HERMIT: Mechanized Reasoning during Compilation in the Glasgow Haskell Compiler

By

Andrew Farmer

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Chairperson Dr. Andy Gill

Dr. Perry Alexander

Dr. Prasad Kulkarni

Dr. James Miller

Dr. Christopher Depcik

Date Defended: April 30, 2015

The Dissertation Committee for Andrew Farmer
certifies that this is the approved version of the following dissertation:

HERMIT: Mechanized Reasoning during Compilation in the Glasgow Haskell Compiler

____
Chairperson Dr. Andy Gill

Date approved: __________________________

Abstract

It is difficult to write programs which are both correct and fast. A promising approach, functional programming, is based on the idea of using pure, mathematical functions to construct programs. With effort, it is possible to establish a connection between a specification written in a functional language, which has been proven correct, and a fast implementation, via program transformation.

When practiced in the functional programming community, this style of reasoning is still typically performed by hand, by either modifying the source code or using pen-and-paper. Unfortunately, performing such *semi-formal* reasoning by directly modifying the source code often obfuscates the program, and pen-and-paper reasoning becomes outdated as the program changes over time. Even so, this semi-formal reasoning prevails because formal reasoning is time-consuming, and requires considerable expertise. Formal reasoning tools often only work for a subset of the target language, or require programs to be implemented in a custom language for reasoning.

This dissertation investigates a solution, called HERMIT, which mechanizes reasoning during compilation. HERMIT can be used to prove properties about programs written in the Haskell functional programming language, or transform them to improve their performance. Reasoning in HERMIT proceeds in a style familiar to practitioners of pen-and-paper reasoning, and mechanization allows these techniques to be applied to real-world programs with greater confidence. HERMIT can also re-check recorded reasoning steps on subsequent compilations, enforcing a connection with the program as the program is developed.

HERMIT is the first system capable of directly reasoning about the full Haskell language. The design and implementation of HERMIT, motivated both by typical reasoning tasks and HERMIT's place in the Haskell ecosystem, is presented in detail. Three case studies investigate HERMIT's capability to reason in practice. These case studies demonstrate that semi-formal reasoning with HERMIT lowers the barrier to writing programs which are both correct and fast.

Acknowledgements

I read somewhere, once, that a dissertation takes a village. That is certainly true in my experience. I am indebted to a great many people, both professionally and personally, over the last six(!) years.

Foremost, I would like to thank my advisor, Andy Gill, for providing me many opportunities I did not even realize existed, and for tolerating my occasional divergences. I have learned an incredible amount in my time as his student, and have always valued his guidance and support. I definitely owe him a non-trivial amount of beer.

I was fortunate to have Neil Sculthorpe as a collaborator for much of HERMIT's development. I learned a lot from Neil, especially in regards to writing about research. Without his excellent work, both on KURE and HERMIT itself, HERMIT would not exist as it does.

Thanks also to HERMIT's first users: Michael Adams, Conal Elliott, and Paul Liu. Their feedback was invaluable, and they were kind enough to put up with HERMIT's ever-changing APIs breaking their code. Thanks to Jim Hook for hosting me for a semester at Portland State and enabling my collaboration with Michael Adams. To my peers and mentors, past and present, in the CSDL lab (and elsewhere): Perry, Prasad, Garrin, Ed, Nick, Mark, Evan, Wes, Megan, Brigid, Mike J, Nathan, Pedro, Laurence, Brent, Richard, Kevin, Tristan, Mike S, Justin, Jason, Ryan, Bowe, and Brad. I learned something from each of you, and appreciate having been able to work with you all at some point. Also thanks to the National Science Foundation, which partially supported HERMIT under grants CCF-1117569 and DGE-0742523.

I could not have accomplished a great many things in life without the support of my parents, who are some of the most selfless people I know. Thanks Mom and Dad, for everything. Thanks also to my brother, Ben, who is someone I look up to, both literally and figuratively. To some great friends: Austin, Bob, Michael, John, Derick, Amy, Jys, Jess, and a great many others. To Larryville, for all the shenanigans, punctuated with the occasional running. Finally, to Karen, for putting up with my absentmindness, for making sure I ate something besides fast food, and for cheering me up when I was stressed out. I love you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Writing a program which is both correct and fast is difficult. Often, the clear, concise, 'obviously correct' version of a program does not perform well. This is because such programs are, by nature, written at a high level of abstraction. If better performance is desired, the program is usually altered to specialize it in some way. Doing so results in a program which is typically more verbose, less clear, and less obviously correct.

With effort, it is possible to establish that the fast version of the program is a *refinement* of the correct version, providing assurance that the fast version is still correct. One example is the formal verification of the seL4 microkernel [Klein et al., 2009], where an executable specification written in the functional language Haskell [Peyton Jones, 2003] was transliterated into Isabelle/HOL [Paulson, 1989, Wenzel and Berghofer, 2012] using a custom translator. The result was then formally connected to a fast C implementation. This required over 200,000 lines of proof and over 20 person-years of effort.

Even in smaller examples, formal verification such as this is often time-consuming, requiring considerable expertise. Formal reasoning tools often require programs to be implemented in a custom language for reasoning. In the case of seL4, the executable Haskell specification had to be translated into Isabelle/HOL, and this translation itself later had to be verified. Tools which actually target the desired language often only work for a subset of the language, making them difficult to apply to existing programs, which may not have been written with the restrictions of the tool in mind.

Due to the high costs of formal verification, the process is often instead performed *semi-formally*, without a formal logic or machine checking the reasoning steps. This semi-formal reasoning is easier if the programs are implemented in a functional, rather than imperative, programming language. In functional languages, computation is expressed using pure functions applied to immutable, persistent values. Pure functions, the kind found in mathematics, cannot mutate their environment. Reasoning about the behavior of pure programs is simpler due to the absence of this mutable state [Hughes, 1989].

When practiced in the functional programming community, semi-formal reasoning is often performed by hand [Sculthorpe and Hutton, 2014, Gibbons and Hutton, 2005, Bird, 2010]. The source code of the correct program is transformed into the fast version using a series of correctness-preserving steps, in a process known as *program transformation*. This offers a high assurance of correctness, but results in an obfuscated program. By destructively modifying the source, access to the intermediate results of the transformations is lost. Future modifications to the program must be made on the now-obfuscated version, lest the transformations be painstakingly repeated.

When an attempt is made to record the intermediate results, it is typically done alongside the code, either in comments or in an entirely separate document. Such pen-and-paper reasoning must be kept up-to-date as the program changes over time, or the correctness assurances will be lost. With nothing enforcing the connection between the recorded reasoning and the program, keeping the reasoning up-to-date is an error-prone, manual process.

Nevertheless, semi-formal reasoning is popular due to its simplicity. This dissertation defends the thesis that mechanizing semi-formal reasoning, during compilation, lowers the burden of writing programs which are both correct and fast. It does so by investigating a solution, called HERMIT, which mechanizes reasoning from within the Glasgow Haskell Compiler (GHC) [GHC Team, 2014]. Haskell is a strongly typed, pure, non-strict functional programming language [Marlow, 2009]. Semi-formal reasoning is popular in the Haskell community. GHC is the flagship Haskell compiler, representing the *de facto* Haskell language standard.

The decision to operate from within GHC makes HERMIT the first system capable of interactively reasoning about the full Haskell language. Other tools for reasoning about Haskell programs typically operate at the source level. They must necessarily contend with a large amount of syntax, and rely on type inference to reason about types. HERMIT targets the syntactically smaller intermediate language used by GHC's optimizer, called GHC Core. GHC Core features explicit, local type information, but is sufficiently similar to Haskell that the same reasoning techniques apply.

HERMIT can be used to prove properties about programs written in Haskell, or transform them to improve their performance, in a style familiar to practitioners of semi-formal reasoning.[1] It is important that HERMIT is able to match the ease and abstraction inherent in pen-and-paper reasoning, as that is one of the major advantages of reasoning semi-formally. The included examples and case studies demonstrate that HERMIT largely succeeds at this.

When reasoning by-hand, working with code larger than a few lines is both tedious and error-prone. Syntactic manipulations muddle the clarity of the reasoning itself, and mistakes are easily made. HERMIT mechanizes this manipulation, allowing the programmer to focus on what needs to be done, rather than how to do it. Mechanization allows semi-formal techniques to be applied to real-world programs with greater confidence. HERMIT can also re-check recorded reasoning steps on subsequent compilations, enforcing a connection with the program as the program is developed.

1.1 Reasoning

HERMIT is a practical system, designed to be applied to real Haskell programs and to accomplish real reasoning tasks which are currently performed semi-formally by the Haskell community. This section provides concrete examples of three such semi-formal reasoning tasks. Each of the case

[1] This dissertation uses the term "proof" as shorthand for the notion of making a systematic argument about correctness using an informal logic, as is done in traditional mathematical proofs expressed using natural language. Chapter 5 elaborates on this notion of proof.

studies included in this dissertation addresses one of these tasks, evaluating HERMIT's effectiveness at mechanizing them.

1.1.1 Proving Properties

It is common for programmers to state expected invariants about the code they are writing. These properties may be part of the program specification, or generated documentation. They often serve as sanity checks, or to facilitate a simpler implementation of key functionality. In Haskell, these properties are common alongside type classes. That is, a given class states that all valid instances must satisfy certain properties.

GHC allows a subset of these properties to be stated as rewrite rules which the optimizer attempts to apply during compilation. This feature is commonly used by library writers to specify library-specific optimizations. The rewrites themselves are typechecked, but no attempt is made to verify their correctness. The assumption is that the programmer specifying the rules has verified them separately.

Research into automated testing of these properties has spawned a number of tools to check them mechanically. The most successful of these in the Haskell eco-system is Quickcheck [Claessen and Hughes, 2000], and variants of Quickcheck are used in other languages, such as Erlang and C [Arts et al., 2008, Arts and Castro, 2011]. Quickcheck allows the programmer to state equational properties about their library, testing these properties on randomly-generated test vectors. As an example, the following Quickcheck property states that reversing a list twice is equivalent to the original list.

$$prop\_reverse\_reverse\ xs = reverse\ (reverse\ xs) == (xs :: [Int])$$

To test the property, Quickcheck generates random test vectors based on the types of the property's arguments. In this case, xs has been ascribed the type $[Int]$, so Quickcheck generates random lists of integers. User-defined datatypes can also be generated randomly by defining an appropriate type class instance for the type.

This automated random testing is good at finding small counter-examples, but a successful test is obviously not a proof of correctness. Combining random testing with code coverage increases the assurance that all paths through the code were exercised by the test, but even that may not be sufficient to catch subtle bugs.

Using the following definition of reverse, the property can actually be proven using structural induction on the list type and simple equational reasoning.

$$\begin{array}{l}
\mathit{reverse} :: [\,a\,] \to [\,a\,] \\
\mathit{reverse}\ [\,] \qquad\quad = [\,] \\
\mathit{reverse}\ (x : \mathit{xs}) = \mathit{reverse}\ \mathit{xs} +\!\!+ [\,x\,]
\end{array}$$

Since lists are a lifted type, the first base case is when xs is $\bot$.

$$\begin{array}{l}
\quad \mathit{reverse}\ (\mathit{reverse}\ \bot) \\
\qquad \{\text{reverse is strict in xs}\} \\
= \mathit{reverse}\ \bot \\
\qquad \{\text{reverse is strict in xs}\} \\
= \bot
\end{array}$$

The other base case is when xs is the empty list.

$$\begin{array}{l}
\quad \mathit{reverse}\ (\mathit{reverse}\ [\,]) \\
\qquad \{\text{unfold reverse}\} \\
= \mathit{reverse}\ [\,] \\
\qquad \{\text{unfold reverse}\} \\
= [\,]
\end{array}$$

Finally, the inductive case. The argument to reverse is a non-empty list and the inductive hypothesis is that the property holds for ys, the tail of the list.

$$\begin{array}{l}
\quad \mathit{reverse}\ (\mathit{reverse}\ (y : \mathit{ys})) \\
\qquad \{\text{unfold reverse}\} \\
= \mathit{reverse}\ (\mathit{reverse}\ \mathit{ys} +\!\!+ [\,y\,]) \\
\qquad \{\mathit{reverse}\ (\mathit{xs} +\!\!+ \mathit{ys}) = \mathit{reverse}\ \mathit{ys} +\!\!+ \mathit{reverse}\ \mathit{xs}\} \\
= \mathit{reverse}\ [\,y\,] +\!\!+ \mathit{reverse}\ (\mathit{reverse}\ \mathit{ys}) \\
\qquad \{\text{inductive hypothesis}\} \\
= \mathit{reverse}\ [\,y\,] +\!\!+ \mathit{ys} \\
\qquad \{\text{unfold reverse}\} \\
= \mathit{reverse}\ [\,] +\!\!+ [\,y\,] +\!\!+ \mathit{ys} \\
\qquad \{\text{unfold reverse}\} \\
= [\,] +\!\!+ [\,y\,] +\!\!+ \mathit{ys} \\
\qquad \{\text{unfold } +\!\!+\} \\
= [\,y\,] +\!\!+ \mathit{ys} \\
\qquad \{\text{unfold } +\!\!+\} \\
= y : \mathit{ys}
\end{array}$$

While performing this proof by hand is straightforward, it is informal in a number of ways. First, there is no check against simple oversight on the part of the programmer. For instance, such proofs often fail to consider the case for $\perp$. Second, it uses a naive, inefficient definition of $reverse$. This is common practice, as the naive definition requires less syntactic manipulation, which is tedious to do by hand. However, this means the property has not been proven for the actual implementation of $reverse$ which is used in practice. It is possible to prove two implementations equivalent by appealing to more reasoning. In general, however, two implementations may be semantically different in subtle ways, such as when dealing with partial or infinite values. Third, an auxiliary lemma stating how $reverse$ distributes over $+\!\!+$ is assumed. Such lemmas may appear obvious, but should also be proven.

HERMIT allows the programmer to perform such a proof similarly to how it it is done by hand, but with mechanical support. HERMIT's notion of structural induction automatically considers cases where partial values matter. Auxiliary lemmas must be stated (and ideally proven) explicitly before HERMIT permits their use. HERMIT's tools for mechanizing the transformation steps themselves lower the burden of manipulation, allowing the proof to be performed on the actual implementation with less tedium. HERMIT's proof-checking is integrated into compilation, meaning the proof can be kept up-to-date as the program changes.

1.1.2 Domain-Specific Optimizations

Modern compilers expend considerable effort to improve the performance of target code. This process, known as *optimization*, is especially important for pure functional programming languages, where the semantic model of computation differs significantly from the execution model of the typical machine. Fortunately, pure functional programming languages are particularly amenable to aggressive optimization due to the absence of mutable state [Peyton Jones and Santos, 1998].

Compiler optimizations fall on a spectrum of generality:

1. The most general optimizations apply to programs written in any language. Consider constant folding, which is the elimination of computation which is completely statically known [Weg-

man and Zadeck, 1991]. Regardless of language, it is better to perform such a computation once, at compilation time, rather than every time the program is executed.

2. More specific optimizations might apply only to programs in a certain class of languages. One example from functional languages is lambda lifting [Johnsson, 1985], which may avoid the repeated allocation of a closure by turning it into a top-level function.

3. More specific still are those that apply to a specific implementation of a functional language. For example, GHC implements Haskell's type class method dispatch using implicit dictionary parameters [Jones, 1995]. Thus, its optimizer is keen to specialize functions to the (statically-known) dictionary arguments.

4. Even more specific, an optimization may only apply to a certain library, and programs which make use of that library. An example is Stream Fusion [Coutts, 2010], which optimizes computations involving sequence data types, such as lists. The Stream Fusion technique generally benefits programs that rely heavily on lists, but has no effect on programs which do not. In fact, it may have a negative effect on certain programs, so it must be selectively enabled.

5. Most specifically, an optimization may target only a specific program. Such an optimization may be a form of specification refinement, where a clear, but inefficient, program is systematically transformed into a more efficient, but obscure program.

Traditionally, a line is drawn between items 3 and 4. Those above the line are considered "generally useful" and included in the compiler's repertoire. Those below the line are *domain-specific optimizations*, as they only apply to a narrow class of programs and may pessimize programs to which they do not apply. Since they are neither widely applicable or generally positive, these optimizations are usually not implemented by the compiler.

However, domain-specific optimizations can have extraordinarily positive effects on programs for which they are designed. As an example, the Stream Fusion technique, mentioned above, regularly provides greater than 50% speedup on first-order, list-heavy programs [Coutts et al., 2007].

Some compilers provide a means for specifying such an optimization. Indeed, GHC offers two facilities for specifying domain-specific optimizations: rewrite rules [Peyton Jones et al., 2001] and plugins, overviewed in Sections 2.2.3 and 2.1, respectively. Rewrite rules are easy to use, but their power is limited. For instance, they can only pattern match on function application, meaning other syntactic constructs, such as case expressions, cannot be transformed. GHC plugins are powerful, providing direct access to the compiler's intermediate representation of the program. Writing a plugin, however, is daunting. It requires specialized knowledge of the compiler's internal data structures and methods. The plugin author must ensure that all invariants on these structures are maintained, and that the compiler's internal bookkeeping is accurate and up-to-date.

HERMIT, itself a plugin, offers the power of plugins without requiring the user to be an expert on the internals of GHC. Rather than manipulating the compiler's data structures directly, the HERMIT user constructs an optimization by combining primitive transformations. Each primitive ensures that it maintains the invariants expected by the compiler. HERMIT's interface is both interactive and scriptable. The details of an optimization can be explored interactively, then it can be saved as a script and refined to address a broader range of programs. This makes HERMIT well-suited as a prototyping tool for optimizations.

1.1.3 Calculational Programming

Combining the notions of proving properties and writing domain-specific optimizations is the practice of *calculational programming* [Hu et al., 2006, Bird, 2010]. This is when the programmer writes a declarative specification of the program, then systematically refines it into an efficient implementation. The goal of the original program is to be clear, concise, and "obviously correct". The goal of the refinement is to arrive at an efficient version of the program that is equivalent to the original program in terms of correctness, but with better performance.

As an example, consider the following "obviously correct" definition of the $mean$ function:

```
mean :: [Double] → Double
mean xs = sum xs / length xs
```

This definiton is inefficient because it traverses the input list twice (once by *sum*, once by *length*). Operationally, this means the list is resident in memory longer than necessary. A more efficient version computes the sum and length of the list in a single pass:

```
mean :: [Double] → Double
mean xs = sm / len
  where (sm, len) = sumlength xs
           sumlength :: [Double] → (Double, Double)
           sumlength []       = (0, 0)
           sumlength (d : ds) = case sumlength ds of
                                  (s, l) → (d + s, 1 + l)
```

This definition is less obviously correct, but considerably more efficient[2]. Importantly, using a series of equational transformations, one can derive, or calculate, the complicated, efficient definition from the simple, declarative one. This particular derivation is performed interactively using HERMIT in Section 4.1.

Calculational programming is also known as *equational reasoning* (in the functional programming community) [Hutton, 2007, Chapter 13] and *specification refinement* (more broadly). This dissertation uses the term ‘calculational programming’ to be more precise in the presence of the discussion of other types of reasoning. HERMIT’s ability to prove program properties and script transformations makes it well-suited to this form of program refinement.

