've been. Version control allows you to do that, by allowing you to take snapshots of points in the history of the directory tree. Another example of leaving behind breadcrumbs is the undo history of an editor. This

undo history might even record branch points in the undo history, similar to how you can branch off in non-linear VCSs. These different mechanisms provide different breadcrumbs which traces your steps from A to B. While these mechanisms can work independent of each other, they can also work in parallel, which here means that they record different kinds of breadcrumbs at the same points. We have already discussed two such alternatives. Using Git as a VCS backend, we store snapshots of the code for each commit, which in our case is a single step in the transformation workflow. But for *transformation steps* we in addition also store meta-data about the transformation that we performed. In this way, we record a breadcrumb about the transformation, which is parallel to the code snapshot.

What point does having multiple breadcrumbs serve? Having different breadcrumbs allows the user to trawl or trace through history in different ways. If the programmer has made an editing mistake, it might be convenient to use the undo history to go back to a good point. If the programmer gets an incomprehensible error due to recent changes, it might be more convenient to use the VCS to go back to the last commit.

What point does having multiple *parallel* breadcrumbs serve? The primary benefit is that they are recorded at the same time. In the case of the transformation workflow, they are recorded at a user-chosen step. Contrast this with having two independent points for code snapshotting and for recording the transformation, which the programmer has to choose. Then she has the overhead of having to choose another point. But if we make these two breadcrumbs parallel, she still only has to manually choose *steps*, and get both a code snapshot and a recording of the transformation sequence.

Inspiration for breadcrumbs

What we call *breadcrumbs* is inspired by Vim's *registers* and journalling and logging in databases and file systems. Vim maintains many registers which contain the last text of some sort. Some of these registers are:

- Numbers 0–9: last yanked[4] text, newest to oldest.
- The last searched text.
- The path of the opened file.
- The last executed command (using the Ex command language).
- The last inserted text (inserted in *insert mode*).

These automatically managed registers allow for things like not repeating the same text input for different input fields (search, commands, and so on). They can be regarded as breadcrumbs that Vim leaves behind for the user as

[4] "Yank" is Vim's term for copied or deleted text, basically.

she uses the editor. And if these are expanded to be lists of the last inserted texts, executed commands, and so on, then the editor can keep track of the whole history of the editing session for the user. And, crucially, it is many aspects of the editing section, not simply one aspect like the undo list or tree.

As mentioned, what we call breadcrumbs is also inspired by journalling and logging in databases and filesystems. Journalling is a technique for committing changes to a filesystems in a more fault-tolerant way. Updating filesystems can take many separate write operations, and so a system crash or power failure can easily leave the filesystems' data structures in a corrupted state. To mitigate this, a journalling filesystem first record what is to be done to the filesystem in a *journal* or *intent log*. Since these data structures are faster to update than the write operations to be committed, there is a greater probability that the relevant operations can be redone or undone in case of some failure. Although the purpose for filesystems and databases is to maintain data integrity, it seems that such automated audit trails could enhance practices like programming. And in case maintaining such breadcrumbs is too costly, the journalling technique can probably be applied to postpone committing full updates until opportune times, or commit writes to disk in batches.

Something related to such filesystem and database techniques is *event sourcing* [Fowler, 2005]. Event sourcing is an application technique that stores all changes to the state of an application as a sequence of events. With this particular kind of log you can not only query the current state of the application and a record of other states that application has been in, but you can reconstruct these pasts. Thus the log functions as a time machine, complete with the power to time travel and in turn make alternate histories of the application's state. For example, instead of storing the current value of some account and a log of the transactions, each transaction would be stored in the event log and can thus be replayed.

10

Conclusion

We conclude this trip through the world of metaprogramming.