1.2 Contributions

Specifically, this dissertation makes the following contributions:

- It presents the design and implementation of HERMIT, a compile-time reasoning assistant for the Haskell language. HERMIT is the first such system capable about reasoning about the entire Haskell language, including language extensions. There are many pragmatic issues to be solved when implementing a system like HERMIT, and this dissertation presents the details of HERMIT’s solutions.

[2] Of course, this version is still not tail-recursive. Further refinements exist.

- It demonstrates that semi-formal reasoning can be mechanized at compile time at a level of abstraction comparable to performing it by hand. This is evidenced by the examples in Sections 4.1 and 5.1 and the case studies in Chapters 6 and 8, which find that transformation and proof scripts in HERMIT largely correspond to their by-hand counterparts in both length and form.

- It provides evidence that an interactive means of exploring optimizations, such as HERMIT's interactive interface, reduces the effort in developing such optimizations. HERMIT is used to prototype optimizations in the case study in Chapter 7 and two projects in Chapter 9.

- It demonstrates that TrieMaps can be extended to support first-order pattern matching in the map key. This functionality is used to implement HERMIT's primitive expression folding capability, which is central to several primitive transformations. A TrieMap is data structure which implements a finite map whose keys are finite sequences, such as strings. TrieMap implementations exist in several languages, including Haskell [Wasserman, 2013] and Scala [Prokopec et al., 2012], but none have yet been extended in this way.

- The case study in Chapter 7 solves a long-standing practical limitation of the Stream Fusion shortcut deforestation system by modifying GHC's optimizer, via HERMIT, to fuse a key higher-order sequence combinator. It allows users of Stream Fusion to write higher-order sequence processing pipelines using modular, reusable combinators, instead of writing a hand-fused loop, without loss of performance. Lifting this limitation allows Stream Fusion to outperform competing systems in many cases in which it previously underperformed, broadening its appeal.

1.3 Organization

The remainder of this dissertation is organized as follows.

Background

- Chapter 2 provides technical background sufficient to make this dissertation self-contained. This includes an introduction to GHC plugins, the GHC Core language, and the University of Kansas' strategic rewriting language, KURE.

HERMIT's Design and Implementation

- Chapter 3 presents the overall architecture of HERMIT, including the design of the HERMIT plugin and HERMIT's low-level transformation manager. It also describes HERMIT's two main user interfaces, the Plugin DSL and the HERMIT Shell.

- Chapter 4 details HERMIT's support for program transformation. It begins with an example transformation to provide intuition for HERMIT's capabilities. It then describes HERMIT's support for transforming GHC Core programs using the KURE strategic rewriting language, and the ability to rewrite expressions using other expressions as patterns. It concludes by surveying the large number of primitive transformations that HERMIT provides.

- Chapter 5 describes HERMIT's support for proving program properties. It presents an interactive proof example before describing HERMIT's encoding of properties in detail. Proof in HERMIT is accomplished by rewriting, in the style of natural deduction, and key transformations relevant to proving in HERMIT are examined in detail.

Primary Evidence

The case studies in Chapters 6 and 7 provide primary evidence of the utility of HERMIT.

- Chapter 6 presents a case study which uses HERMIT to prove type-class laws for data types in the Haskell standard libraries. These laws are properties which are expected to hold for instances of the type class but are not enforced by the type system. They are instead left as proof obligations to the programmer which, when proved at all, are typically proved by hand. The case study mechanizes these proofs using the actual data types and instances defined in the standard libraries. It also shows how, once scripted, the proofs can be automatically checked during subsequent compilation of the libraries, enforcing a correspondence with the code as it changes over time. It concludes by reflecting on the pragmatics of performing this kind of reasoning at compile-time. This case study, which was led by the author, was investigated jointly with Neil Sculthorpe, a Postdoctorate Fellow at the University of Kansas, and is currently under peer review.

- Chapter 7 develops a domain-specific optimization pass using HERMIT. Both the problem and approach are presented in detail, along with key simplification steps necessary to apply the transformation in practice. These simplifications were developed empirically, using HERMIT's interactive capabilities to investigate the optimization as it happened. Users of the optimization benefit by being able to express computation at a higher-level of abstraction, with greater safety, without loss of performance. This case study, which was led by the author, was investigated jointly with Christian Höner zu Siederdissen, Postdoctoral fellow at the University of Vienna, and published in Farmer et al. [2014].

Secondary Evidence

The case study in Chapter 8 and the projects in Chapter 9 reflect investigations where the author played a critical supporting role as HERMIT expert, and are offered as secondary evidence of the utility of HERMIT.

- Chapter 8 is a third significant HERMIT case study which mechanizes a calculational programming derivation taken from a textbook on the subject. The proofs presented in the textbook, along with many properties which were assumed, are mechanized with HERMIT. The resulting properties are used to transform a program to improve its performance. The study concludes by reflecting on HERMIT's success at matching the level of abstraction found in semi-formal derivations such as these. This chapter reflects joint work with Neil Sculthorpe, a Postdoctorate Fellow at the University of Kansas, who led an earlier, unpublished version of the case study. The study has since been significantly revised and extended by the author, with Neil's mentorship, and is currently under peer review.

- Chapter 9 gives a high-level summary of various other efforts which use HERMIT as a central enabling technology, reflecting on HERMIT's role and on the effect these efforts had on HERMIT's development.

Closing

- Chapter 10 provides research context about other systems for reasoning about programs in Haskell and more broadly.

- Chapter 11 concludes, and reflects on HERMIT's development. It also discusses potential future work both on improving HERMIT, and on applying it to reasoning tasks.

Chapter 2

Technical Background

In order that this dissertation be self-contained, this chapter presents background material relevant to the implementation and discussion of HERMIT. Knowledge of the Haskell language itself is assumed, and discussion focuses on the architecture of the Glasgow Haskell Compiler (GHC), GHC's plugin system, GHC's internal intermediate language (GHC Core), and the KURE strategic rewriting language. HERMIT is implemented as a GHC plugin which uses KURE to transform GHC Core.

Thoughout this dissertation, some GHC types will be replaced with more familiar, morally-equivalent types for clarity. For example, GHC pervasively uses its own string representation, which offers fast comparison and compact memory layout. As these details are not important for the discussion of HERMIT, Haskell's standard $String$ type is used instead.

2.1 GHC Plugins

GHC, like most compilers, is structured as a sequence of compiler phases. Broadly, these can be divided into the *front end*, which includes parsing, renaming, typechecking, and desugaring; the *optimizer*, which is a series of passes that transform an intermediate representation of the program; and the *back end*, which includes low-level optimization and code generation [Marlow and Peyton Jones, 2012].

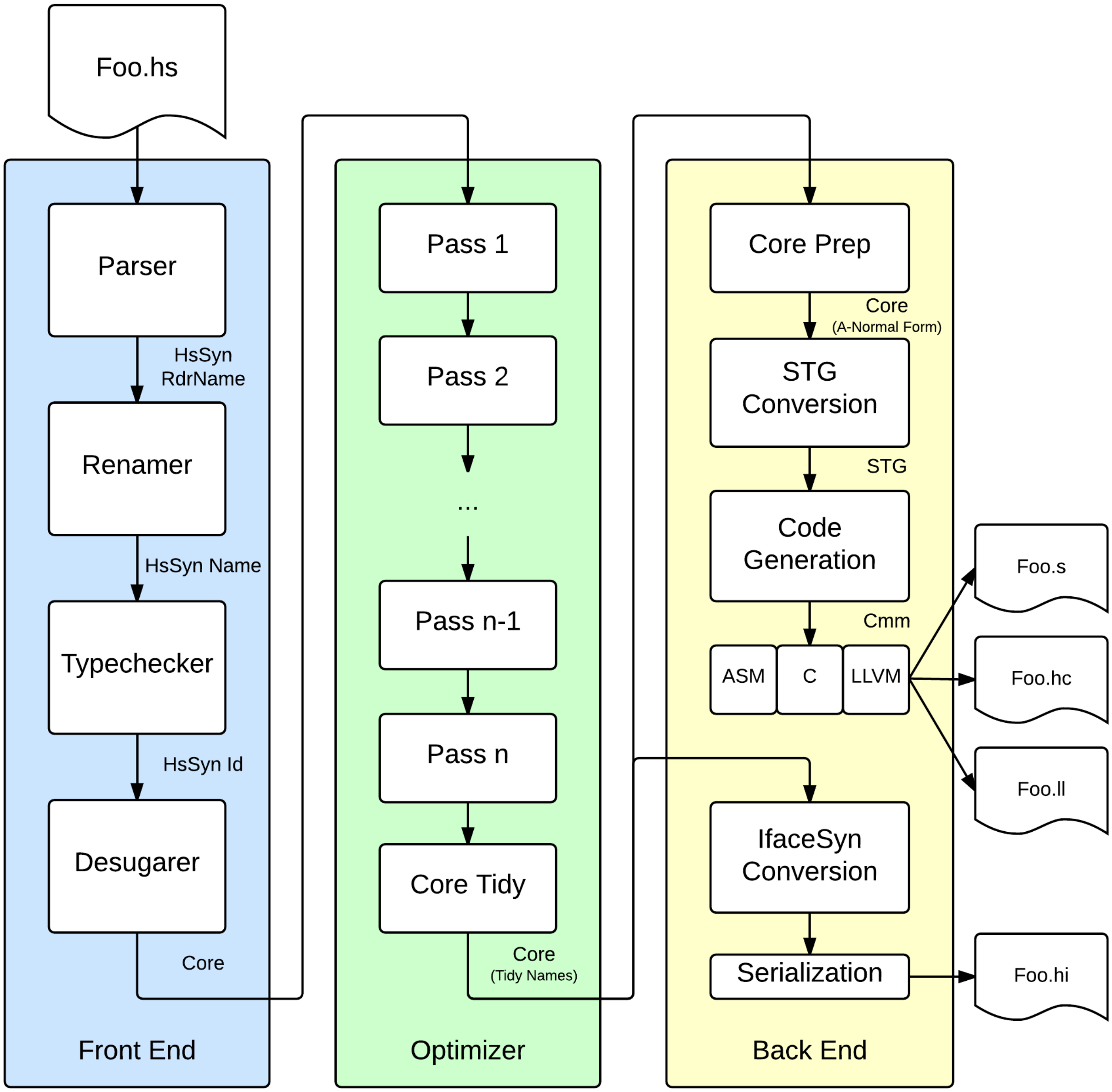


Figure 2.1: GHC Architecture

Figure 2.1 diagrams the major components of GHC. Arrows between components are annotated with the intermediate representation used at that point. The front end primarily uses the *HsSyn* type, which captures all of Haskell's source syntax. *HsSyn* is annotated with the type of named identifier used at that stage. Names are progressively refined by each stage in the pipeline, a process described in Section 2.2.1.

The output of the front end is an intermediate representation of the program in the GHC Core Language (Section 2.2). The optimizer is structured as a series of passes which accept a GHC Core program as input and produce a new GHC Core program. The program produced by the final optimizer pass is the input to the back end. While Figure 2.1 includes the back end for completeness, this disseration focuses exclusively on the optimizer and those parts of the front-end that are useful when reasoning about GHC Core Language programs.

GHC's plugin mechanism allows the programmer to modify the list of optimization passes, including inserting arbitrary passes between existing passes. A *plugin* itself is a function which takes a list of command-line flags and a list of passes and returns a new list of passes, which are then run by GHC. The $CoreM$ monad provides access to global optimizer state, a unique name supply, and IO; and collects statistics.

$\mathbf{type}\ Plugin = [CommandLineOption] \rightarrow [Pass] \rightarrow CoreM\ [Pass]$

A plugin is only run once, and can only determine which passes GHC runs, and in what order. It cannot, for instance, change the pass pipeline on a per-module basis. Once GHC begins running the passes, the pipeline is fixed.

Each *pass* is a monadic computation in GHC's $CoreM$ monad.

$\mathbf{type}\ Pass = ModGuts \rightarrow CoreM\ ModGuts$

A pass accepts and produces GHC's $ModGuts$ data type, which encapsulates all the details of the GHC Core program. For the purposes of this dissertation, $ModGuts$ can be thought of as a list of top-level binding groups along with relevent information about the typing environment such as type class instances [Wadler and Blott, 1989], type family instances [Chakravarty et al., 2005], and GHC rewrite rules [Peyton Jones et al., 2001] which are in scope.

2.2 GHC Core

GHC desugars source programs into a strongly-typed intermediate representation called GHC Core. GHC Core is an implementation of the System FC calculus [Sulzmann et al., 2007], which descends from Girard and Reynolds' System F [Girard et al., 1989]. System FC, and GHC's implementation via GHC Core, has evolved over time. Figure 2.2 presents the language of both terms and types in System FC as it currently implemented within GHC.

Names in GHC Core are unique identifiers for both term- and type-level entities. *Variables* are names which have been annotated with their type (or kind). *Identifiers* are term-level variables. *Expressions* are those typical to System F, including variables, literals, application, and abstraction. As in System F, expressions may be abstracted over types, formalizing the notion of parametric polymorphism. Accordingly, types may appear at the expression level as the arguments to polymorphic abstractions. Abstractions may never return types, however.

System FC extends System F at the expression level with let-binding, abstract data, and type casting. Types may be bound by non-recursive let-bindings, whereas values may be bound both recursively and non-recursively. Abstract data is created by applying *constructors*. This data can be deconstructed using a case expression, which binds the arguments of the matched constructor in each case alternative.

Casts wrap an expression, changing its type. A cast requires evidence, in the form of a *coercion* from the expression's actual type to the desired type. These coercions are constructed by the typechecker and manipulated by the optimizer. They support Haskell's zero-cost type abstraction [Breitner et al., 2014a] and Generalized Abstract Datatype [Schrijvers et al., 2009] features. Expressions may be abstracted over coercions. Thus, like types, they may appear at the term level.

The language of coercions are omitted from Figure 2.2 because the HERMIT user is generally not concerned with transforming coercions directly, relying instead on the correctness of HERMIT's rewrites for manipulating casts. It is sufficient to understand that coercions exist, and are GHC Core's means of passing around evidence of type equality.

Lastly, *Ticks* are used by GHC to annotate expressions with profiling and debugging information. Ticks are both created by source-level annotations and generated automatically by the front end based on compilation flags. While the optimizer has limited scope to move ticks, they generally pass through unmolested to the back end, where they affect code generation.

$$
\begin{array}{lll}
n & ::= String \times Int & \textit{Type- or term-level name} \\
v, \alpha & ::= n^{\tau} & \textit{Type- or term-level variables} \\
l & ::= \ldots\ machine\ literals\ \ldots & \textit{Literals} \\
\textit{Expressions} & & \\
e & ::= v & \textit{Variables} \\
 & \mid\ l & \textit{Literals} \\
 & \mid\ e_1\ e_2 & \textit{Application} \\
 & \mid\ \lambda\ v.\ \ e & \textit{Abstraction} \\
 & \mid\ \tau & \textit{Type} \\
 & \mid\ \textbf{let}\ b\ \textbf{in}\ e & \textit{Let Binding} \\
 & \mid\ \textbf{case}\ e\ \textbf{as}\ v\ \textbf{return}\ \tau\ \textbf{of}\ \overline{alt_i} & \textit{Pattern Matching} \\
 & \mid\ e\ \rhd\ \gamma & \textit{Cast} \\
 & \mid\ \gamma & \textit{Coercion} \\
 & \mid\ e_{tick} & \textit{Tick} \\
\textit{Bindings} & & \\
b & ::= v = e & \textit{Non-recursive} \\
 & \mid\ rec\ \overline{v_i = e_i} & \textit{Recursive} \\
\textit{Alternatives} & & \\
\overline{alt_i} & ::= \mathcal{K}\ \overline{vs} \qquad\quad \rightarrow e & \textit{Data} \\
 & \mid\ l \qquad\qquad\quad \rightarrow e & \textit{Literal} \\
 & \mid\ DEFAULT \rightarrow e & \textit{Default} \\
\textit{Types and Kinds} & & \\
\tau\ \kappa & ::= \alpha & \textit{Variables} \\
 & \mid\ \tau_1\ \tau_2 & \textit{Application} \\
 & \mid\ \mathcal{T}\ \overline{\tau_i} & \textit{Type constructor application} \\
 & \mid\ \tau_1 \rightarrow \tau_2 & \textit{Function} \\
 & \mid\ \forall\ \alpha.\ \ \tau & \textit{Polymorphism} \\
 & \mid\ L & \textit{Type Literals} \\
\textit{Type Literals} & & \\
L & ::= BIGN & \textit{Integers} \\
 & \mid\ String \times Int & \textit{Symbols}
\end{array}
$$

Figure 2.2: The GHC Core Language

At the type level, GHC Core features a flat type hierarchy. That is, types and kinds are encoded using the same data types. Thus, features at the type level, such as polymorphism, are available at the kind level. This supports Haskell's datatype promotion [Yorgey et al., 2012] and kind polymorphism [Yorgey et al., 2012] features.

2.2.1 Names

GHC offers several notions of 'named identifier' at various stages of compilation (Figure 2.1). While the optimizer, and thus HERMIT, primarily works with the $Name$ and Var types, it is important to understand the other types used in the front end in order to discuss HERMIT's notion of names (Section 4.3).

2.2.1.1 OccName

An $OccName$, or *occurrence name*, is a pair of the string portion of a name and a namespace.

$\mathbf{data}\ OccName = \mathsf{OccName}\ NameSpace\ String$

The $String$ is the unqualified, human-readable portion of the name. GHC currently enumerates four possible namespaces: value-level, type-level, data constructors, and type constructors. Class names are type constructors.

For instance, the unit type constructor () and its single data constructor () have the same string representation as two parentheses. However, the appropriate namespace is inferred by GHC's parser from the context of the name as it appears in the program (as part of a type or expression).

2.2.1.2 RdrName

The $RdrName$ type, or *reader name*, pairs an $OccName$ with information about where it is defined.

$$
\begin{aligned}
\mathbf{data}\ RdrName = {} & \mathsf{Unqual}\ OccName \\
\mid {} & \mathsf{Qual}\ ModuleName\ OccName \\
\mid {} & \mathsf{Orig}\ Module\ OccName \\
\mid {} & \mathsf{Exact}\ Name
\end{aligned}
$$

Unqualified names are just $OccName$s lifted by the Unqual constructor. They denote identifiers defined in the current module. Qualified names, using the Qual constructor, pair an $OccName$ with a module name. This may be the module that defines the identifier or another module which re-exports the identifier. The $ModuleName$ type may be thought of as a string such as “Data.List”.

The remaining two constructors to $RdrName$ are used internally by GHC. The Orig constructor is used to generate reader names which point to an identifier from a specific module in a specific package. The $Module$ type is a pair of $ModuleName$ and package identifier, meaning it is more specific than a $ModuleName$ alone. Whereas a Qual reader name will be resolved to a specific package using a complicated set of rules regarding package visibility, an Orig reader name bypasses this, specifying explicitly which package (and version) the module is found in.

The Exact constructor is used for built-in syntax, such as $[\,]$ and $(,)$, and for names generated by Template Haskell [Sheard and Peyton Jones, 2002]. In these cases, GHC already knows the more specific $Name$ (Section 2.2.1.3), but may need to return a $RdrName$.

2.2.1.3 Name

A $Name$ is a fully resolved, unique named identifier. It can be thought of as a triple of the $OccName$, a unique integer, and the provenance of the name, denoted by a $NameSort$ type. Eliding the details of $NameSort$, it denotes the specific module and package a name is defined in, whether it is visible externally, whether it was user- or compiler-generated, and whether it is wired-in to the compiler.

```
data Name = Name NameSort OccName Int
```

The important aspect of $Name$ is the integer, which serves as the globally unique identifier for the name within a single GHC session, and is used for fast comparison. Multiple distinct reader names may resolve to the same $Name$, but distinct $Name$s refer to distinct entities. The unique identifier associated with a given entity is cached, so if two $Name$s are created which in fact refer to the same entity, they receive the same unique identifier.

2.2.1.4 Var

A Var pairs a $Name$ with an associated type or kind. A value-level Var is also called an Id, and may contain additional metadata, such as arity and unfolding information. This metadata is stored using the $IdInfo$ type, and is attached directly to the Id in question so it is locally available.

The information in $IdInfo$ is extensive. Unfolding information and GHC RULES (Section 2.2.3) for the Id contain entire expressions representing the desired replacement expression. It is important to update this information appropriately during transformations. For instance, substitution must be done in the expressions appearing in $IdInfo$ fields.

While creating new local Ids with no $IdInfo$ is straightforward, the importance of proper $IdInfo$ means global Vars are looked up in a cache by $Name$. This cache is populated when GHC loads the interface file for an imported module. Exported Vars are serialized into the interface file in the back end.

2.2.2 Dictionaries

GHC Core supports Haskell's parametric polymorphism [Strachey, 1967] with explicit type application and abstraction. Similiarly, it supports *ad-hoc polymorphism*, embodied by the type class language feature [Wadler and Blott, 1989], with explicit application and abstraction over class *dictionaries* [Jones, 1995].

A dictionary is, conceptually, an n-ary tuple of functions implementing the n methods of a given class for a specific type. Class methods are, in turn, selector functions, selecting the implementation for a given method from a given dictionary. A single dictionary value exists for each instance of a given class. As the contents of the dictionary are entirely determined by the dictionary type, dictionaries are implicit in Haskell.

As an example, consider the $Functor$ type class and its instance for the $Maybe$ type.

```
class Functor f where
  fmap :: (a → b) → f a → f b
instance Functor Maybe where
  fmap :: (a → b) → Maybe a → Maybe b
  fmap _ Nothing = Nothing
  fmap g (Just x) = Just (g x)
```

In Haskell, the full type of the *fmap* class method is:

```
fmap :: ∀ f . Functor f ⇒ ∀ a b . (a → b) → f a → f b
```

This type signature makes clear that *fmap* is parametrically polymorphic in types a and b, and ad-hoc polymorphic in f, due to the *Functor* constraint.

The implicit arguments to *fmap* bound by the ∀s and to the left of the ⇒ become explicit arguments in GHC Core. As the method implementation for a specific instance is carried by the dictionary, *fmap* in GHC Core just projects that implementation out of the dictionary.

```
fmap :: ∀ f . Functor f → (∀ a b . (a → b) → f a → f b)
fmap = λf $dFunctor → case $dFunctor of
                          Functor g → g
```

Lastly, compare corresponding applications of *fmap* in Haskell and GHC Core.

```
digitToInt :: Char → Int
  -- Haskell
fmap digitToInt (Just '5')
  -- GHC Core
fmap Maybe $fFunctorMaybe Char Int digitToInt (Just Char '5')
```

In GHC Core, all the implicit arguments to the application of *fmap* have been provided by the typechecker. The first, third, and fourth arguments are *types*. The second argument is the concrete dictionary for the *Maybe* instance of *Functor*. The remaining arguments are values. The ordering of these arguments follows the order of their appearance in *fmap*'s type signature. In general, GHC prefixes the names of dictionary *binders* with $d and concrete dictionary *values* with $f. These prefixes have no special semantic significance, but provide a visual clue as to when a value or binder has a dictionary type.

2.2.3 RULES

A compiler, even a sophisticaled one like GHC, is limited in the reasoning it can perform automatically to ensure that a given program optimization is correct. The programmer, on the other hand, has a much deeper understanding of their program. For instance, the programmer may know that converting a finite map to a list of key-value pairs and back again will result in the original map.

$$\forall\, m.\ \ \mathit{fromList}\ (\mathit{toList}\ m)\ \equiv\ m$$

However, the optimizer cannot work this out itself, especially if the data structure used to implement the map is at all complicated, which it is likely to be for performance reasons. A programmer is unlikely to do such a redundant conversion on purpose, but such situations commonly arise as the result of aggressive inlining and simplification. Haskell's (and GHC Core's) purity means GHC can be very aggressive about inlining and simplification, since it does not have to worry about duplicating or reordering side-effects [Peyton Jones and Marlow, 2002].

Rather than maintain a large collection of ad-hoc rules about libraries like the one above, GHC implements a general mechanism allowing the programmer to specify such properties in their program using a RULES pragma. GHC then uses the properties as rewrites during simplification. For instance, the above property can be specified using the following RULES pragma:

{-# RULES “toList-fromList” $\textbf{forall}\ m.\ \ \mathit{fromList}\ (\mathit{toList}\ m) = m$ #-}

GHC will apply this property left-to-right whenever an expression matches the left-hand side. Rules are not allowed to have pre- or side-conditions. Pattern matching is first order, with ∀-quantified variables in the pattern matching the corresponding sub-expression in the target expression. Also, the head of each side of the rule is syntactically limited to applications of known, in-scope functions. GHC does not check that the rule is correct (because it cannot), thus an implicit proof obligation is left to the programmer. In fact, GHC doesn't even ensure that the set of rules in scope is consistent or terminating. Thus, rules are intended to be used primarily by expert programmers and library writers.

However, in practice, the use of rules is widespread in the Haskell community. Despite the restrictions on form, a large number of useful properties can be expressed as rules. Again, purity means a great many properties can be specified in Haskell syntax, without a meta-language for pre- or side-conditions.

2.3 KURE

The Kansas University Rewrite Engine (KURE) is a strategic programming language, implemented as a Haskell-embedded domain-specific language [Sculthorpe et al., 2014]. Strategic programming is concerned with *composing* transformations over tree-structured data using *strategies* [Visser, 2005]. KURE supports rewriting strongly-typed data using typed transformations and strategies.

HERMIT uses KURE as its primary means of specifying and applying transformations to a GHC Core program. KURE offers a principled means of performing typed generic traversals [Jeuring et al., 2009] of mutually recursive data types while maintaining a rewriting context and handling transformations with effects. It also comes with a substantial library of useful strategies for common traversal and error-handling patterns.

This section will present KURE from the point of view of a KURE *user*, as this is the knowledge needed to understand HERMIT's implementation. Readers interested in the details of KURE's design and implementation are encouraged to read Sculthorpe et al. [2014]. Throughout, examples will be presented for the following simple expression language:

$\mathbf{data}\ \mathit{Expr} = \mathsf{Var}\ \mathit{String} \mid \mathsf{App}\ \mathit{Expr}\ \mathit{Expr} \mid \mathsf{Let}\ \mathit{Bind}\ \mathit{Expr}$
$\mathbf{data}\ \mathit{Bind} = \mathsf{Bind}\ \mathit{String}\ \mathit{Expr}$

2.3.1 Transformations

The principle type in KURE is $Transform$, implemented as a function which accepts both a context c and an expression e, and produces a result r in monad m.

$\mathbf{newtype}\ Transform\ c\ m\ e\ r = Transform\ \{\mathit{apply} :: c \to e \to m\ r\}$

When the expression type e and the result type r are the same, a transformation is refered to as a *rewrite*.

$\mathbf{type}\ \mathit{Rewrite}\ c\ m\ e = Transform\ c\ m\ e\ e$

By convention, KURE transformations are suffixed with either **R** or **T** to indicate whether they are rewrites or transformations, respectively. This mnemonic assists in reading dense KURE code.

KURE provides a number of smart constructors for transformations, some of which will appear in code samples in this dissertation.

```
  -- Build a Transform from a function.
transform :: (c → e → m r) → Transform c m e r
  -- Build a Rewrite from a function.
rewrite :: (c → e → m e) → Rewrite c m e
  -- Build a Transform that does not depend on the context.
contextfreeT :: (e → m r) → Transform c m e r
```

Additionally, KURE offers several primitive transformations and strategies, including the identity rewrite *idR*, deterministic choice <+, and sequential composition ⋙.

```
idR   :: Rewrite c m e
(<+) :: Transform c m e r → Transform c m e r → Transform c m e r
(⋙) :: Transform c m a b → Transform c m b c → Transform c m a c
```

These primitives can be composed to produce more complex strategies. For instance, a strategy which turns a potentially failing rewrite into one which always succeeds can be implemented as:

```
tryR r = r <+ idR
```

That is, *tryR* attempts to apply the rewrite r. If r fails, then the identity rewrite is performed.

2.3.2 Monad

The monad [Peyton Jones and Wadler, 1993] instance for KURE transformations is a reader transformer [Liang et al., 1995] where the environment is both the context and the expression to be transformed. Note that the same expression and context are passed to each transformation by ≫=.

```
instance Monad m ⇒ Monad (Transform c m e) where
  return :: r → Transform c m e r
  return = contextfreeT return
  (≫=) :: Transform c m e r₁ → (r₁ → Transform c m e r₂) → Transform c m e r₂
  t ≫= k = transform $ λc e → do r₁ ← applyT t c e
                                 applyT (k r₁) c e
```

Thus, a series of transformations sequenced by ≫= all have access to the same context and input expression. To sequence a series of transformations in a pipeline, where each transformations operates on the result of the previous transformation, KURE's sequencing operator ⋙ is used.

2.3.3 MonadCatch

The most common effect required by transformations is failure, and catching failure. The ability to fail is included in Haskell's $Monad$ type class, but the ability to catch failure is not. Correspondingly, KURE introduces a subclass of $Monad$ for this purpose, called $MonadCatch$.

```
class Monad m ⇒ MonadCatch m where
  catchM :: m a → (String → m a) → m a
```

The $String$ argument allows failure to include a message to the handler. This *catchM* primitive is used to implement KURE's deterministic choice operator at the general $MonadCatch$ type.

```
(<+) :: MonadCatch m ⇒ m a → m a → m a
ma <+ mb = ma ‘catchM‘ const mb
```

An instance of $MonadCatch$ is provided for $Transform$, allowing <+ to be used on transformations as described in Section 2.3.1.

2.3.4 Traversal

The primitive *traversal* strategy in KURE is *allR*, which applies a rewrite to each child of the current node. *allR* is overloaded in both the context type and the expression type.

```
class Walker c u where
  allR :: MonadCatch m ⇒ Rewrite c m u → Rewrite c m u
```

As an example of defining a traversal in terms of *allR*, the following strategy, provided by KURE, applies a rewrite to every node in a top-down (pre-order) traversal.

```
alltdR r = r ⋙ allR (alltdR r)
```

Instances of $Walker$ are typically not defined directly on the data type being traversed. Doing so would limit *allR* to traversing a homogenous tree made up of a single type. In order to traverse heterogeneous data (trees featuring more than a single type), the $Walker$ instance is defined for a universe type. A *universe* is a disjoint sum of all the types which need to be traversed.

For example, in order to rewrite all expressions in the language of $Expr$ and $Bind$, the traversal must be able to visit both $Expr$ and $Bind$ nodes. This is because expressions appear as children to bindings. Thus, the universe for this task includes both types.

```
data EB = EBExpr Expr | EBBind Bind
```

Note that *String* is not included in the universe, so traversals do not descend into strings. The ability to control which nodes in a tree are traversed based on their type is called *static selectivity*, and is important for performance.

With a universe selected, two tasks remain. First, data must be lifted into the universe and projected back out of it. Doing so is required because *allR* (and thus, any traversal) will only apply to universe types. KURE provides a type class called *Injection* for this purpose.

```
class Injection a u where
  inject :: a → u
  project :: u → Maybe a
```

Note that projection from the universe can fail if the value is not of the desired type. While static typing guarantees that *allR* only visits nodes in the universe, and that the rewrite argument to *allR* can be applied to all nodes in the universe, it does not guarantee that the rewrite will not transform a node from one type to another type in the same universe. This guarantee is made by the dynamic type check implicitly provided by the *project* function. A rewrite over the universe type which does not preserve the node type will fail.

Instances of *Injection* are trivial to define:

```
instance Injection Expr EB where
  inject :: Expr → EB
  inject = EBExpr

  project :: EB → Maybe Expr
  project (EBExpr e) = Just e
  project _          = Nothing
```

Once such *Injection* instances are defined, the combinators in Figure 2.3, which are provided by KURE, can be used to lift and lower both values and transformations between target data types and their universe.

2.3.4.1 Context

The second task is to define a *Walker* instance for the universe. The *Walker* class is parameterized over the context type so that *allR* can update the context as it descends the tree. Properly updating

```
injectT :: (Monad m, Injection a u) ⇒ Transform c m a u
injectT = contextfreeT (return ∘ inject)

projectT :: (Monad m, Injection a u) ⇒ Transform c m u a
projectT = contextfreeT (λu → case project u of
                                   Nothing → fail "projectT failed"
                                   Just a   → return a)

promoteT :: (Monad m, Injection a u) ⇒ Transform c m a b → Transform c m u b
promoteT t = projectT ⋙ t

extractT :: (Monad m, Injection a u) ⇒ Transform c m u b → Transform c m a b
extractT t = injectT ⋙ t

promoteR :: (Monad m, Injection a u) ⇒ Rewrite c m a → Rewrite c m u
promoteR rr = projectT ⋙ rr ⋙ injectT

extractR :: (Monad m, Injection a u) ⇒ Rewrite c m u → Rewrite c m a
extractR rr = injectT ⋙ rr ⋙ projectT
```

Figure 2.3: Projection and Injection Transformations Provided By KURE

the context in *allR* means all other traversals, which are defined in terms of *allR*, will automatically update the context accordingly.

For this expression language, one might wish to maintain a collection of in-scope bindings in the context, making them locally available to any rewrite.

```
type Context = [Bind]
```

Thus, a *Walker* instance for *Context* and *EB* can be defined. Notice the case for Let in *allRExpr*, where the binding is added to the context while rewriting the let body.

```
instance Walker Context EB where
  allR :: MonadCatch m ⇒ Rewrite Context m EB → Rewrite Context m EB
  allR rr = rewrite (λc u → case u of
                              EBExpr e → EBExpr <$> allRExpr c e
                              EBBind b → EBBind <$> allRBind c b)
    where
      allRExpr :: MonadCatch m ⇒ Context → Expr → m Expr
      allRExpr _ (Var s)        = pure (Var s)
      allRExpr c (App e₁ e₂) = App <$> apply (extractR rr) c e₁ ⊛ apply (extractR rr) c e₂
      allRExpr c (Let b e)     = Let  <$> apply (extractR rr) c b  ⊛ apply (extractR rr) (b : c) e

      allRBind :: MonadCatch m ⇒ Context → Bind → m Bind
      allRBind c (Bind s e)  = Bind s <$> apply (extract rr) c e
```

Transformations can now rely on this contextual information. As one example, an inlining rewrite can be implemented which replaces variable occurrences with the corresponding expression.

```
inlineR :: MonadCatch m ⇒ Rewrite Context m Expr
inlineR = rewrite (λc e → case e of
                              Var s → case lookup s c of
                                          Nothing → fail "variable not in scope."
                                          Just e2  → return e2
                              _     → fail "not a variable.")
  where
    lookup :: String → [Bind] → Maybe Expr
    lookup _  []   = Nothing
    lookup s1 (Bind s2 e : bs)
      | s1 == s2  = Just e
      | otherwise = lookup s1 bs
```

This *inlineR* rewrite only handles the local case where the expression being rewritten is a single variable occurrence. This is by design! Rather than write a traversal which handles all the constructors of *Expr* by hand, *inlineR* can be lifted using one of KURE's traversal strategies. For instance, KURE's *anytdR* strategy can be used to lift *inlineR* into a rewrite which inlines all in-scope variables in a larger expression or binding.

```
inlineAnyR :: Rewrite Context m EB
inlineAnyR = anytdR (promoteR inlineR)
```

2.3.4.2 Congruence Combinators

Primitive transformations must also update the context. To reduce code duplication, it is recommended that the KURE user define a set of *congruence combinators* for their target types. Each congruence combinator can be thought of as a version of *allR* specialized to a particular constructor, accepting one transformation for each interesting child of the constructor. For example, the congruence combinators for the Let constructor are:

```
letT :: Transform Context m Bind a → Transform Context m Expr b → (a → b → d)
     → Transform Context m Expr d
letT t1 t2 f = transform (λc e → case e of
                                     Let b e → f <$> apply t1 c b ⊛ apply t2 (b : c) e
                                     _       → fail "not a let.")
letR :: Rewrite Context m Bind → Rewrite Context m Expr → Rewrite Context m Expr
letR r1 r2 = letT r1 r2 Let
```

```
instance Walker Context EB where
  allR :: MonadCatch m ⇒ Rewrite Context m EB → Rewrite Context m EB
  allR rr = rewrite (λc u → case u of
                              EBExpr e → EBExpr $ apply allRExpr c e
                              EBBind b → EBBind $ apply allRBind c b)
    where
      allRExpr :: MonadCatch m ⇒ Rewrite Context m Expr
      allRExpr =    appR (extractR rr) (extractR rr)
                 <+ letR (extractR rr) (extractR rr)
                 <+ varR

      allRBind :: MonadCatch m ⇒ Rewrite Context m Bind
      allRBind = bindR (extractR rr)
```

Figure 2.4: An Instance of *Walker* Defined using Congruence Combinators.

Similiar combinators can be defined for Var, App, and Bind. Note that *letT* updates the context appropriately. Also note that congruence combinators are defined on the target type directly, not on the universe type. This is because they are local, only rewriting the direct children of the current node.

Figure 2.4 redefines the instance of *Walker* for *EB* using the congruence combinators, making them the sole place where context updates must be explicitly made. This is the way KURE recommends implementing *allR*.

2.3.5 Summary

To summarize, the KURE user must select a universe of types to be traversed, create a universe sum type, create *Injection* instances, define a desired context type, and write an instance of *Walker* for the given context and universe. It is recommended that the user also implement a set of congruence combinators, one for each constructor, because they are useful to ensure the context is always updated appropriately.

More than one universe may be used for a given set of types. Each universe may have different traversal behavior, determined both by the types in the universe and the implementation of *allR*. The user is free to write an instance of *Walker* that selectively traverses based on *position* instead of type, only descending into the left-hand side of applications, for instance. Additionally, it is possible to use more than one type of context. Having conguence combinators reduces the burden of implementing multiple $Walker$ instances.

Chapter 3

HERMIT Architecture

This chapter overviews the architecture of the HERMIT system. It describes HERMIT's place in the GHC optimization pipeline and the three levels of API provided by HERMIT: the kernel, which is a low-level client-server interface; the plugin DSL, a monadic domain-specific language for sequential transformation; and the Shell, a read-eval-print loop for interactive transformation and proof.