We have explored program transformations in the context of the Magnolia programming language. We have argued that program transformations is useful to the craft and process of software engineering. We have explained how program transformations should be applied to Magnolia, given the design of Magnolia and its intended uses. We have contrasted with other languages' approach to metaprogramming, like Scheme and C. We have looked more in depth at two program transformation techniques, partial evaluation and slicing. We have also considered two cases for applying program transformation to Magnolia code: Modulus Group and deriving Set from Dictionary. We have also considered usability, tooling, and workflows that a more structured approach to programming can enable, by discussing how to interface with a transformation system and how a bookkeeping system can aid the programmer. We believe that a transformation system does not live in a vacuum, and that languages and tooling should help to both complement the strengths of such a system, and to help with any weaknesses.

The author hopes that he has managed to make the case for this style of programming—a more interactive and feedback-driven form of programming, using high-level tools to inform and assist the process. The time will probably never come for the *sufficiently smart compiler* that manages to delegate all optimization task, but we can do better than have to either blindly trust a black-box compiler or to do it all ourselves. And program transformations go beyond optimizations, of course—just like we can make arbitrary programs, we can make arbitrary metaprograms. Just like we can use computers to crunch numbers, we can use them to crunch programs represented as text, as trees, and automate at ever-higher levels.

But I also hope that I have managed to make the case that program transformations can be utilized as a modest extension to the typical modern software development. Program transformations do not call for perfect rigor, or for expressing all changes to code through semantics-preserving transformations. The approach I have proposed is a middle way of allowing for interleaving regular programming and program transformations. Donald Knuth wrote in 1974 (via Tony Hoare) that an ideal language should be designed so that an optimizing compiler can describe its optimizations in the source language. I

think such ideas are as relevant as they ever were, and hope that they take more foothold in the present day.

Bibliography

Go version 1.4. <https://blog.golang.org/go1.4>. Accessed: 2017-06-27.

The r commander. <http://socserv.mcmaster.ca/jfox/Misc/Rcmdr/>. Accessed: 2017-06-26.

Scheme factorial. <https://rosettacode.org/wiki/Factorial#Scheme>. Accessed: 2017-06-20.

A. Aho and J. Ullman. *Principles of compiler design.* Addison wesley, 1977.

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques.* Addison wesley, 1986.

A. H. Bagge and M. Haveraaen. Interfacing concepts: Why declaration style shouldn't matter. *Electronic Notes in Theoretical Computer Science*, 253 (7):37–50, 2010.

R. Balzer, N. Goldman, and D. Wile. On the transformational implementation approach to programming. In *Proceedings of the 2nd international conference on Software engineering*, pages 337–344. IEEE Computer Society Press, 1976.

F. L. Bauer. Programming as an evolutionary process. In *Proceedings of the 2nd international conference on Software engineering*, pages 223–234. IEEE Computer Society Press, 1976.

J. Bentley, D. Knuth, and D. McIlroy. Programming pearls: A literate program. *Communications of the ACM*, 29(6):471–483, 1986.

R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.

D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. In *ACM SIGPLAN Notices*, volume 50, pages 117–130. ACM, 2015.

C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 493–501. ACM, 1993.

A. De Lucia. Program slicing: Methods and applications. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 0144–0144. IEEE Computer Society, 2001.

A. P. Ershov. On the partial computation principle. *Information processing letters*, 6(2):38–41, 1977.

A. P. Ershov. On the essence of compilation. *Formal Description of Programming Concepts*, pages 391–420, 1978.

A. P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18(1):41–67, 1982.

M. S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(1):1–20, 1982.

M. Fowler. Event sourcing. <https://martinfowler.com/eaaDev/EventSourcing.html>, 2005. Accessed: 2017-06-27.

Y. Futamura. Partial computation of programs. In *RIMS Symposia on Software Science and Engineering*, pages 1–35. Springer, 1983.

Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

J. Hatcliff, M. Dwyer, and S. Laubach. Staging static analyses using abstraction-based program specialization. In *Principles of Declarative Programming*, pages 134–151. Springer, 1998.