Figure 3.1 summarizes the major components comprising HERMIT. The arrows in the figure indicate a *depends on* relationship. The first major component is the core of HERMIT. The HERMIT core provides the interfaces and types for using KURE on the GHC Core Language. It also implements the Kernel, which acts as the main transformation loop for a single HERMIT pass; Folds, a means to use one GHC Core expression as a pattern for matching another expression; a unified interface for names; Lemmas, which are the principal method of performing equational reasoning in HERMIT; and a common API which unites various GHC functionality. The Kernel is discussed in this chapter, with the remainder of the HERMIT core featuring in Chapters 4 and 5.

The second major component is the HERMIT Dictionary, a collection of commonly used transformations on GHC Core. These include fundamental rewrites for folding, unfolding, and other local transformations, as well as higher-level rewrites for reasoning, accessing GHC functionality, debugging, navigating the AST, and constructing expressions. In essence, any transformation

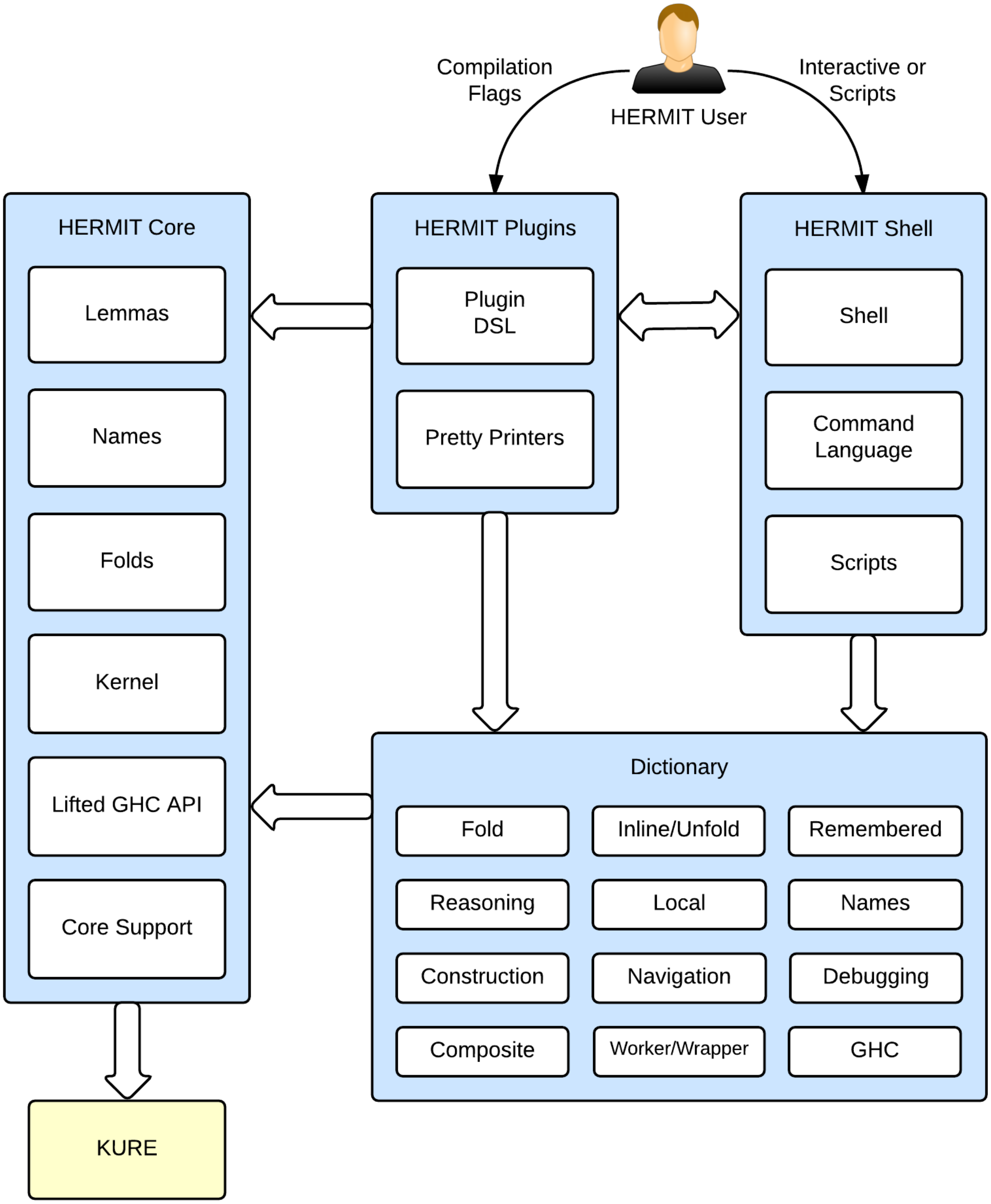


Figure 3.1: HERMIT Architecture

provided directly to the user is defined in the dictionary, and it represents the majority of the user-facing API for HERMIT. Capabilities of the Dictionary are highlighted in Sections 4.5 and 5.10.

The final two major components are the Plugin API and HERMIT Shell. The Plugin API implements the functionality for scheduling HERMIT passes in the GHC optimization pipeline, and is discussed in Section 3.1. It implements a small DSL for structuring HERMIT transformations which involve multiple passes. This Plugin DSL (Section 3.3) is more powerful than the Shell alone, and is intended primarily for creating automated HERMIT passes. The Shell (Section 3.4) features a language that is less expressive, but safer, and is meant for interactive transformation and proof. The two components are interdependent: the Shell can be invoked by a HERMIT plugin and is itself built using the Plugin DSL.

3.1 Plugin

Recall from Section 2.1 that a GHC plugin is allowed to modify the collection of optimization passes normally run by the optimizer, including inserting new passes. However, once the new list of passes is handed back to GHC, there is no way to further modify the pipeline. Additionally, when compiling multiple modules, there is no way to specify different pipelines on a per-module basis, the same pipeline of passes is applied to every module.

Typical HERMIT use cases can require more flexibility. The programmer may wish to target each module at different points in the pipeline, or skip some modules altogether, allowing GHC to compile them as normal. The particular passes which HERMIT should target may not be statically known. It is also useful to be able to see the results of GHC's own passes, especially when constructing a domain-specific optimization.

To provide this extra flexibility, HERMIT inserts its own pass at *every* point in the pipeline (Figure 3.2). Each pass, when invoked by GHC, may decide whether to act or immediately return the program unmodified. This decision can be made based on the module being compiled, command-line flags, or the pass's position in the pipeline.

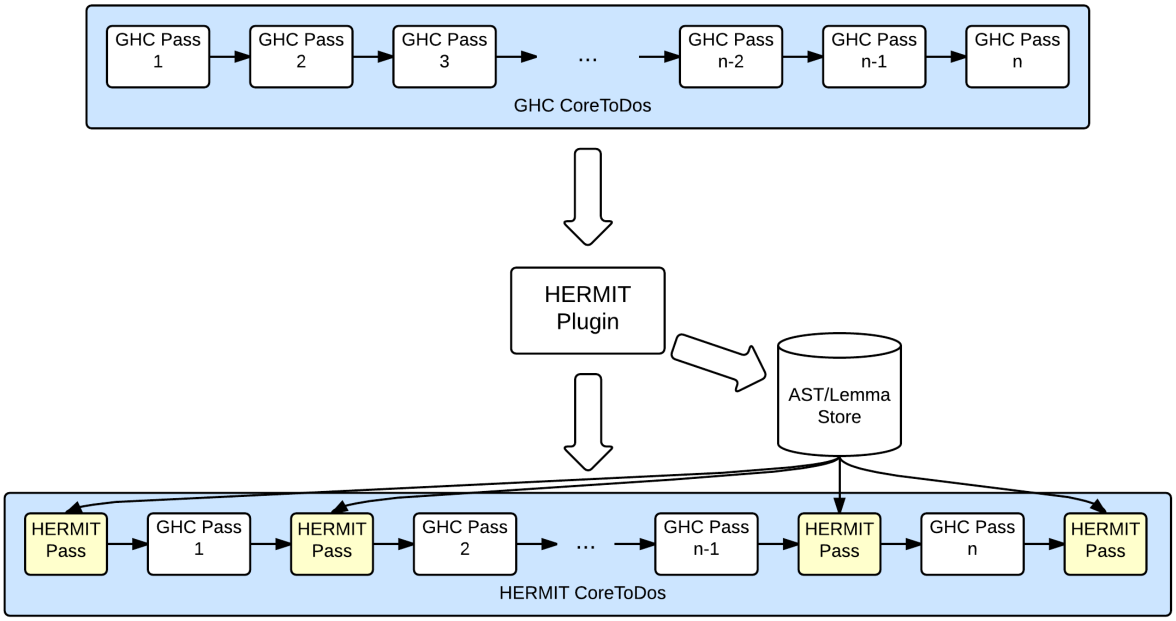


Figure 3.2: HERMIT Plugin - Installed Core-to-Core Passes

In order to maintain a history of transformations across passes, including GHC passes, the plugin creates a mutable, global AST store. This store can be empty if no transformation has yet occurred. When populated, it is tagged with a module name, ensuring the transformation histories of previously compiled modules are discarded after they are no longer useful. (GHC only actively compiles one module at a time.) The store is mutable, because the design of the GHC plugin system otherwise prevents HERMIT passes from communicating their changes to the store.

$\textbf{type}\ \mathit{GlobalStore} = \mathit{IORef}\ (\mathit{Maybe}\ (\mathit{ModuleName}, \mathit{ModuleStore}))$

When a HERMIT pass opts to transform a module, it will first select the $\mathit{ModuleStore}$ for that module. If no store exists, or the module store present belongs to another module, a new one is created and inserted into the global store with the name of the current module being transformed.

$\textbf{type}\ \mathit{ModuleStore} = \mathit{IORef}\ (\mathit{Maybe}\ (\mathit{ASTId}, \mathit{ASTMap}))$

The module store itself is mutable, containing a unique identifier for the GHC Core program returned by the previous HERMIT pass (ASTId) and a complete transformation history of that program ($ASTMap$). If this is the first HERMIT pass in the pipeline, the store will be empty, indicated by a Nothing value.

Recall that a single HERMIT pass is a $ModGuts \to CoreM\ ModGuts$ function. The first thing each pass does is add the input $ModGuts$ to the store, attributing it to the GHC pass which just completed (if any). This effectively versions GHC passes as if they were themselves monolithic transformations. The last thing the pass does is update the store with the $ASTId$ of its result, and the updated transformation history.

This design presents a number of capabilities which are useful in practice, including access to definitions and program properties from prior passes. As an example, the user may load a GHC rewrite rule, prove it valid once in the first pass, then apply it throughout the rest of the pipeline, without reloading or re-proving it.

3.2 Kernel

The kernel is HERMIT's lowest level API. It acts as a store for snapshots of each version of the GHC Core program being transformed, as well as arbitrating access to GHC's shared state among (potentially) multiple clients.

The HERMIT kernel is implemented using a client/server model. A single *server* loop, operating within the thread of the GHC optimizer, services requests from multiple clients. This server is started by the $hermitKernel$ function, which accepts a callback implementing the client.

$$hermitKernel :: (Kernel \to ASTId \to IO\ ()) \to ModGuts \to CoreM\ ModGuts$$

The server invokes this callback once, creating a separate thread and passing it a $Kernel$ object and a unique identifier representing the initial abstract syntax tree of the GHC Core program which is being transformed. Figure 3.3 diagrams the request/reponse cycle of the client and server threads.

The $Kernel$ object is a set of IO functions for making requests to the server, and is itself stateless. Thus, it may be duplicated as necessary, allowing the single client callback to spawn an arbitrary number of client threads. Figure 3.4 contains the full kernel client API.

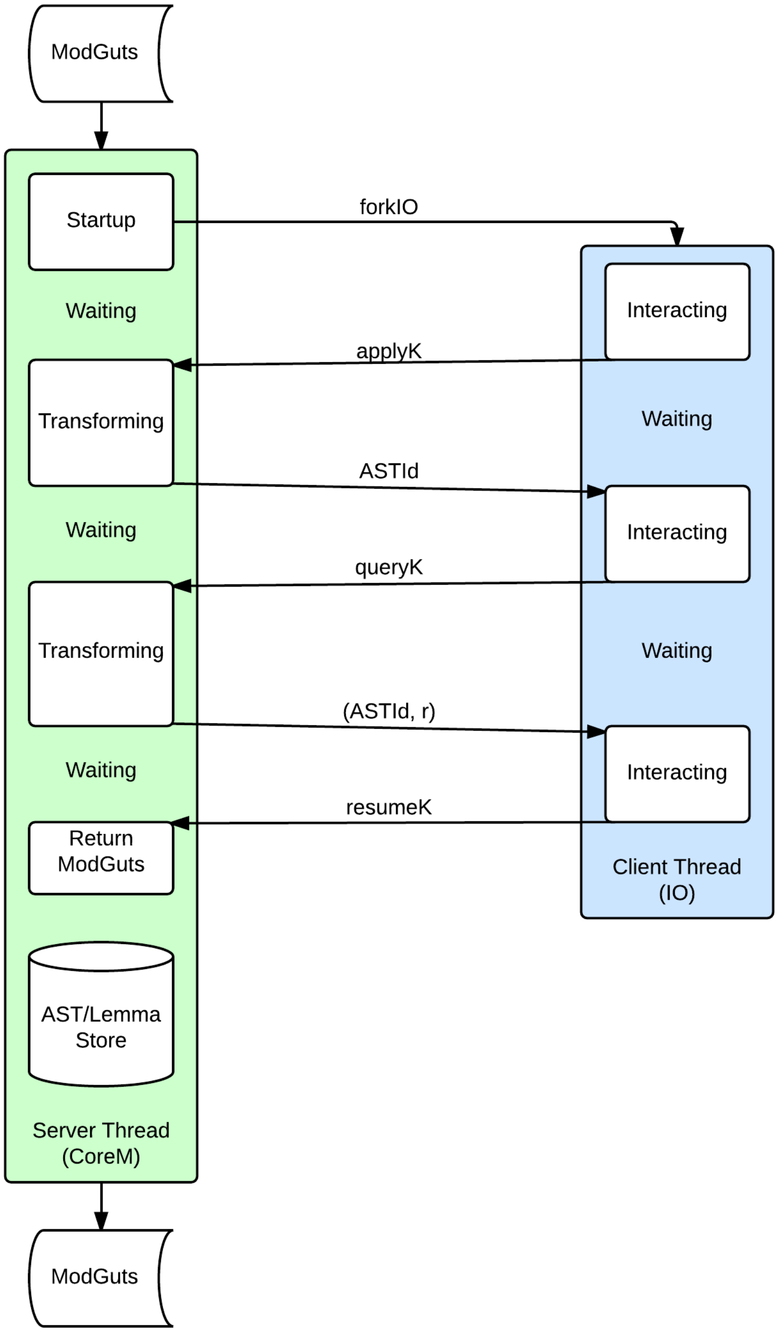


Figure 3.3: Kernel Request/Response Cycle

Once partially applied to the callback function for the client, the $hermitKernel$ function has the type $ModGuts \rightarrow CoreM\ ModGuts$. Recall (from Section 2.1) that this is the type of a GHC optimization pass. Note that the client callback is an IO computation, not a $CoreM$ computation.

```
data Kernel = Kernel
  {resumeK :: MonadIO m ⇒ ASTId → m ()
  , abortK  :: MonadIO m ⇒          m ()
  , applyK  :: (MonadIO m, MonadCatch m)
             ⇒ RewriteH ModGuts       → CommitMsg → KernelEnv → ASTId
             → m ASTId
  , queryK  :: (MonadIO m, MonadCatch m)
             ⇒ TransformH ModGuts a → CommitMsg → KernelEnv → ASTId
             → m (ASTId, a)
  , deleteK :: MonadIO m ⇒ ASTId → m ()
  , listK   :: MonadIO m ⇒ m [(ASTId, Maybe String, Maybe ASTId)]
  }
```

Figure 3.4: Kernel API

The client is run in its own thread, which does not have access to GHC's monadic state. Thus, all transformations must happen in the server thread.

This is not an arbitrary restriction of HERMIT's design, but rather intentional. GHC's optimizer relies on shared, often global, state for performance reasons, and is not designed to be multi-threaded. A critical function of the HERMIT kernel is to multiplex requests from multiple client threads into this single stateful GHC thread for transformation.

The other key function of the kernel server is to maintain a collection of *KernelState* objects which are identified by unique keys. A client may ask to list the objects, delete an object, rewrite an object, or query an object. Each *KernelState* object is a pair of GHC Core abstract syntax tree (GHC's own *ModGuts* type) and collection of HERMIT lemmas (Section 5.2).

```
data KernelState = KernelState ModGuts (Map LemmaName Lemma)
```

If a rewrite or query request modifies a *KernelState* object, it is saved with a new unique key. Since *KernelState* objects may be added or deleted, but never modified, requests are idempotent.

3.3 Plugin DSL

The Kernel arbitrates access to GHC's internal state and is used to define a single pass in the pipeline. The Plugin DSL builds on the Kernel to allow the specification of an entire plugin, determining the transformations to be applied and at what stage in the pipeline to do so. It was developed with domain-specific optimizations in mind, but it is also the primary means of extending HERMIT (Section 3.4.3).

Domain-specific optimizations are typically automatic transformation passes. They consist of applying one or more transformations in sequence, modifying the AST statefully. A transformation may need to run in a specific optimization pass, or in multiple passes. HERMIT's Plugin DSL is a small, monadic language for expressing a linear sequence of transformations in one or more stages of the pipeline.

The $PluginM$ monad encapsulates the effects in the DSL. These include failure, and an appropriate $MonadCatch$ instance (Section 2.3.3); a reader for access to the Kernel and information about the current pass's position in the pipeline; state for the current $ASTId$, as well as pretty-printer options; and IO. A $\mathit{hermitPlugin}$ function is provided to lift a $PluginM$ computation into a GHC Plugin.

$$\mathit{hermitPlugin} :: ([\mathit{CommandLineOption}] \to \mathit{PluginM}\ ()) \to \mathit{Plugin}$$

Note that this is at a higher level than the $\mathit{hermitKernel}$ function defined in Section 3.2. The $PluginM$ computation provided to $\mathit{hermitPlugin}$ is run for *every* HERMIT pass in the pipeline, whereas $\mathit{hermitKernel}$ is only used to define a single pass.

$PluginM$ computations are constructed from a set of primitives which parallel those of the Kernel interface in Figure 3.4, except that the $ASTId$ and $KernelEnv$ parameters are supplied implicitly by the monadic state. These combinators are summarized in Figure 3.5.

The information summarizing the current pass in $PluginM$'s reader environment allows the computation to behave differently at different points in the pipeline. *Temporal guards* selectively enable or disable a sub-computation by applying a predicate to this pass information. Unguarded computations are run in every HERMIT pass (before and after every GHC pass). A selection of commonly used temporal guards is listed in Figure 3.6. In the figure, the $PassInfo$ type embodies information about the current pass, including numeric position in the pipeline, a list of passes

```
abort  :: PluginM a
resume :: PluginM a
apply  :: (Injection ModGuts g, Walker HermitC g)
       ⇒ CommitMsg → RewriteH g       → PluginM ()
query  :: (Injection ModGuts g, Walker HermitC g)
       ⇒ CommitMsg → TransformH g a → PluginM a
list   :: PluginM [(ASTId, Maybe String, Maybe ASTId)]
delete :: ASTId → PluginM ()
```

Figure 3.5: Plugin DSL - Transformation Functions

```
guard     :: (PassInfo → Bool) → PluginM () → PluginM ()
pass      :: Int               → PluginM () → PluginM ()
after     :: CorePass          → PluginM () → PluginM ()
before    :: CorePass          → PluginM () → PluginM ()
until     :: CorePass          → PluginM () → PluginM ()
firstPass ::                     PluginM () → PluginM ()
lastPass  ::                     PluginM () → PluginM ()
```

Figure 3.6: Plugin DSL - Temporal Guards

already executed, and a list of passes that remain. The $CorePass$ type is an enumeration type for each type of GHC pass.