S. Horwitz, T. Reps, and D. Binkley. *Interprocedural slicing using dependence graphs*, volume 23. ACM, 1988.

N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In *Rewriting techniques and applications*, pages 124–140. Springer, 1985.

N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic computation*, 2(1):9–50, 1989.

N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

S. C. Kleene, N. de Bruijn, J. de Groot, and A. C. Zaanen. Introduction to metamathematics. 1952.

D. E. Knuth. Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974.

J. Komorowski. *Synthesis of programs in the framework of partial deduction*. Åbo akademi, 1989.

Z. Manna and R. Waldinger. Synthesis: dreams→ programs. *IEEE Transactions on Software Engineering*, (4):294–328, 1979.

S. Mimram and C. Di Giusto. A categorical theory of patches. *Electronic notes in theoretical computer science*, 298:283–307, 2013.

K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *ACM Sigplan Notices*, volume 19, pages 177–184. ACM, 1984.

H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys (CSUR)*, 15(3):199–236, 1983.

T. Reps and T. Turnidge. *Program specialization via program slicing*. Springer, 1996.

A. D. Robison. Impact of economics on compiler optimization. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 1–10. ACM, 2001.

P. Sestoft. *The structure of a self-applicable partial evaluator*. Springer, 1986.

T. Sheard and S. P. Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Notices*, volume 32, pages 203–217. ACM, 1997.

F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

Glossary

abstraction A representation of something with some details of the original thing removed. Alternatively, a conceptual interface that simplifies another, so-called *underlying*, interface. 23, 44, 45

associated value A value that is associated with something, like a key. For example, a Dictionary is a data structure that stores keys that are used to look up associated values. 53–55, 79

Associative Array See: Dictionary. 53

axiom A function which consists of a sequence of statements. It holds if none of the assertions fail. Axioms are used in concepts to declare invariants. 20, 21, 79

block In the context of imperative programs, a sequence of statements. 3

BTA A program analysis which aims to find out which values can be evaluated at compile-time. Used in partial evaluation.. 29

code as data A slogan and insight associated with people who like the Lisp family of languages. To treat code as data means to make programs that take code as input and give code as output; i.e., the data that is being manipulated is code. Program transformations in general treat code as data. 15, 80

cohesion In the context of bookkeeping, consistency between representational levels. This means that all representational levels should agree on the same change to the code. 70

compile-time At the time of compiling the program. Contrast with runtime, i.e., at the time of running the program. 15, 25, 28, 43, 62, 79, 82

concept An interface extended with axioms. 20, 21

constant folding A compiler optimisation that evaluates constant expressions at compile-time instead of leaving them unevaluated until runtime. 15, 25, 28, 49

data at rest A term for inactive data in IT, in contrast to *data in use* and *data in transit*. But for our purposes we use this to refer to simple storage of data without any dynamic checks to maintain consistency or other properties. 70

deterministic transformation A transformation that is a well-defined function which gives the same output when given the same input. Contrast with non-deterministic transformation. 15, 81

Dictionary An abstract data type which exposes operations to store, remove, and look up key–value pairs. 53–55, 73, 79, 81, 82

DSL A programming language specialised to some particular domain or application. Contrast with general-purpose languages.. 57, 58, 62, 65

first-class In the context of programming language features or capabilities, something which is supported directly by the language. For example, in a programming language with structured control flow, the *if-then-else* construct tends to be a first-class feature. In a programming language with only goto statements it would be implemented as a sort of goto *pattern*. 19, 80

fresh parameter See: fresh variable. 49

fresh variable A fresh variable is a variable that is introduced into a scope which does not conflict with or shadow any of the already-defined variables in that scope. Introducing a fresh veriable can be useful when lifting a constant value to a variable value. 49, 80

Git note A piece of metadata that can be associated with a Git commit, and indeed more generally to any Git object. The intended use-case is to be able to save and update information and metadata about a commit. Perhaps the biggest advantage compared to storing the same data in a commit message is that a Git note is mutable, while changing a commit message changes the hash of a commit. 68

glue language A programming language used to *glue* different components together, where the different components might have no connection or knowledge of each other. For example, a shell language can be used to combine different programs which might be written in different languages. Glue languages have historically mostly been scripting languages. 62

homoiconicity A concrete syntax that is close to the abstract syntax of the language. Having this feature makes treating code as data much easier. 14

HPC Applications of computing which puts a high demand on the resources used for the computations, for example by having to process a lot of data in a reasonable time. The high demands means that things like supercomputers and paralell processing between clusters of computers are put to use, instead of regular servers or desktops. Put to use in domains like simulation of physical phenomena.. 1, 19, 20, 23

IDE A fleshed-out environment for programming. Normally consists of a code editor, build automation tools, and a debugger. It is *integrated* in the sense that many tools come in the same package; contrast with using a loose collection of different tools for things like code editing, building, debugging, etc.. 3

inline To replace a function call with the body of the function at the call site. 6, 29, 65, 82

interface A collection of abstract types and operations on those types. A feature of many programming languages that allows things like several implementations for the same abstract behaviours. 79

IR A representation of the input code (source code) that the compiler uses internally, i.e. during compilation.. 4

Magnolia item Anything that can be declared. I.e., functions, types, concepts, programs. 60, 61

Magnolia PD "Magnolia Plus Directives". A superset of Magnolia extended with directives. 64–66

Map See: Dictionary. 53

named parameter A parameter to a function that is named, and associated with the argument at the call site by putting it next to the argument. This allows for more descriptive function calls. 60

non-deterministic transformation A transformation that relies on opaque heuristics, and so cannot be relied on to produce the same output given the same input. Contrast with deterministic transformation. 16, 80

operation In the context of Magnolia, either a function, predicate, or procedure. 20, 21, 23, 55, 59, 81

over-specify To implement a program which solves a problem in a too rigid way, which makes it so that there is less room for things like compiler optimisation. An example is to implement a transformation over an array in an imperative way, which can preclude optimisations that take advantage of things like parallelism since data dependencies are obscured. 20

partial evaluation A source-to-source program transformation technique for specializing programs with respect to parts of their input [Consel and Danvy, 1993, p. 1]. 2, 25–28, 30, 31, 34, 41, 42, 47, 52, 73, 79

runtime At the time of running the program. Contrast with compile-time, i.e., at the time of compiling the programming. 79

Set In the context of programming, an abstract datatype that has the same behavior as Sets in set theory. This is a collection of values where no value occurs twice (no duplicates). Depending on the programming language that it is implemented in, there might be other constraints not found in set theory. For example, in a statically typed programming language, it might be demanded that the values in the set all have the same type, which is not a requirement in set theory. 53–55, 73

sigil In computer programming in general, a symbol attached to a variable name, usually prefix. For our purposes, it is a symbol prefixed to any alphanumeric name. 60

step In the context of a transformation session, a discrete point which consists of either a transformation, or normal coding (hand-coding). Conceptually one can consider a transformation step to be stored as the sequence of transformation commands that were done in that step, while a coding step can be stored as a set of patches. 67, 70, 71

Symbol Table See: Dictionary. 53

transformation session a user-initiated and user-stopped period of time where the user works on the programs, using both transformations and manual coding, or *hand-coding*. 67, 82

undefined behavior In the context of programming language semantics, behavior that is explicitly left undefined in order to lend flexibility to implementors of the language. This in principle means that the program can do "whatever it wants" once undefined behavior is encountered while executing the program. An example of undefined behavior is the behavior (result) of addition on a fixed-width integer in the case when the result is too large for the integer to contain; although on modern computers the behavior is often *wrap around*, this is undefined behavior in the C specification. 19

unfold See: inline. 29

unified diff A patch format. Widely used in software development for exchanging changes between files. See the patch(1) and diff(1) utilities. 70