Finally, there are several utility combinators for displaying the current Core program and changing the behavior of the pretty-printer. This is primarily to allow plugins utilize the debugging combinators in Section 4.5.6. For instance, a plugin can dump an expression before and after transformation, in a manner similar to GHC's own debugging dumps, but with control over the level of detail of the pretty-printed information.

3.3.1 Example Plugin

As a small example of the Plugin DSL in action, the following HERMIT Plugin observes the effect of GHC's Worker/Wrapper pass, which attempts to unbox function arguments based on strictness information. Before and after the Worker/Wrapper pass, it pretty-prints the definition of each function in the list, which is supplied via command-line flags.

To print each function, it finds a path to the definition of that function using the *bindingOfT* transformation (Section 4.5.5), then pretty-prints the GHC Core expression at that location using the *observeT* debugging transformation (Section 4.5.6). As the applied transformation does not alter the Core program, there will be no change to the history maintained by the Kernel, so no commit message is supplied.

```
module DumpWorkerWrapper where
import ...
plugin :: Plugin
plugin = hermitPlugin $ λopts → do
  let printAll = query NoMessage $
                   forM_ opts $ λnm → do
                     path ← bindingOfT (cmpHN2Var (fromString nm))
                     localPathT path (observeT nm)
  before WorkerWrapper printAll
  after  WorkerWrapper printAll
```

3.3.2 Pretty Printer

GHC implements a pretty-printer for the GHC Core language. This pretty-printer is used by various debugging flags to dump the GHC Core program, or parts thereof, at different stages of optimization, for manual inspection. This is primarily useful to performance-conscious programmers trying to determine the effect of source annotations such as RULES, INLINE, and SPECIALIZE pragmas citepSPJ:02:Inliner on optimization. It is also useful to GHC developers while developing new optimization passes or diagnosing poor or unexpected optimization behavior.

With these use-cases in mind, the GHC Core pretty-printer is designed to display a lot of detailed information in a reasonably compact format. Occurrence names are printed with their unique identifiers, and identifiers from other modules tend to be printed with their fully-qualified names. Type and dictionary applications are explicit, and the former is indicated with an explicit @ symbol. Binders are annotated with types, and the structure of coercions is presented in detail. Applications, even of infix operators, are in prefix notation. In general, it displays all available information, and since this information is normally viewed in a terminal, or written to a file, no syntax highlighting is used. As an example, the output for the expression $(f * t) + e <= c$ is:

```
(GHC.Classes.<=
  @ GHC.Types.Int
  GHC.Classes.$fOrdInt
  (GHC.Num.+
     @ GHC.Types.Int
     GHC.Num.$fNumInt
     (GHC.Num.* @ GHC.Types.Int GHC.Num.$fNumInt f t)
     e)
  c)
```

This pretty-printer is not well suited for interactive use because the output tends to be dense. The density stems from the fact that it is inteded for post-mortem inspection. For interactive use, the default pretty-printer should produce output which obviously corresponds to the source program as much as possible. If more information is desired, settings can be modified and the expression re-displayed. Also, since the actual data structures representing the expression are available, information can be extracted in other ways besides pretty-printing.

To this end, HERMIT defines its own pretty printer which attempts to be more compact by default. The expression above, rendered by HERMIT's default settings, looks like:

```
(<=) ▲ $fOrdInt ((+) ▲ $fNumInt ((*) ▲ $fNumInt f t) e) c
```

The green triangles are type expressions. Types are always displayed in green, to help distinguish them from terms. They appear as this abstract symbol by default so that the overall expression more closely resembles source-level Haskell, where type application is implicit. If the user decides they wish to see the types explicitly, HERMIT can be instructed to do so.

```
(<=) Int $fOrdInt ((+) Int $fNumInt ((*) Int $fNumInt f t) e) c
```

Types and dictionaries can also be hidden altogether.

```
(<=) ((+) ((*) f t) e) c
```

This results in an expression which corresponds most closely with the Haskell expression. This is useful at times for viewing large expressions, but hides details which are important for navigation. For instance, in this view it is not obvious how many arguments the $+$ function is applied to, making navigation to a particular argument difficult. For this reason, HERMIT defaults to the abstract view.

HERMIT's pretty printer has several other optional views which can be enabled or disabled. These include the ability to see the unique identifier for each variable (as in the GHC pretty-printer); the ability to view the structure of coercions, or only the type of the coercion, or hide coercions and casts altogether; and the ability to view the difference between two versions of the program.

The rendering of the pretty-printer is also overloaded, allowing the output to be rendered in ASCII (with no color or unicode symbols), unicode (the default), latex, JSON, or HTML. The latter two have been used to build prototypes of rich browser-based clients for HERMIT, as they are formats that are well understood by browsers.

HERMIT also permits the use of the GHC pretty-printer if desired. An additional third pretty-printer displays the raw constructors of the datatypes which make up a GHC Core expression, and is primarily used for debugging HERMIT itself.

3.4 Shell

An important HERMIT feature is the ability to explore a transformation or proof *interactively*. The HERMIT user is free to experiment on their program, gaining intuition for the desired transformation or proof. Interactive use occurs in a read-eval-print loop (REPL) style common to other interactive theorem provers. The HERMIT REPL is called the HERMIT *Shell* because it is an interface suited for use in a command-line terminal.

The implementation of the Shell is largely a matter of engineering, and not particularly relevant to the discussion of HERMIT as a system. However, the capabilities offered by the Shell represent a significant aspect of HERMIT's user interface. With this in mind, the discussion in this section is focused on *what* the Shell can do, and not necessarily *how* it is done.

3.4.1 Interpreted Command Language

The primary function of the Shell is to allow the user to easily apply transformations to a specific part of the GHC Core program. To that end, the Shell maintains a *focus* into the program which may be changed by navigation commands, similar to a cursor in a word processor. Each command is applied to the focused part of the program. When a command alters the focused part of the program, it is redisplayed. When proving a property, that property is the focus of command application, rather than the underlying program.

Commands are entered into the Shell in an applicative style using a small interpreted command language. Expressions in the language are monomorphic, and the language itself is both strongly and dynamically typed, with built-in support for ad-hoc polymorphism [Strachey, 1967].

The command language itself is made up of statements and expressions. Statements are delimited by semicolons or carriage returns. A sequence of statements is called a script. The parser for this language is extremely liberal regarding the placement of statement separators in order that scripts may use whitespace for organization, or themselves be generated.

Expressions may be fully-applied commands or infix operators, identifiers in the target program (prefixed with an apostrophe), strings or names (such as GHC rewrite rule names), or a list of expressions of the same type. Strings and names may optionally be enclosed in double quotes if they contain whitespace.

A single statement is comprised of a top-level expression. A top-level expression is a fully applied call to a command in the HERMIT Dictionary. The Dictionary is a list of dynamically typed values which wrap monomorphic Haskell functions or values that implement the command. Each of these values is principally a pairing of command name string and a *Dynamic* [Peterson] value, plus additional metadata used for HERMIT's internal command documentation.

```
type Dictionary = [External]
data External = E String Dynamic ... other metadata ...
```

An interpreter uses this dictionary to associate command names with a list of dynamically-typed values. If a single command name is associated with multiple values, then all of the values are possible interpretations of the command. The ambiguity is resolved during application, where the number and types of the arguments determine which command was intended. If this ambiguity cannot be resolved, or no command exists with the desired type, the expression is ill-typed. Thus, commands in the language are ad-hoc polymorphic, or overloaded.

For example, given the expression `foo bar`, both `foo` and `bar` are looked up in the dictionary, each returning one or more dynamic values. The values for `foo` are applied to the values for `bar` in a cross product fashion. For any pair, if the application is ill-typed, the result of the application, itself a dynamic value, is discarded. The result of the application is a list of one or more dynamically-typed partial applications which can be applied to the next argument. If no arguments remain, the interpreter attempts to cast each dynamic value in the list of results to one of a set of known command types. The first such cast to succeed is the action taken by the Shell.

By convention, command names in the Shell are taken from the underlying KURE transformation which implements the command. To distinguish the two during discussion, KURE transformations are named in the camelCase style typical to Haskell, whereas the equivalent Shell command is hyphen-delimited, in the style of Lisp. For example, the *caseFloatArgR* transformation is accessed in the Shell using the command name `case-float-arg`.

This language is both simple and easily extensible, a capability explored in Section 3.4.3. However, its lack of abstraction and parametric polymorphism, in particular, have become pain points as the Dictionary has grown and HERMIT has been applied to ever larger examples. As future work, it will likely be replaced by a full Haskell interpreter, in a manner similiar to GHC's own GHCi. This would allow the KURE transformations underlying the current commands to be used directly, along with the full power of the Haskell language.

This replacement has not been made to date because running an in-process GHCi is problematic for several technical reasons. In large part this is due to GHC's use of global state internally, as state relating to the target program would become mixed up with state relating to interpreting HERMIT commands. A significant amount of refactoring of GHC itself, considered outside the scope of this dissertation, would be involved in enabling this.

In any case, the deficiency of the Shell language relative to a full Haskell interpreter is not a fundamental limitation of HERMIT itself. It has proven sufficient to accomplish a wide array of proof and transformation tasks.

3.4.2 Scripts

Once a transformation has been performed interactively, it is important that the sequence of commands can be saved and replayed on subsequent compilations. This replay ability is key to enforcing a correspondence between the transformation script and the underlying program as it changes over time. In the case of proving properties, the ability to replay a sequence of commands implements proof checking.

A HERMIT script is a sequence of Shell command statements. HERMIT provides commands for saving, loading, and running scripts, as well as turning a subset of scripts into KURE transformations.

To save a script, the Shell follows the command history maintained by the Kernel from the current version of the program to the initial version, recording the list of commands. The script may be written out to a file, optionally including the focused expression as a comment between each command to aid script legibility. By convention, script files are saved with the extension `hec`, for 'HERMIT Command'.

To load a script, the Shell reads a specified script file and parses the contents as statements. The distinction between loading and running is that loading does not interpret the statements, merely storing them under an associated script name. Once loaded, a script may be run, causing it to be interpreted in the current context of the Shell session. This means scripts can be used as a primitive means of abstraction for Shell commands, with identifiers in the script dynamically scoped.

A subset of scripts can be used to generate transformations that are added to the Dictionary. This subset includes scripts that consist solely of a sequence of rewrites on the GHC Core expression in focus. Commands which extract information from the expression, or otherwise change

Shell behavior cannot be lifted in this way. This is useful for higher-order commands, as the argument command can be constructed from a script.

3.4.3 Extending HERMIT

The Shell itself is implemented as a computation in the *PluginM* monad of the Plugin DSL. It accepts a Dictionary of commands and a list of command line options.

```
interactive :: Dictionary → [CommandLineOption] → PluginM ()
```

Command line options are interpreted as the filenames of scripts which should be run automatically. The plugin that comes with HERMIT is defined thus:

```
module HERMIT where
import HERMIT.Dictionary (dictionary)
import HERMIT.Plugin
plugin :: Plugin
plugin = hermitPlugin (firstPass ∘ interactive dictionary)
```

The Shell may be used by user-defined plugins as well. For instance, a user may specify a custom plugin which attempts a transformation and falls back on the Shell when an error is encountered.

```
plugin :: Plugin
plugin = hermitPlugin $ λopts → firstPass $ do
  apply NoMessage fooRewriteR ‘catchM‘ (λ_errMsg → interactive dictionary opts)
```

To extend HERMIT, commands can be added to the default Dictionary included with HERMIT before invoking *interactive*.

3.4.4 Proving in the Shell

To support proof, the Shell maintains a stack of proof obligations. This stack is initially empty, but obligations may be added explicitly by the user, or implicitly by transformations (Section 5.7). Whenever the stack is non-empty, the Shell considers the property on top of the stack to be the focus of commands. Additionally, the Dictionary is extended to include commands which only apply during proof, such as the ability to end a proof. When a proof is successfully completed, the obligation is popped from the stack.

3.5 Invoking HERMIT

HERMIT installs an eponymous executable which is the primary means of starting the HERMIT system. The only required argument is the name of the Haskell source file.

```
$ hermit Foo.hs
```

This executable is a simple option parser, turning a set of HERMIT flags into a set of GHC flags, and is provided for convenience. An invocation of HERMIT is actually just an invocation of GHC with the flags generated by this option parser. The above invocation of `hermit` becomes:

```
$ ghc Foo.hs -fplugin=HERMIT -fplugin-opt=HERMIT:*:
```

This invokes GHC on `Foo.hs` with a modified optimization pipeline which runs the Shell for all modules targeted for compilation, in the very first pass. Thus, the user is operating on the result of desugaring, before any other GHC passes are run. This is the default because the GHC Core program at this point is most similiar to the original Haskell source, making this a useful point to prove properties and perform calculational programming. The Shell's position in the pipeline may be altered with a flag.

Recall that the same pipeline is run on every module targeted for compilation, not just the one contained by `Foo.hs`. In general, GHC compiles any modules included by the target file which are in the same package and have changed since they were last compiled. A module-target flag may be added to instruct HERMIT to target a specific module, including modules defined in files besides `Foo.hs`.

```
$ hermit Foo.hs +Bar
```

This will invoke GHC as before, with the Shell as the first pass in the pipeline, but only for the module named *Bar*. Any options or flags after the module-target flag are considered per-module flags, up to the subsequent module-target flag. These per-module flags are the ones passed to the callback function by the *hermitPlugin* function in Section 3.3.

The default HERMIT plugin invokes the Shell at the beginning of the pipeline. To invoke a custom HERMIT plugin defined using the Plugin DSL, a `-plugin` flag may be specified after the file name, but before any module-target flags. For example, the *DumpWorkerWrapper* plugin defined in Section 3.3.1 can be invoked on module *Bar* while compiling file `Foo.hs`:

```
$ hermit Foo.hs -plugin=DumpWorkerWrapper +Bar
```

Note that, due to a technical restriction in GHC, the *DumpWorkerWrapper* module must be installed in GHC's package database (using `cabal install`) before it may be used. Thus, custom plugins are typically defined in separate packages from the code which they target.

Chapter 4

Transformation

At its core, HERMIT is a system for transforming GHC Core programs. HERMIT provides a large set of standard transformations for this task, as well as the necessary supporting infrastructure for a HERMIT user to define their own transformations. This chapter presents this infrastructure, detailing key design decisions and capabilities.

It begins by walking through an example program transformation to give a feel for interactive transformation in HERMIT (Section 4.1). It then discusses HERMIT's chosen language for specifying transformations, KURE. This includes motivation of the capabilities needed in a transformation DSL, why KURE was selected, and the infrastructure HERMIT provides for applying KURE to GHC Core (Section 4.2). Next, it presents HERMIT's notion of named identifier, which differs from GHC's various name types (Section 4.3). Subsequently, HERMIT's primitive expression folding capability is described in detail (Section 4.4), as it is key to applying transformations which are only known at HERMIT runtime, such as GHC rewrite rules (Section 2.2.3) or HERMIT's lemmas (Section 5.2). Finally, it closes with a survey of the HERMIT Dictionary, highlighting key transformations as a means of demonstrating the scope of HERMIT's transformation capabilities (Section 4.5).

4.1 Example

One of HERMIT's defining features is the ability to transform GHC Core programs interactively, allowing the user to explore transformations in an ad-hoc manner and gain intuition for general transformations from specific instances. This chapter informally introduces the reader to HERMIT's interactive transformation capabilities by performing a small example program transformation.

The intent is to give a sense of HERMIT's capabilities, and the reader is encouraged to install HERMIT and follow along. HERMIT can be installed using Haskell's package management system, Cabal [Jones, 2005]. Assuming a working installation of the Haskell Platform [Coutts et al., 2008], the following two commands are sufficient:

```
$ cabal update
$ cabal install hermit
```

To demonstrate this interactive capability, an inefficient definition of the *mean* function is transformed into a more efficient version. The transformation begins with the following clear specification of *mean*:

```
mean :: [Int] → Int
mean xs = sum xs ‘div‘ length xs
```

This definition is inefficient because it traverses the input list *xs* twice, requiring space linear in the length of the list. After the first traversal, the entire list will be resident in memory, as it cannot be garbage collected until the second traversal is made. The goal is to derive the following more efficient version of *mean* using a series of correctness-preserving transformations.

```
mean :: [Int] → Int
mean xs = case sumlength xs of
                (s, l) → s ‘div‘ l
  where sumlength :: [Int] → (Int, Int)
        sumlength []       = (0, 0)
        sumlength (i : is) = case sumlength is of
                                  (s, l) → (i + s, 1 + l)
```

This version of *mean* is less-obviously correct, but more efficient because it only traverses the input list once and, after GHC's other optimization passes, requires only constant space.

```
module Main where
import Prelude hiding (sum, length)
mean :: [Int] → Int
mean xs = sum xs ‘div‘ length xs
sum :: [Int] → Int
sum []       = 0
sum (x : xs) = x + sum xs
length :: [Int] → Int
length []       = 0
length (x : xs) = 1 + length xs
main :: IO ()
main = print $ mean [1 . . 10]
```

Figure 4.1: Mean.hs: Haskell Source for the Mean Example.

HERMIT is invoked on the Haskell source file listed in Figure 4.1. Definitions for *sum* and *length* are provided so the transformation is more concise for presentation purposes. An equivalent transformation could be performed with the *sum* and *length* defined in the Haskell prelude.

```
$ hermit Mean.hs
```

This causes GHC to parse, typecheck, and desugar the Haskell source to GHC Core, which HERMIT then presents to the user, along with a prompt.

```
module main:Main where
  sum :: [Int] → Int
  length :: [Int] → Int
  mean :: [Int] → Int
  main :: IO ()
  main :: IO ()

hermit>
```

HERMIT initially presents a summary of the module, displaying only the type signatures of top-level functions. In general, HERMIT's pretty-printer displays types in the color green.

In order to transform *mean*, the `rhs-of` command is used to focus on the right-hand side of *mean*'s binding. Note that names in the target code are prefixed with an apostrophe when referenced in HERMIT command arguments.

```
hermit> rhs-of 'mean

λ xs → div ▲ $fIntegralInt (sum xs) (length xs)
```

The *div* function, called in infix position in the Haskell source, is now in prefix position. *div*'s first argument, displayed as a green triangle, is a type argument. In this case, the type is *Int*, but HERMIT elides type arguments by default, instead using these triangular placeholders. Visually, this helps maintain the correspondence to the Haskell source, while still indicating the type is present. The *$fIntegralInt* argument to *div* is a dictionary (Section 2.2.2).