VCS A system for managing different versions of digital artifacts, where each version might be the state of the artifacts at some point in time. Often used to manage source code.. 2, 58, 67–71

wrapper type A composite type that only consists of a single type. In Magnolia, this is a struct consisting of only one member. 46

A

Modulus Group

```
/**
 * Three different designs for a modulus group.
 *
 * Can we automatically transform between
 * these using inlining/partial evaluation?
 * - or some other set of rules?
 * Which one of these is the best starting point?
 *
 * @author Magne Haveraaen
 * @since 2015-08-17
 */
package BasicGeneral.ModulusGroupCxx
imports
  Basic.FiniteInteger,
  BasicCxx.IntegerCxx,
  Mathematics.Group;


/** Modulus group on constant {@link bn}. */
implementation mgConstant = {
  require signature(BoundedInteger);

  /** Base number for the modulus group. */
  require function bn() : Integer;

  type MG = struct{ var v:Integer; };
  predicate datainvariant ( x:MG ) = zero() <= x.v && x.v < bn();

  function plus ( a:MG, b:MG ) : MG =
    MG{ v = ( a.v + b.v ) % bn() };
  function uminus ( a:MG ) : MG =
    MG{
      v = if a.v == zero()
      then zero()
```

```
    else bn() - a.v
    end
   };
  function zeroMG () : MG =
   MG{ v = zero() };
};
satisfaction mgConstant_is_Group = {
  use BoundedInteger;

  function bn() : Integer;
  axiom positiveBnAxiom () {
   assert zero() < bn();
  };
} with mgConstant models {
  use Group[ T => MG, zero => zeroMG ];

  axiom preserveDatainvariantNullary () {
   assert datainvariant(zeroMG());
  };
  axiom preserveDatainvariantUnary ( x:MG ) {
   assert datainvariant(x)
    => datainvariant(uminus(x));
  };
  axiom preserveDatainvariantBinary ( x:MG, y:MG ) {
   assert datainvariant(x)
    && datainvariant(y)
    => datainvariant(plus(x,y));
  };
};
program mgConstant7Cxx = {
  use boundedInteger8bitCxx;
  function bn() : Integer
   = one() + one() + one()
    + one() + one() + one() + one();
  use mgConstant_is_Group;
};



/**
 * Modulus group where the base number is given
 * by an argument {@code bn} to the constructors.
 */
implementation mgArgument = {
  require signature(BoundedInteger);
```

```
  type MG = struct{ var v:Integer; var bn:Integer; };
  predicate datainvariant ( x:MG ) = zero() <= x.v && x.v < x.bn;

  /** Both arguments must belong to the same
   * modulus group, i.e., have the same base
   * number {@code bn}.
   */
  function plus ( a:MG, b:MG ) : MG guard a.bn == b.bn =
    MG{
      v = ( a.v + b.v )
        % a.bn, bn = a.bn
    };
  function uminus ( a:MG ) : MG =
    MG{
      v = if a.v == zero()
      then zero()
      else a.bn - a.v end, bn = a.bn
    };
  function zeroMG ( bn:Integer ) : MG
     guard
       zero() < bn = // && gplus(bn-one(),bn-one()) =
    MG{
      v = zero(), bn=bn
    };
};
satisfaction mgArgument_is_Group = {
  use BoundedInteger;

  function bn() : Integer;
  axiom positiveBnAxiom () {
    assert zero() < bn();
  };
} with {
  use mgArgument;
  function zeroMG(): MG = zeroMG(bn);
} models {
  use Group[ T => MG, zero => zeroMG ];

  axiom preserveDatainvariantNullary ( bn:Integer ) {
    assert datainvariant(zeroMG(bn));
  };
  axiom preserveDatainvariantUnary ( x:MG ) {
    assert datainvariant(x)
      => datainvariant(uminus(x));
```

```
  };
  axiom preserveDatainvariantBinary ( x:MG, y:MG ) {
    assert datainvariant(x)
      && datainvariant(y)
      => datainvariant(plus(x,y));
  };
};
program mgArgumentCxx = {
  use boundedInteger8bitCxx;
  function bn() : Integer
    = one() + one() + one() + one()
      + one() + one() + one();
  use mgArgument_is_Group;
};


/** Modulus group on constant {@link bn}. */
implementation mg7 = {
  require signature(BoundedInteger);

  /** Base number for the modulus group. */
  function seven() : Integer
    = one() + one() + one() + one()
      + one() + one() + one();

  type MG = struct{ var v:Integer; };
  predicate datainvariant ( x:MG )
    = zero() <= x.v && x.v < seven();

  function plus ( a:MG, b:MG ) : MG =
    MG{ v = ( a.v + b.v ) % seven() };
  function uminus ( a:MG ) : MG =
    MG{
      v = if a.v == zero()
      then zero()
      else seven() - a.v end
    };
  function zeroMG () : MG =
    MG{ v = zero() };
};
satisfaction mg7_is_Group = {
  use BoundedInteger;
} with mg7 models {
  use Group[ T => MG, zero => zeroMG ];
```

```
  axiom preserveDatainvariantNullary () {
    assert datainvariant(zeroMG());
  };
  axiom preserveDatainvariantUnary ( x:MG ) {
    assert datainvariant(x)
      => datainvariant(uminus(x));
  };
  axiom preserveDatainvariantBinary ( x:MG, y:MG ) {
    assert datainvariant(x)
      && datainvariant(y)
      => datainvariant(plus(x,y));
  };
};
program mg7Cxx = {
  use boundedInteger8bitCxx;
  use mg7_is_Group;
};
```

B

Dictionary and Set

```
 * Basic specifications of a dictionary abstraction.
 * @author Magne Haveraaen
 * @since 2014-04-12
 */
package Collections.Dictionary
imports Indexable.Indexable;

/**
 * A dictionary are indexable structures indexed by
 * keys with data elements. The terminology is more
 * related to data bases than to arrays.
 */
concept Dictionary = {
  type Key;
  type Data;
  type Dictionary;

  function createEmpty () : Dictionary;
  function insertNew ( d:Dictionary, k:Key, e:Data )
     : Dictionary;
  function replaceData ( d:Dictionary, k:Key, e:Data )
    : Dictionary;
  predicate isPresent ( d:Dictionary, k:Key );
  function find ( d:Dictionary, k:Key ) : Data;
  function remove ( d:Dictionary, k:Key )
    : Dictionary;

  axiom createEmptyNotPresent ( k:Key ) {
    assert ! isPresent ( createEmpty(), k );
  };
  axiom removeNotPresent ( d:Dictionary, k:Key ) {
    assert ! isPresent ( remove(d,k), k );
  };
  axiom changeIsPresent (
```

```
    d:Dictionary, k:Key, e:Data
    ) {
    assert isPresent(
      insertNew(d,k,e), k
    );
    assert find(
      insertNew(d,k,e), k
      ) == e;
    assert isPresent(
      replaceData(d,k,e), k
      );
    assert find(
      replaceData(d,k,e), k
      ) == e;
  };
  axiom removeOtherPresent (
    d:Dictionary,
    k:Key,
    k2:Key
    ) guard k != k2 {
    assert isPresent (
      remove(d,k), k2
      ) <=> isPresent(d,k2);
  };
  axiom changeIsPresent (
    d:Dictionary, k:Key, e:Data, k2:Key
    ) guard k != k2 {
    assert isPresent(
      insertNew(d,k,e), k2
      ) <=> isPresent(d,k2);
    assert find(
      insertNew(d,k,e), k2
      ) == find(d,k2);
    assert isPresent(
      replaceData(d,k,e), k
      ) <=> isPresent(d,k2);
    assert find(
      replaceData(d,k,e), k2
      ) == find(d,k2);
  };

};

satisfaction Dictionary_models_PartialIndexable =
Dictionary models
```

```
PartialIndexable[
  A => Dictionary,
  I => Key,
  E => Data,
  get => find,
  accessible => isPresent
];
```

```
/*
* Manual slice of Dictionary in order to obtain a Set
* concept.
*/

package Collections.SetSlice

imports
  Collections.Indexable;

/**
* Set concept
*/
concept SetSlice = {
  type Key;
  // Sliced
  // type Data;
  type Dictionary;

  function createEmpty () : Dictionary;
  // Sliced
  function insertNew ( d:Dictionary, k:Key /*, e:Data*/)
    : Dictionary;
  function replaceData ( d:Dictionary, k:Key /*, e:Data*/ )
    : Dictionary;
  predicate isPresent ( d:Dictionary, k:Key );
  // Sliced
  // function find ( d:Dictionary, k:Key ) : Data;