This example is not concerned with types, so they can be hidden altogether.

```
hermit> set-pp-type Omit

λ xs → div $fIntegralInt (sum xs) (length xs)
```

One of HERMIT's *crumb* commands can be used to move into the body of the function (Section 4.2.2). Crumbs take their name from the idea of leaving a trail of bread crumbs. A sequence of crumbs denotes a path in the abstract-syntax tree. HERMIT provides a crumb for each combination of parent and child node. For instance, from an application node, one can descend into the function with the `app-fun` crumb, or the argument, with the `app-arg` crumb.

```
hermit> lam-body

div $fIntegralInt (sum xs) (length xs)
```

The essence of this transformation is to return the result of *sum* and *length* in a tuple. HERMIT provides a means of tupling let bindings, so the first step is to create let bindings for the applications of the two functions. Rather than navigate down to the application expressions using crumbs, the `application-of` navigation command is used.

```
hermit> { application-of 'sum ; let-intro 's }
hermit> { application-of 'length ; let-intro 'l }

div $fIntegralInt (let s = sum xs in s) (let l = length xs in l)
```

HERMIT maintains a stack of foci in the AST. The open brace ({) pushes the current focus from the stack, and the close brace (}) pops the top focus on the stack. In this case, HERMIT is told to remember the current focus, move the focus to the application of *sum* (or *length*), introduce the let expression, then return to the current focus. The semicolon is a statement separator.

Now to float the newly introduced let expressions outward, in preparation for tupling the bindings. This is done with a general purpose `let-float` command, which floats let expressions from several different kinds of expressions. A high-level strategy named `innermost` applies `let-float` in a bottom-up manner. Whenever let floating succeeds, `innermost` will repeat the traversal. An innermost traversal terminates when the given transformation no longer applies anywhere in the AST.

```
hermit> innermost let-float

let l = length xs
    s = sum xs
in div $fIntegralInt s l
```

The last bit of housekeeping is to reorder the bindings to the match the desired order of the tuple.

```
hermit> reorder-lets ['s,'l]

let s = sum xs
    l = length xs
in div $fIntegralInt s l
```

Now to tuple the let bindings. Let tupling can be seen as the dual of case reduction, in that it introduces a tuple constructor application, which is immediately scrutinized by a case expression. Such a transformation is generally not implemented in an automatic program optimizer, because it increases both allocation and runtime, but is necessary to proceed with the derivation. The `'sl` argument to `let-tuple` is the desired name for the case binder which will be introduced.

```
hermit> let-tuple 'sl

case (,) (sum xs) (length xs) of sl
  (,) s l → div $fIntegralInt s l
```

The case scrutinee is an expression which builds a tuple of sum and length values using the *sum* and *length* functions. The rest of the derivation calculates a more efficient function which does this. To do so, we start by moving into the case scrutinee with the `case-expr` crumb and abstracting it over the input list.

```
hermit> case-expr ; abstract 'xs
(λ xs → (,) (sum xs) (length xs)) xs
```

Now to name the new function and let-bind it. Recall that the `app-fun` crumb moves the focus to the left-hand side of an application.

```
hermit> app-fun ; let-intro 'sumlength
let sumlength = λ xs → (,) (sum xs) (length xs) in sumlength
```

The intent is that *sumlength* will eventually be a recursive function. The let expression we just introduced is non-recursive, so we descend into the let binding and convert it to a recursive binding.

```
hermit> let-bind ; nonrec-to-rec
rec sumlength = λ xs → (,) (sum xs) (length xs)
```

Before transforming *sumlength* in earnest, we tell HERMIT to remember the current definition under the name "sumlen". Remembered bindings can later be folded or unfolded, and make explicit the notion of "looking up the page" in a pen and paper derivation. Recall that a recursive binding group may contain multiple bindings. Even though this newly created group contains only one binding, we must explicitly focus on it in order to remember it.

```
hermit> binding-of 'sumlength ; remember sumlen
sumlength = λ xs → (,) (sum xs) (length xs)
```

Moving into the function body, case-splitting on *xs* introduces a case expression with an alternative for each constructor of *xs*'s type. The right-hand side of each alternative is the original expression with *xs* replaced by the data constructor application making up the left-hand side of the alternative.

```
hermit> def-rhs ; lam-body ; case-split-inline 'xs
case xs of w
  [] → (,) (sum []) (length [])
  (:) a b → (,) (sum ((:) a b)) (length ((:) a b))
```

Each application of *sum* and *length* is to a manifest data constructor, meaning we can perform a bit of evaluation. We unfold both functions at each call site. The command we give can be read: "at any application site, try unfolding either *sum* or *length*".

```
hermit> any-call (unfold 'sum <+ unfold 'length)

case xs of w
  [] →
    (,) (case [] of wild
           [] → I# 0
           (:) x xs → (+) $fNumInt x (sum xs))
        (case [] of wild
           [] → I# 0
           (:) x xs → (+) $fNumInt (I# 1) (length xs))
  (:) a b →
    (,) (case (:) a b of wild
           [] → I# 0
           (:) x xs → (+) $fNumInt x (sum xs))
        (case (:) a b of wild
           [] → I# 0
           (:) x xs → (+) $fNumInt (I# 1) (length xs))
```

With the bodies of *sum* and *length* inlined, case reduction is possible in several places. Rather than explicitly case reduce, we use `simplify`, which performs case reduction, dead let elimination, and other cleanup.

```
hermit> simplify

case xs of w
  [] → (,) (I# 0) (I# 0)
  (:) a b →
    (,) ((+) $fNumInt a (sum b)) ((+) $fNumInt (I# 1) (length b))
```

The desired base case for an empty list is now established. We focus on the alternative for the non-empty list which calls *sum* and *length* on the tail of the list.

```
hermit> case-alt 1 ; alt-rhs

(,) ((+) $fNumInt a (sum b)) ((+) $fNumInt (I# 1) (length b))
```

Again, we want to tuple these calls so we can replace them with a call to *sumlength*. As before, we introduce let bindings for each call and float them upward.

```
hermit> { application-of 'sum ; let-intro 's }
hermit> { application-of 'length ; let-intro 'l }
hermit> innermost let-float

let l = length b
    s = sum b
in (,) ((+) $fNumInt a s) ((+) $fNumInt (I# 1) l)
```

As before, we reorder and tuple the let bindings, creating the case expression which projects the results for the tail of the list from a tuple.

```
hermit> reorder-lets ['s,'l] ; let-tuple 'sl

case (,) (sum b) (length b) of sl
  (,) s l → (,) ((+) $fNumInt a s) ((+) $fNumInt (I# 1) l)
```

Now the key step in this derivation. The case scrutinee is an instance of the body of *sumlength* which we told HERMIT to remember. We can tell HERMIT to fold the remembered definition, replacing the instantiated body with an application of *sumlength* to the tail of the list. This eliminates the calls to *sum* and *length*, and makes *sumlength* self-recursive.

```
hermit> { case-expr ; fold-remembered sumlen }

case sumlength b of sl
  (,) s l → (,) ((+) $fNumInt a s) ((+) $fNumInt (I# 1) l)
```

Though the derivation is completed, for presentation purposes, we move the focus back to the top of the module, then focus on the binding for *mean*, to view the result. The new *sumlength* function is bound within the case scrutinee of *mean*, so we float it outward to make clear the correspondence with the desired result.

```
hermit> top ; binding-of 'mean ; innermost let-float

mean =
  let rec sumlength = λ xs →
            case xs of w
              [] → (,) (I# 0) (I# 0)
              (:) a b →
                case sumlength b of sl
                  (,) s l → (,) ((+) $fNumInt a s) ((+) $fNumInt (I# 1) l)
  in λ xs →
       case sumlength xs of sl
         (,) s l → div $fIntegralInt s l
```

This example demonstrates the equivalence of the two definitions of *mean* using a series of correctness-preserving transformations to transform one into the other. This sort of reasoning, motivated in Section 1.1.3, can be seen as specification refinement. An executable specification of *mean* has been refined into a more efficient implementation.

We performed this transformation interactively, though the derivation can be saved as a HERMIT script for future use. To do so, we invoke the `save` command, which writes out the commands in this session to a file.

```
hermit> save "Mean.hec"
[saving Mean.hec]
```

To run the derivation script automatically in the future and compile the transformed result, we can invoke HERMIT on `Mean.hs`, telling it to target the *Main* module with the `Mean.hec` script, resuming compilation if the script executes successfully.

```
$ hermit Mean.hs +Main Mean.hec resume
```

Since the `hermit` command itself is a thin wrapper which invokes GHC with special flags, the derivation can be integrated into existing build scripts directly. This is discussed in detail in Section 6.2.

The similarity between this interactive transformation and a pen and paper derivation is intentional. Recall that the goal of HERMIT is to *mechanize* the sort of semi-formal reasoning Haskell programmers already do, rather than to automate any given transformation (though HERMIT can certainly be used to construct automated transformations).

Mechanizing these reasoning steps allows the programmer to focus on what needs to be done, rather than getting lost in the details of how to manipulate the program correctly. For instance, the `reorder-lets` command ensures that no variables are captured or left unbound. More powerful commands such as `simplify` perform many tedious substeps that pen and paper derivations often gloss over. As demonstrated, HERMIT also allows the derivations to be re-used during future compilation, enforcing a correspondence between a changing specification and its derived implementation.

4.2 KURE

A significant portion of the HERMIT implementation is dedicated to specifying transformations over GHC Core programs. To ease this implementation effort, a means of specifying modular, reusable transformations was required. Additionally, due to the large number of primitive transformations, it was paramount that transformations be both composable and reusable to the extent possible. GHC Core programs are composed of multiple mutually recursive data types, so support for generic traversals of these types was also important. Finally, HERMIT's interactive features required that transformations could be targeted to a specific point in the tree.

Strategic programming languages [Visser, 2005] are a promising approach to this problem, but previous strategic languages were either untyped [Bravenboer et al., 2008] or required run-time type comparisons [Lämmel and Visser, 2002]. Given that HERMIT is implemented in Haskell, the transformation language provided by HERMIT would ideally be strongly-typed. GHC Core programs tend to be large trees, so the ability to express *statically selective* traversals, which do not descend into subtrees with certain types, was important for efficiency reasons.

Thus, the Kansas University Rewrite Engine (KURE) was developed to meet these needs. KURE is a strongly typed strategic programming language which supports static selectivity, and has the ability to rewrite nodes of different types during the same traversal, automatically maintain a context during generic traversals, and express traversals with arbitrary monadic effects. No other library for strategic or generic programming provides this combination of features. KURE was initially included as part of HERMIT, but has since developed into an independently-useful library with broad applications [Sculthorpe et al., 2014].

This section describes HERMIT's support for using KURE to transform GHC Core programs. It does so by describing the various types used to specialize generic KURE strategies to be GHC Core transformations. The concepts of KURE, such as universes, traversals, promotion, and congruence combinators, were presented in Section 2.3.

4.2.1 Universes

GHC implements GHC Core using several different data types. The entire module currently being compiled is encapsulated by the $ModGuts$ type. Within $ModGuts$, the $CoreProgram$ type is a list of top-level binding groups. Each binding group consists of $CoreBind$ values. A $CoreBind$ is either a single, non-recursive pair of Var and $CoreExpr$, or a list of pairs of Var and $CoreExpr$

representing a recursive binding group. Expressions, defined by the *CoreExpr* type, can include *Var*s, *Literal*s, *CoreAlt*s, *CoreBind*s, *Type*s, *Coercion*s, and *Tickish*s.

HERMIT is primarily concerned with transforming expressions, but occasionally a transformation may also need to traverse types and coercions. To this end, HERMIT defines two universes. The first, *Core*, is a universe for traversing nodes which contain expressions. The second, *CoreTC*, is the *Core* universe plus *Type*s and *Coercion*s. Note that *CoreTC* is actually defined in terms of *Core* and a third universe of only *Type*s and *Coercion*s.

```
data Core = GutsCore ModGuts    -- The module.
          | ProgCore CoreProg   -- A program (a telescope of top-level binding groups).
          | BindCore CoreBind   -- A binding group.
          | DefCore  CoreDef    -- A recursive definition.
          | ExprCore CoreExpr   -- An expression.
          | AltCore  CoreAlt    -- A case alternative.
data CoreTC = Core Core | TyCo TyCo
data TyCo = TypeCore Type | CoercionCore Coercion
```

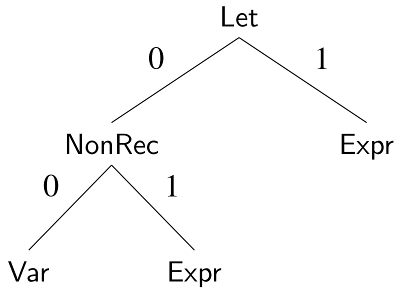
The *Core* universe will be the focus on the remainder of this section. *CoreTC* is used less often, mostly by the pretty printer and some navigation commands, and is in any case entirely analogous. When rewriting *CoreExpr*s, the *Core* universe is targeted because it does not traverse type and coercion terms, which do not contain expressions. This static selectivity results in better traversal performance.

4.2.2 Crumbs

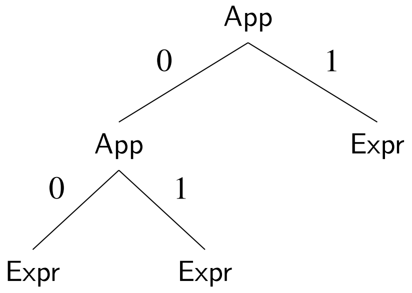
Targeting transformation using a *path* is a key operation in HERMIT. For instance, the user may wish to transform the body of a specific function definition. A path is used to descend to the desired location, which can then be transformed.

KURE provides strategies for generating and using generic paths in this way. A path in KURE is a list of *crumbs*, so named because they act as proverbial bread crumbs, determining which child to descend into at each point along the path, starting at the root of the tree. The strategies KURE provides are necessarily polymorphic in the crumb type, allowing crumbs to be specific to the universe type being traversed.

A simple means of constructing such a path would be to use a list of integers. If the children of each node were numbered in some arbitrary fashion, say left-to-right, from zero, then each crumb in the path would be the integer of the child into which the traversal should descend. For example, the path to the right-hand side of the binding in a non-recursive let-expression would be $[0, 1]$.



However, denoting paths this way is not very specific. A given path of integers may apply to many different ASTs. For example, the path $[0, 1]$ would apply equally well to a tree of applications.



In both cases, an expression is targeted by the path, so a transformation applied using the path may succeed, even if the user only intended it to apply to the right-hand side of a non-recursive let binding.

In practice, a more specific means of specifying paths was found to be necessary to make transformations more robust to changes in the target program. This is especially true when manually specifying paths in scripts. Changes in the source code of a module targeted by HERMIT usually result in different GHC Core. The less specific integer paths may inadvertently still apply, but result in an unexpected destination, causing the intended rewrite to fail. Worse, the intended rewrite

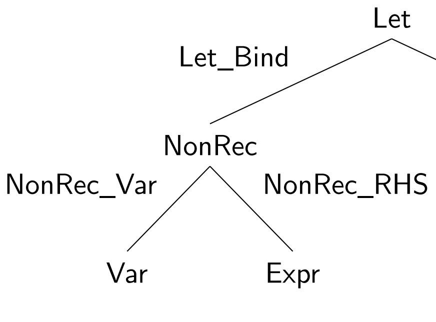
```
data Crumb = ...
           | NonRec_RHS | NonRec_Var
           | Rec_Def Int
           | ...
           | Def_Id | Def_RHS
           | ...
           | App_Fun | App_Arg
           | Lam_Var | Lam_Body
           | Let_Bind | Let_Body
           | Case_Scrutinee | Case_Binder | Case_Type | Case_Alt Int
           | ...
           | Alt_Con | Alt_Var Int | Alt_RHS
           | ...
```

Figure 4.2: The *Crumb* Type.

may succeed, but in the wrong place. More specific paths give a better error message (that the path is invalid) to the user, and make scripts easier to read.

To this end, HERMIT defines a crumb type, called *Crumb*, which is specialized to the *Core* universe. It is a large data type, with one constructor for each possible combination of parent and child. Figure 4.2 gives a sampling of the *Crumb* type.

In addition to specifying which child to descend into, it specifies the expected current node. Thus, each crumb denotes movement from a specific parent to a specific child, rather than from an arbitrary parent to an arbitrary child that happens to be in the correct position. With these crumbs, the path $[0, 1]$ would in fact be [Let_Bind, NonRec_RHS], which would not apply to a tree of applications.



4.2.3 The HERMIT Context

The context for HERMIT's transformations over GHC Core is implemented by the *HermitC* type. However, all the transformations in the HERMIT Dictionary are overloaded in the context type so that they may be used with any context that supplies the necessary information. This is useful if the user desires to extend the context with additional information. *HermitC* is a context type which implements all the interfaces required by transformations in the Dictionary. Accordingly, this section describes *HermitC* in terms of these interfaces, rather than as a concrete implementation.

4.2.3.1 Recording Bindings

The most important function of the context is to collect in-scope bindings during traversal. This makes all type- and value-level bindings which are in-scope locally available to a transformation. HERMIT supplies a type class for contexts which can accumulate the information HERMIT needs about bindings. This class is used by congruence combinators to update the context.

```
class AddBindings c where
  addHermitBindings :: [(Var, HermitBindingSite, AbsolutePathH)] → c → c
```

The *addHermitBindings* class method adds *parallel binding groups* to the context. A parallel binding group is a group of bindings which occur at the same depth in the tree. Examples of parallel groups with multiple binders include case alternative patterns and recursive binding groups. Other forms of binding give rise to singleton groups.

The information for a single binding is the binder itself (a *Var*), the path to the binding, and information about the type of the binding, which is encapsulated by the *HermitBindingSite* type.

HermitBindingSite records the nature of the binding (whether it is bound by a lambda, a let expression, case alternative, etc) and, potentially, unfolding information.

```
data HermitBindingSite = LAM
                       | NONREC CoreExpr
                       | REC CoreExpr
                       | SELFREC
                       | MUTUALREC CoreExpr
                       | CASEALT
                       | CASEBINDER CoreExpr (AltCon, [Var])
                       | FORALL
```

Binders bound by lambdas, universal quantifers (in types), and case alternatives are recorded by LAM, FORALL, and CASEALT, respectively. These binding sites do not record unfolding information because doing so would require evaluation. For instance, while applying a transformation to the body of a lambda expression, the context contains x as a LAM binding site.

$(\lambda x \rightarrow \mathit{body})\ \mathit{arg}$

In order to get the unfolding for x, the entire expression would have to be β-reduced. Even though, in this case, such a β-reduction is available, HERMIT makes no attempt to do this automatically when building the context.

When applying a transformation to the body of a let expression, the let binders are recorded in the context using either NONREC or REC depending on whether the let is non-recursive or recursive, respectively. These constructors carry a $\mathit{CoreExpr}$, which is the right-hand side of the binding. This can be used to inline the variable in question. Noting whether a binding is recursive or non-recursive is important when performing the depth check during inlining (Section 4.5.1).