  function remove ( d:Dictionary, k:Key )
    : Dictionary;

  axiom createEmptyNotPresent ( k:Key ) {
    assert ! isPresent ( createEmpty(), k );
  };
  axiom removeNotPresent ( d:Dictionary, k:Key ) {
    assert ! isPresent ( remove(d,k), k );
```

```
  };
  axiom changeIsPresent (
    d:Dictionary, k:Key /*, e:Data*/
    ) {
    assert isPresent( insertNew(d,k), k);
    // Sliced
    // assert find( insertNew(d,k,e), k) == e;
    assert isPresent(
      replaceData(d,k /*,e*/ ), k
    );
    // Sliced
    // assert find( replaceData(d,k,e), k) == e;
  };
  axiom removeOtherPresent (
    d:Dictionary, k:Key, k2:Key
    ) guard k != k2 {
    assert isPresent (
      remove(d,k), k2
      ) <=> isPresent(d,k2);
  };
  axiom changeIsPresent (
    d:Dictionary, k:Key /*, e:Data*/, k2:Key
    ) guard k != k2 {
    assert isPresent( insertNew(d,k), k2) <=> isPresent(d,k2);
    // Sliced
    // assert find( insertNew(d,k,e), k2) == find(d,k2);
    // Sliced
    assert isPresent(
      replaceData(d,k /*, e*/), k
      ) <=> isPresent(d,k2);
    // assert find(
    //  replaceData(d,k,e), k2
    //  ) == find(d,k2);
  };

};

// Slicing: PartialIndexable to PartialIndexableSlice
//  See: manually sliced file Indexable.mg.
satisfaction Dictionary_models_PartialIndexableSlice =
SetSlice models
PartialIndexableSlice[ A => Dictionary, I => Key,
// Sliced
// E => Data, get => find,
accessible => isPresent ];
```

```
// Taken from src/Indexable/Indexable.mg
// Manual slice of PartialIndexable, due to a satisfaction in the original
// Dictionary.mg which was sliced to SetSlice.mg.
package Collections.Indexable;

/**
 * The indexable specification, <code>get</code> is the
 * indexing operation. The (implicit) guard may limit
 * the actual indices to some subset of the type.
 */
concept Indexable = {
  /** the indexable (array) type */
  type A;
  /** the index type */
  type I;
  /** the element type*/
  // Sliced
  // type E;

  /** The indexing function. */
  // Sliced
  // function get (a:A, i:I) : E;
};


/**
 * The partial indexable specification, which guards the
 * {@code get} by an accessiblity predicate.
 */
concept PartialIndexableSlice = {
  /**
   * An indexable: types {@code A}, {@code I},
   * {@code E} and {@code get} operation.
   */
  use Indexable;

  /**
   * Checking if an index is accessible for getting
   * elements from the indexable.
   */
  predicate accessible ( a:A, i:I ) ;

  /** Guarding the get function. */
  // Sliced
  // protect function get (a:A, i:I) : E guard accessible(a,i);
```

```
};
```