Recall that case expressions in GHC Core (Section 2.2) differ from case expressions in Haskell in that they have an explicit type annotation and a *case binder*. The case binder binds the scrutinized expression over the right-hand side of each alternative. This binder is unique in that it actually has two possible unfoldings. Consider the following case expression, where b is the case binder:

```
case f x y of b
  Just z → ... b ...
  Nothing → ...
```

If b were to be unfolded in the right-hand side of the first alternative, both $f\ x\ y$ and Just z are valid unfoldings. In most cases, the latter behavior is prefered because it includes the result of the computation performed by the case expression. However, occasionally the first behavior is desired because it enables a subsequent transformation, even though it nominally duplicates computation. For instance, inlining the scrutinized expression may enable the application of a GHC rewrite rule.

For this reason, case binders are recorded with both possible unfoldings using the CASEBINDER constructor. The $\mathit{CoreExpr}$ argument to CASEBINDER is the scrutinee, while the pattern for the current alternative is stored as a constructor along with a list of pattern binders for that constructor.

The most interesting constructors for $\mathit{HermitBindingSite}$ are SELFREC and MUTUALREC. These are used to record binders in a recursive binding group when descending into the right-hand side of one of the binders in the group. As an example, consider descending into the right-hand side of x in the following recursive let expression.

$$\begin{array}{l}\textbf{let}\ x = e_1\\ \quad\ \ y = e_2\\ \quad\ \ z = e_3\\ \textbf{in}\ \ldots\end{array}$$

When descending into e_1, the context will be extended with two MUTUALREC entries, for y and z. These contain the appropriate right-hand sides as unfoldings (e_2 and e_3, respectively). It will also be extended with a SELFREC entry for x. Note that SELFREC *does not* carry an unfolding, so there is no unfolding information for x while rewriting its own right-hand side.

This is for good reason. Consider, hypothetically, that the context did provide an unfolding for x, like it does for other bindings in the recursive group, and that x is recursive. The following two rewrites would behave in subtly different ways when applied to let expressions.

$$\begin{array}{l}\mathit{rr1} = \mathit{replicateR}\ 2\ (\mathit{onetdR}\ (\mathit{promoteExprR}\ (\mathit{inlineR}\ (== x))))\\ \mathit{rr2} = \mathit{focusR}\ (\mathit{rhsOfT}\ x)\ (\mathit{replicateR}\ 2\ (\mathit{onetdR}\ (\mathit{promoteExprR}\ (\mathit{inlineR}\ (== x)))))\end{array}$$

Both rewrites would unfold the definition of x twice, but the result would differ. The first rewrite ($\mathit{rr1}$) begins at the let expression and performs a top-down traversal, applying $\mathit{inlineR}$ to the first place it succeeds (an occurrence of x). It then performs a second top-down traversal, again starting from the overall let-expression, performing a second inlining. This second $\mathit{inlineR}$ will use the new definition of x which was the result of the first inlining. This is because the rewrite descends past the binding of x twice. The second rewrite ($\mathit{rr2}$) descends past the binding of x once using rhsOfT, then applies the two top-down traversals with their inlining steps, using the same definition of x each time.

The subtlety of which unfolding of x is used compounds in more complex composite rewrites, and makes refactoring such rewrites difficult. Similiar problems arise when converting HERMIT scripts into rewrites, as described in Section 3.4.2. This conversion replaces the statement sequencing operator (;) of the Shell language with the KURE sequencing operator ($\ggg$). Two rewrite statements separated by (;) each begin at the top of the module, meaning the second statement occurs in a (potentially) different context than the first. This is similiar to the *rr1* example. The same two rewrites combined using ($\ggg$) will see the same context, similiar to *rr2*.

To avoid tripping over this subtle difference in semantics, HERMIT elects to not include an unfolding with SELFREC. To unfold x within its own right-hand side requires first telling HERMIT to remember the definition of x (Section 5.10.6). In this way, the user is explicit about which unfolding is desired.

4.2.3.2 Accessing Bindings

HERMIT defines two classes for accessing the binding information stored in the context. The first returns a set of Vars using the GHC-defined *VarSet* type. This can be used in situations where it is only important to determine if a variable is bound, and unfolding information is not needed.

```
class BoundVars c where
  boundVars :: c → VarSet
```

To access unfolding information requires the *ReadBindings* interface. It has methods for accessing the current binding depth, as well as the unfolding information recorded by *AddBindings*.

```
class BoundVars c ⇒ ReadBindings c where
  hermitDepth :: c → BindingDepth
  hermitBindings :: c → Map Var HermitBinding
data HermitBinding = HB BindingDepth HermitBindingSite AbsolutePathH
type BindingDepth  = Int
```

The binding depth represents the number of parallel bindings groups that have been added to the context by *addHermitBindings*. Note that this means the binding depth is not equivalent to the length of the path to that binding. Many nodes in the AST do not bind values (application nodes, for instance), and hence are not counted for depth purposes. Depth is recorded in order to avoid variable capture during inlining, which is discussed in Section 4.5.1.

4.2.3.3 In-scope RULES

GHC rewrite rules for the current module are stored in the *IdInfo* (Section 2.2.1.4) of the binder which forms the head of the left-hand side of the RULES pragma. For instance, the following rewrite rule would be stored in the *IdInfo* for *abs*.

```
{-# RULES "abs-rep-id" [~] forall e. abs (rep e) = e #-}
```

GHC does this for efficiency reasons. *IdInfo* is propagated from binder to occurrence by GHC's substitution algorithm, meaning applicable rules are always available exactly where they might be applied, and only when the appropriate identifiers are in scope. Additionally, when GHC generates specializations for a function, these specializations are stored as rules on the binder for the function.

Similar to bindings, HERMIT accumulates these rules in the context as it descends the AST during a traversal. This means all in-scope rules are locally available to tranformations using a context that implements the following class:

```
class HasCoreRules c where
  hermitCoreRules :: c → [CoreRule]
```

This class is only for reading the rule environment. Since rules only appear in a top-level environment and on binders, they are added to the context by the *AddBindings* instance for *HermitC*.

4.2.3.4 Paths

KURE defines its path generating and focusing strategies in terms of two type classes. This allows the strategies to be defined generically for any context and crumb type which together implement an instance of these classes.

```
class ExtendPath c crumb | c → crumb where
  (@@) :: c → crumb → c
class ReadPath c crumb | c → crumb where
  absPath :: c → AbsolutePath crumb
```

These classes are used primarily by congruence combinators (Section 4.2.4) to update the path during traversal, and to provide the current path to calls of *addHermitBindings*.

```
lamT :: (AddBindings c, ExtendPath c Crumb, ReadPath c Crumb, Monad m)
      ⇒ Transform c m Var a1
      → Transform c m CoreExpr a2
      → (a1 → a2 → b)
      → Transform c m CoreExpr b
lamT t1 t2 f =
  transform (λc exp →
              case exp of
                Lam v e → let c′ = addHermitBindings [(v, LAM, absPath c @@ Lam_Var)] c
                           in f $⃝ applyT t1 (c  @@ Lam_Var)   v
                                ⊛ applyT t2 (c′ @@ Lam_Body) e
                _       → fail "not a lambda.")
lamAllR :: (AddBindings c, ExtendPath c Crumb, ReadPath c Crumb, Monad m)
         ⇒ Rewrite c m Var
         → Rewrite c m CoreExpr
         → Rewrite c m CoreExpr
lamAllR r1 r2 = lamT r1 r2 Lam
```

Figure 4.3: Congruence Combinators for the Lam Constructor of $CoreExpr$.

4.2.4 Congruence Combinators

HERMIT defines a set of *congruence combinators* for the types in the $Core$ universe, as recommended by KURE (Section 2.3.4.2). Congruence combinators serve both as guards, to ensure a transformation is applied to an expression with the desired structure, and as means of ensuring that contextual information is properly updated while traversing the target program.

Two congruence combinators, one for transformations and one for rewrites, are defined for each constructor of each type in the $Core$ universe. While they cannot be automatically generated (due to the non-regularity of updating the context), their form is otherwise extremely regular. The congruence combinator for rewrites is always defined in terms of the one for transformations. As an example, the two combinators which are defined for the Lam constructor to $CoreExpr$ are featured in Figure 4.3.

In contrast to the example congruence combinators in Section 2.3.4.2, those in HERMIT are overloaded in the context type, constraining it only to the interfaces necessary for updating the context during traversal. This permits reuse of the congruence combinators for different context types which may extend the default *HermitC* context. In this case, the call to *addHermitBindings* requires an *AddBindings* constraint (Section 4.2.3.1). The call to *absPath* requires a *ReadPath* constraint and the call to the (@@) combinator requires the *ExtendPath* constraint (both in Section 4.2.3.4).

The instance of KURE's *Walker* class for the *Core* universe is defined in terms of these congruence combinators, in the recommended style of Figure 2.4 in Section 2.3.4.2. Thus, this *Walker* instance is also overloaded in the context type, meaning custom traversals of the *Core* universe can also be reused for other context types.

Use of congruence combinators leads to an idiomatic style of constructing local transformations. A typical transformation projects components from the structure of the expression which are relevant to the transformation, extracts information from those components, then uses the information to construct a result. If the computation which extracts information from the components relies on contextual information, this can lead to subtle bugs. These bugs can be mitigated by using congruence combinators.

For example, consider defining a hypothetical rewrite which inlines a specific variable, but only when it occurs in the body of a lambda abstraction. One might start with the following implementation, which matches on explicit lambda expressions, then calls a KURE strategy for applying the inlining anywhere in the body in a bottom-up traversal. A valid implementation of *inlineR*, which always inlines a given variable, is assumed.

```
 -- Note: this definition is incorrect, see discussion below
inlineInBodyR :: Monad m ⇒ Var → Rewrite HermitC m CoreExpr
inlineInBodyR v = do
  Lam b e ← idR
  e′ ← return e ⋙ extractR (anybuR (promoteExprR (inlineR (== v))))
  return $ Lam b e′
```

Following the pattern, this transformation projects the relevant components of the expression (the binder and body of the lambda expression), extracts information from them (the new body, with *v* inlined), then constructs a result (the new lambda expression).

There is a subtle bug is this implementation, however. While not obvious from this code, one of the safety checks performed by *inlineR* is to ensure that all variables in the result of the inlining are in scope. This check is necessarily context-dependent, since the context is the source of information about in-scoped-ness. If the result happens to contain an occurrence of the lambda-binder b, this check will fail. The call to *anybuR* happens in the context of the overall lambda-expression, not the context of the actual body.

The definition could be altered to manually ensure that b is in the context of the call to *anybuR* by projecting the current context and calling *addHermitBindings* (Section 4.2.3.1) to construct a new one. But this is exactly the problem that congruence combinators solve! Thus, it is better to rewrite the transformation in terms of the *lamAllR* congruence combinator from Figure 4.3.

```
inlineInBodyR :: Monad m ⇒ Var → Rewrite HermitC m CoreExpr
inlineInBodyR v = lamAllR idR (extractR (anybuR (promoteExprR (inlineR (== v)))))
```

This example was contrived, but bugs arising from incorrectly maintaining the context were common early in HERMIT's development because congruence combinators were not exploited. Any time a transformation is applied to a component of the current expression without wrapping that transformation in a congruence combinator, care must be taken to ensure a proper context is provided. A 'proper' context does not just include appropriate bindings. The context also tracks information related to shadowing (binding depth), the current path, and in-scope GHC rewrite rules. Any transformation that is sensitive to this information may fail in unexpected ways. Relying on congruence combinators eliminates this large class of bugs in practice.

Congruence combinators can also be used to construct non-structural guards. Ordinary monadic pattern matching can be used to guard on the structure of an expression. Congruence combinators can be used to guard on both structural and non-structural aspects of the expression.

For example, the following rewrite attempts to float a let-expression from the right-hand side of an application, failing if variable capture would occur. It relies on two rewrites, *freeVarsT* and *letVarsT*, which return the free variables of an expression and the variables bound by a let-expression, respectively. The intersection of the free variables of the left-hand side and the bound variables of the right-hand side is computed. If the intersection is non-empty, capture would occur.

```
letFloatArgR :: Monad m ⇒ Rewrite HermitC m CoreExpr
letFloatArgR = do
  captures ← appT freeVarsT letVarsT intersect
  guardMsg (null captures) "floating would lead to variable capture."
  App f (Let b e) ← idR
  return $ Let b (App f e)
```

Congruence combinators, overall, were found to be very advantageous when defining primitive transformations because they alleviate the need to explicitly manage the context.

4.2.5 The HERMIT Monad

The monad used by HERMIT transformations is called $HermitM$. Conceptually, it is a reader and state transformer on top of GHC's $CoreM$ monad. The reader environment provides access to a debugging channel which is passed in as part of the $KernelEnv$ argument supplied to calls to the Kernel API (Figure 3.4). This channel is used primarily by the debugging primitives in Section 4.5.6. The state carried by $HermitM$ is the list of available lemmas, which are discussed in Section 5.2. The initial set of lemmas is provided by the Kernel, modified by the transformation, then saved by the Kernel, alongside the resulting GHC Core program. The interface for accessing lemma state is entirely conventional to state monads.

The rest of the functionality of $HermitM$ is inherited from $CoreM$. This includes an interface for generating unique identifiers, which is used by HERMIT's name creation primitives. It also includes functionality for looking up Vars by $Name$, which HERMIT uses to find identifiers in other modules. Since $CoreM$ is built on IO, it also includes a $MonadIO$ instance.

In general, the HERMIT user should never deal with $HermitM$ directly. All the functionality has been lifted into KURE transformations which are exposed in the Dictionary (Section 4.5).

4.2.6 Conventions

As transformations are often performed using the $HermitC$ context and $HermitM$ monad types, HERMIT provides the following type synonyms to simplify type signatures.

```
type TransformH a b = Transform HermitC HermitM a b
type RewriteH a = Rewrite HermitC HermitM a
```

Additionally, all transformations provided by HERMIT follow some conventions:

- Rewrites either modify the term, or fail. This makes it viable to use such rewrites with iteration strategies which repeat until failure. Succeeding with an unmodified term would lead to unbounded iteration.

- Primitive rewrites do not perform traversal. They apply only to the local expression, and are lifted into traversals using KURE's strategy combinators. For instance, the primitive $inlineR$ matches on a single variable, replacing it with its unfolding expression. It can then be lifted, using a traversal strategy such as $anytdR$, to apply anywhere in a given tree.

- The type of a transformation is as specific as possible. For instance, $inlineR$ applies to $CoreExpr$, not one of the universe types, because it is a rewrite on expressions. It can be promoted, if necessary, using KURE's promotion combinators (Section 2.3.4), to any universe it is a member of.

4.3 Names

An important practical aspect of transforming GHC Core programs is working with, and creating, named identifiers. These identifiers may be bound locally in the module being transformed, or imported from another module, possibly in another package.

GHC's internal types for named identifiers were summarized in Section 2.2.1. HERMIT opts for a simpler data type. The goal is that a Haskell programmer, unfamiliar with the details of GHC's internal name types, but familiar with Haskell's simple module hierarchy, can productively create and manipulate names using HERMIT.

Ultimately, HERMIT transformations must modify or introduce the Var type, as it is the type-annotated identifier used by GHC Core. HERMIT's monad provides a unique supply for creating new local variables, and primitive transformations for modifying existing variables. More challenging is introducing a variable which represents an external, imported identifier.

To do so using GHC's existing plugin API, one must first create an $OccName$, specifying both the string representation of the identifier and the desired namespace. Then, a qualified $RdrName$ may be created by specifying a module name. If the specified module's interface has not already been loaded into GHC's caches, it must be. Modules are loaded on an as-needed basis, when they are imported explicitly in the source. With all this done, the $RdrName$ may be looked up in the cache, returning a $Name$. The $Name$ may in turn be looked up, returning a Var.

This process, and the design of GHC's name types, follows from the steps taken by GHC's front end. The namespace is paired with the occurrence name because both are known, by the parser, when the $OccName$ is created. The $RdrName$ adds the module name because this is determined later, by the renamer, which resolves the scoping of module import statements. The necessary module interfaces are loaded after determining which packages are in-scope to the compilation session. Once this package information has been determined, a $Name$ can be created. Next, typechecking creates the full-fledged Var.

Given the occurrence name, module name, and namespace, all the remaining steps can be performed automatically. This informs the design of HERMIT's identifier type:

```
data HermitName = HermitName (Maybe ModuleName) String
```

Note that a $HermitName$ does not specify a namespace. This is instead determined at the time of use. The module name is optional so that $HermitName$ can represent local, unqualified names.

This type is easy to construct (recall that $ModuleName$ is essentially a $String$), and leads to the simple interface for finding external identifiers described in Section 4.5.3.

4.4 Folds

KURE transformations are a powerful means of expressing transformations when all the matching conditions of the transformation are known in advance. For instance, a β-reduction transformation must always match on an application where the function is an explicit lambda expression.

However, it is often the case where the matching conditions are only known at HERMIT runtime. The most common example is when folding a function definition, in the course of fold/unfold reasoning [Burstall and Darlington, 1977]. The matching conditions are determined by the structure of the expression representing the function body, and vary by function.

A *fold* is a first-order pattern matching operation for replacing an expression which matches a pattern with another expression. Folds are used to implement several key transformations in HERMIT, and the performance of the matching operation has been found to be critical when transforming large programs. This section formalizes HERMIT's fold operation and describes key parts of its implementation.

4.4.1 Definition

First, some terminology. An *expression context* is a GHC Core expression with a *hole*, into which an arbitrary expression of the appropriate type can be placed. A *multi-hole expression context* is a GHC Core expression with zero or more named holes. Two holes with the same name must have the same type and be filled with the same expression. A *pattern* is a multi-hole expression context used for matching on a concrete expression. A *template* is a multi-hole expression context used to instantiate the resulting expression. An *equality* is a triple of pattern, template, and a list of named holes. An equality states that the pattern and template are equivalent for all possible assignments to the holes. An equality is only valid if the named holes in the template are a subset of those in the pattern.

Given a pattern $\mathbb{C}$ with a named hole h and expression e, $\mathbb{C}\ [e/h]$ is the operation of substituting e for all occurrences of h in $\mathbb{C}$. If $\mathbb{C}$ has multiple distinct named holes, then $\mathbb{C}\ [\overline{es}\ /\ \overline{hs}]$ is the operation of filling all of the holes with their corresponding expressions.

A *fold*, in the sense of fold/unfold reasoning [Burstall and Darlington, 1977], is the following operation:

$$\frac{(\overline{hs}, \mathbb{C}, \mathbb{D}) \qquad \mathbb{C}\ [\overline{es}\ /\ \overline{hs}]\ \equiv\ e}{e \Longrightarrow \mathbb{D}\ [\overline{es}\ /\ \overline{hs}]}\ \textsc{Fold}$$

That is, given an equality between $\mathbb{C}$ and $\mathbb{D}$ with holes $\overline{hs}$, if $\mathbb{C}$, with holes instantiated to expressions $\overline{es}$, is equivalent to expression e, then e can be rewritten to $\mathbb{D}$, instantiated to the same expressions $\overline{es}$.

4.4.2 Implementation

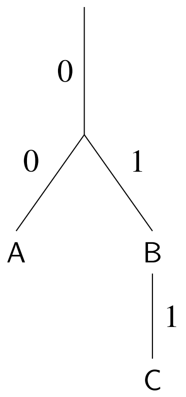
HERMIT's implementation of the fold operation is based on TrieMaps. TrieMaps are a well known means of mapping complex keys to values [Hinze, 2000], and are used by GHC itself for common sub-expression elimination and determining α-equivalence. The primary benefit, and thus motivation, of implementing fold using TrieMaps is that multiple patterns can be checked at once. This has dramatic performance implications for certain primitive operations in HERMIT.

This section develops HERMIT's implementation of TrieMaps using a small expression language as a running example. Beginning from tries, it illustrates GHC's implementation of TrieMaps. Then it extends the matching operation of GHC's TrieMap to patterns which contain holes. Finally, the TrieMap implementation is used to implement the fold operation. Only the lookup function of the TrieMap is presented. The insertion operation is entirely straightforward, but dense, providing no additional illumination beyond that gained from understanding lookup. It is left as an exercise to the reader.

4.4.2.1 Tries

A trie, or radix tree, is an efficient means for associating keys with values when the keys are finite strings. It is efficient in the sense that insertion and lookup operations are both linear in the length of the *key*, not the size of the trie. It is also space efficient because redundant prefixes of keys are only stored once.

A trie can be used to associate bit strings with values. For instance, the map $\{00 \Rightarrow \mathsf{A}, 01 \Rightarrow \mathsf{B}, 011 \Rightarrow \mathsf{C}\}$ can be represented by the following trie:



Looking up a string in the trie involves following the edges corresponding to the components of the string. For instance, to check if the bit string ‘01’ is in the trie above, the lookup operation begins at the root and follows the edge labeled ‘0’, then the edge labeled ‘1’, arriving at the node containing B, which is the value returned. If no value is associated with the node reached when the string is exhausted, the key is not in the map. For instance, looking up the bit string ‘0’ will end at a node with no value, so ‘0’ is not in the map.

4.4.2.2 TrieMaps

Rather than construct an explicit tree, a trie can be implemented as a map of maps, hence a *TrieMap*. Each level of the trie optionally has a value (if the empty string is in the map), as well as a map whose keys are single bits, and whose values are other tries. The lookup operation looks up each successive bit in the map returned by the lookup of the previous bit.

```
newtype BitTrie a = BTrie (Maybe a) (Map Bit (BitTrie a))
lookupBT :: [Bit] → BitTrie a → Maybe a
lookupBT []       (BTrie v _ ) = v
lookupBT (b : bs) (BTrie _ m)  = case lookup b m of
                                   Nothing → Nothing
                                   Just t → lookupBT bs t
```

The idea can be extended to any key which can represented by a finite structure. That is, rather than require the *value* of the key to be a string, the *structure* used to encode the value can be turned into a string which is appropriate to use as a key. To see how, consider using the following small expression type as a key:

```
data Expr = App Expr Expr | Var String
```

Using the idea that tries are maps of maps, each level of the trie for *Expr* is a map whose keys are *constructors* of *Expr* and whose values are other tries. A standard Haskell *Map* cannot be keyed on constructors, but observe that there are a finite number of constructors for any Haskell type, so an n-ary tuple (or record) will suffice.

```
data ExprTrie a = ETrie { etApp :: ExprTrie (ExprTrie a), etVar :: Map String a }
                | EEmpty
```

The ETrie constructor can be seen as a map with two possible keys, one for each constructor of *Expr*. Lookup is a matter of choosing between these keys based on whether the Var or App constructor was matched, then recursively looking up the components of the constructor.

```
lookupE :: Expr → ExprTrie a → Maybe a
lookupE _            EEmpty = Nothing
lookupE (Var s)      trie   = lookup s (etVar trie)
lookupE (App e1 e2) trie    = case lookupE e1 (etApp trie) of
                                Nothing    → Nothing
                                Just trie' → lookupE e2 trie'
```

Looking up a variable consists of looking up the variable's string in the map held in the *etVar* field. The interesting case is the one for *AppCon*. The *etApp* field contains a *ExprTrie* whose values are themselves *ExprTrie*s. The left subexpression is looked up in the outer *ExprTrie*, returning an inner *ExprTrie*, if present. The right subtree is then looked up in this inner *ExprTrie*. Intuitively, this corresponds to flattening the AST for the expression into a sequence of nodes using a pre-order depth-first traversal, then using the resulting sequence as a key to a trie.

This technique depends on two advanced features of Haskell's type system. Notice that *ExprTrie* is a *non-regular*, or *nested* datatype [Bird and Meertens, 1998]. A nested datatype is one where the recursive calls on the right-hand side of the data definition are substitution instances (not copies) of the left-hand side of the definition. In this case *ExprTrie* is nested because the *etApp* field has type *ExprTrie* (*ExprTrie a*). Any time a key has a constructor with more than one field, the corresponding trie will be a nested datatype.

Accordingly, functions which operate on nested datatypes, such as *lookupE*, require a non-regular, or polymorphic, form of recursion [Hallett and Kfoury, 2005]. In the right-hand side of the last case of $\mathit{lookupE}$, the two recursive calls are made at different types. The first $\mathit{lookupE}$ has type:

$$\mathit{Expr} \to \mathit{ExprTrie}\ (\mathit{ExprTrie}\ a) \to \mathit{Maybe}\ (\mathit{ExprTrie}\ a)$$

whereas the second has type $\mathit{Expr} \to \mathit{ExprTrie}\ a \to \mathit{Maybe}\ a$.

4.4.2.3 α-equivalence

Imagine adding abstraction to the Expr language.

```
data Expr = App Expr Expr
          | Var String
          | Lam String Expr
```

The $\mathit{ExprTrie}$ type and $\mathit{lookupE}$ function could be extended to handle Lam in a manner similiar to the handling of App. However, the resulting trie would only match keys that are strictly structurally equivalent. Languages like Expr usually have a notion of α-equivalence, where equivalence is defined modulo binding names. In such languages, the expression Lam "x" (Var "x") and Lam "y" (Var "y") are said to be α-equivalent, because they represent the same function, only differing in choice of binding name. It is natural when using Expr as a key that matching should occur up to α-equivalence.

This can be accomplished by distinguishing between free and bound vars when looking up variable occurrences. To do this, a new VarMap type is introduced. A VarMap is actually a pair of maps: one is keyed on the names of free variables, the other is keyed on the De Bruijn index of bound variables.

```
data VarMap a = VarMap {vmBound :: Map Int a, vmFree :: Map String a}
```

In order to distinguish free variables from bound variables, the lookup function for VarMaps requires a *renaming environment*, which is just a mapping from variable names to De Bruijn indices, and a supply of fresh indices.

```
data RenameEnv = RNEnv Int (Map String Int)
emptyEnv :: RenameEnv
emptyEnv = RNEnv 0 empty
extendEnv :: String → RenameEnv → RenameEnv
extendEnv s (RNEnv i m) = RNEnv (i + 1) (insert s i m)
lookupEnv :: String → RenameEnv → Maybe Int
lookupEnv s (RNEnv _ m) = lookup s m
```

The lookup function for *VarMap*s uses this renaming environment to determine whether a variable is bound, and thus which of its maps to look in.

```
lookupVM :: RenameEnv → String → VarMap a → Maybe a
lookupVM env s m = case lookupEnv s env of
                     Nothing → lookup s (vmFree m)
                     Just i  → lookup i (vmBound m)
```

With this in place, *ExprTrie* can be modified to use *VarMap* for the *etVar* field.

```
data ExprTrie a = ETrie { etApp :: ExprTrie (ExprTrie a)
                        , etVar :: VarMap a
                        , etLam :: ExprTrie a
                        }
                | EEmpty
```

Now, *lookupE* updates the renaming environment whenever a binding is encountered and the updated environment is used for looking up the body of the abstraction. Thus, any occurrences of the variable are now bound and De Bruijn indexed.

```
lookupE :: RenameEnv → Expr → ExprTrie a → Maybe a
lookupE _   _             EEmpty = Nothing
lookupE env (Var s)       trie   = lookupVM env s (etVar trie)
lookupE env (App e1 e2)   trie   = case lookupE env e1 (etApp trie) of
                                     Nothing   → Nothing
                                     Just trie' → lookupE env e2 trie'
lookupE env (Lam s e)     trie   = let env' = extendEnv s env
                                   in lookupE env' e (etLam trie)
```

4.4.2.4 Adding Holes

The TrieMaps described thus far are those implemented by GHC. Appropriate trie datatypes and insertion and lookup functions are defined for GHC Core's various types, including *CoreExpr*, *Type*, *CoreBind*, and *CoreAlt*. They are used to check expressions for α-equivalence by inserting one expression into the map, then looking up the other expression. In this case, the value associated with the key is irrelevant, it is only important that the keys match. They are also used for common sub-expression elimination. An expression is used as the key, and a binder which is bound to that expression as the value. Subsequent expressions are looked up in the TrieMap and, if found, are replaced with an occurrence of the associated binder.

HERMIT extends TrieMaps further, with the notion of *holes*. Holes can be thought of as meta-variables which match any expression. A key which features one or more holes is a pattern. A successful lookup operation returns a mapping from holes to matched expressions, in addition to the value associated with the key.

To be concrete, consider modifying *Expr* and *ExprTrie* to enable matching with holes. First the *Expr* type needs some notion of named hole occurrences. This could be a distinct constructor which carries the hole name.

```
data Expr = Hole String
          | ... as before ...
```

However, this new Hole constructor is isomorphic to the existing Var constructor, so it is also valid to say that certain variables will be treated as holes. This is what is done for GHC Core, since HERMIT wishes to use the GHC-defined *CoreExpr* type as keys to the map. With that in mind, this example will assume that no distinct Hole constructor is added to *Expr*.

The *ExprTrie* types *does* need to distinguish holes from regular variable occurrences. Since holes unify with expressions, a field for holes must be added to *ExprTrie*, not *VarMap*.

```
data ExprTrie a = ETrie { etApp  :: ExprTrie (ExprTrie a)
                        , etVar  :: VarMap a
                        , etLam  :: ExprTrie a
                        , etHole :: Map String a
                        }
                | EEmpty
```

When a variable is encountered during insertion, it is checked against the list of holes. If it is a hole, insertion proceeds with the *etHole* field. Otherwise, it is a normal variable and the insertion is made in the *etVar* field.

The *lookupE* function must be modified in two ways. First, it must accumulate a mapping from holes to expressions. Second, it must be capable of returning multiple values. The reason

```
lookupE :: Map String Expr → RenameEnv → Expr → ExprTrie a → [(Map String Expr, a)]
lookupE _ _ _ EEmpty = []
lookupE hs env expr trie    = hss ⧺ go expr
  where hss = [r | (h, v) ← toList (etHole trie)
                 , r      ← case lookup h hs of
                                Nothing → [(insert h expr hs, v)]
                                Just e  → [(hs, v) | exprAlphaEq e expr]
              ]
        go (Var s)        = case lookupVM env s (etVar trie) of
                                Nothing → []
                                Just v  → [(hs, v)]
        go (App e1 e2) = [ r | (hs', trie') ← lookupE hs env e1 (etApp trie)
                             , r            ← lookupE hs' env e2 trie']
        go (Lam s e)     = let env' = extendEnv s env
                           in lookupE hs env' e (etLam trie)
```

Figure 4.4: Lookup Function for *ExprTrie* with Holes.

for the latter change is that holes allow patterns to overlap. Consider inserting the following two expressions into the map, where variables `hole1` and `hole2` are holes.

```
App (Var "hole1") (Var "x")
App (Var "hole2") (Var "y")
```

The left-hand side of the applications overlap. Only the right-hand side differentiates the patterns. Looking up the expression App (Var "f") (Var "x") will begin by looking up (Var "f") in the *ExprTrie* for the left-hand side of the application. This will match both patterns, with both holes assigned the expression (Var "f") and two possible sub-maps suitable for looking up the right-hand side. Consequently (Var "x"), will need to be looked up in *both*. It will only be found in one of them. The final version of *lookupE* is found in Figure 4.4.

Notice in Figure 4.4 that *hss* attempts to assign the current expression to every hole at this point in the pattern. Recall that a hole may occur in a pattern multiple times, but the same expression must appear in all holes with the same name. If the hole has been previously assigned an expression, then the previous assignment and this expression are compared for α-equivalence. If the equivalence check fails, then the same hole has matched two different expressions, and the result is discarded. The other (non-hole) cases are largely as before, but instead of returning zero or one result with the $Maybe$ type, zero or more results may be returned.

4.4.2.5 Implementing Folds

The fold transformation defined in Section 4.4.1 can be implemented by building a TrieMap keyed on patterns, with the corresponding templates as values. To fold an expression, it is looked up in the map. The holes of the resulting template are filled with their matching expressions.

```
subst :: (String, Expr) → Expr → Expr
subst = ...
insertE :: [String] → RenameEnv → Expr → a → ExprTrie a
insertE = ...
fold :: [String] → Expr → Expr → Expr → Maybe Expr
fold holes pattern template expr =
  let  trie = insertE holes emptyEnv pattern template EEmpty
  in   case lookupE empty emptyEnv expr trie of
         [(hs, tmpl)] → Just (foldr subst tmpl (toList hs))
         _            → Nothing   -- no match, or ambiguous match
```

4.4.3 Applications

Many transformations in HERMIT are instances of the lookup operation of this TrieMap. As previously mentioned, folding an instance of a function body into a call to that function is one such transformation. For example, given the following definition of $mean$:

```
mean xs = sum xs `div` length xs
```

An equality is formed by using the function parameter xs as the hole, the right-hand side as the pattern, and the left-hand side as the template. A fold operation using this equality would transform the expression $sum\ ys$ ‘div‘ $length\ ys$ into $mean\ \ ys$. This is known as *folding* the definition in fold/unfold reasoning [Burstall and Darlington, 1977], and is the source of the ‘fold’ name.

Of course, swapping the pattern and template causes the transformation to act as an unfold. Thus, folds can be used to implement that transformation, which is extremely common in practice, as well.

Applying GHC rewrite rules is also a fold operation. Looking up a rewrite rule in GHC's internal state returns a set universal quantifiers and GHC Core expressions for the left-hand side and right-hand side of the rule. The quantifiers are used as holes, and either side can be the pattern and template, depending on which direction the rule is being applied. Thus, while GHC only applies its rewrite rules left-to-right, HERMIT can apply them in either direction.

Like GHC, HERMIT's TrieMaps can be used to determine α-equivalence. In that case, there are no holes. Folds are also used to implement several transformations necessary for proving properties in HERMIT. These are discussed in Sections 5.8 and 5.10.

4.5 Dictionary

HERMIT provides a substantial dictionary of rewrites and transformations for GHC Core. Some present capabilities already built into GHC with a more convenient user interface. Others use KURE to encode operations commonly used in pen-and-paper reasoning. Together, they serve as a general toolkit from which HERMIT users can construct their own transformations.

GHC's APIs for working with GHC Core are powerful and flexible, but not particularly safe. They allow a plugin author to construct any Core expression desired, including ones which do not typecheck! More subtly, GHC's optimizer maintains a number of invariants in Core expressions which are not enforced by the types. A significant challenge when writing a GHC plugin is to ensure that transformed or generated code is type safe and maintains these invariants.

HERMIT's primitive transformations provide significant value as the base of a modular and composable transformation language which maintains these invariants. This section surveys some transformations offered by HERMIT's library of transformations, refered to as 'the Dictionary'. The Dictionary contains hundreds more transformations not covered here. Though, in practice, the user only need remember the high-level ones, as they combine many simple transformations.

4.5.1 Fold/Unfold

Fundamental to semi-formal reasoning is the ability to fold [Burstall and Darlington, 1977] or unfold the definition of a function. HERMIT implements the fold rewrite by lifting a primitive fold operation implemented using TrieMaps (Section 4.4.2.5) into a KURE rewrite. In contrast, the unfold operation is implemented in terms of a primitive inlining transformation. Both operations have to ensure that variable capture does not occur. This is done using a *depth check*.

To perform the depth check, the depth of each binding is recorded in the context. The depth begins at zero, and grows as each parallel binding group is added to the context during descent. When a bound variable is folded or unfolded, its binding depth is compared with the binding depths of all of the free variables in the resulting expression. If any of the free variables was bound at a depth greater than the depth of the binder, then that free variable has been redefined since the unfolding was recorded for the binder. Replacing the binder with the resulting expression would lead to variable capture by this redefinition, so the transformation must fail.

As an alternative approach, for each binder added to the context, all unfoldings featuring that binder could be invalidated. Thus, unfoldings which would result in variable capture would simply not be available. The depth-based approach was taken instead for performance reasons. Comparing depths is less performance intensive than updating unfolding information for every binder in the context each time a new binder is added.

4.5.2 Local Transformations

HERMIT provides a large suite of local algebraic transformations of GHC Core expressions. These are the lowest-level primitives, implementing operations such as case reduction, let-floating, or the case-of-case transformation. Many of these transformations are taken from Santos [1995]. All of these transformations are careful to avoid pitfalls such as variable capture, or violations of the invariants of GHC Core, making them a useful base for constructing larger transformations.

4.5.3 Creating and Finding Variables

Recall that variables in GHC Core are unique names which have been annotated with their type (or kind) and other information, such as specialization rules or an unfolding. Creating local variables is relatively straightforward, requiring only the desired name and type, using the monadic name supply to generate a unique identifier. HERMIT supplies transformations for this purpose.

Creating non-local variables is more difficult, because the unique identifier of the resulting variable needs to match the unique identifier assigned to that entity previously in the same compilation session. GHC provides a name cache for this purpose, allowing variables to be looked up by $RdrName$. HERMIT lifts this capability into transformations which look up $HermitName$s (Section 4.3), which are more convenient for user-specification.

If the desired variable is from a module which has not been imported by the current module, the exports of that module will not have been loaded into GHC's name cache. Thus, looking up the variable will fail. This creates a fragile situation where the target program needs to explicitly import every module from which HERMIT might look up a variable, even if the source program does not depend on those modules.

To avoid this, HERMIT's lookup transformations provide a key extension to GHC's own lookup functions. GHC's dynamic loading facilities are used to dynamically load the module in question, adding its exports to the name cache, allowing lookup to succeed. This is akin to injecting a dependency on that particular module. GHC rewrite rules can be dynamically loaded in a similar manner.

The ability to dynamically inject module dependencies in this way has been found to be critical for domain-specific optimizations which rely on auxiliary GHC rewrite rules, or any data-refinement transformation which introduces a new data type to the program. This includes many Worker/Wrapper derivations (Section 9.1).

4.5.4 Constructing Expressions

GHC provides a library of smart constructors for building GHC Core expressions which respect the required invariants. Even with these functions, constructing expressions can be a tedious process, requiring careful thought to ensure all the explicit type and dictionary arguments are in their proper places. This is especially true when the types in question must be extracted from some existing expression.

To ease this burden, HERMIT's dictionary includes a handful of transformations for constructing GHC Core expressions. These transformations lift the GHC smart constructors for constructing expressions into KURE transformations, and additionally attempt to handle types automatically. The most primitive of these is $buildAppT$ which, given two expressions, builds an application. The domain type of the function expression is unified with the type of the argument expression by calling GHC's type unification functionality. From this, a type substitution is generated which can be used to specialize both expressions, making them suitable for pairing in an application.

Building on $buildAppT$, several transformations are provided which build commonly needed expressions, including applications of the id and fix functions and function composition, as well as applications to $undefined$. The latter is used when generating strictness lemmas. The ability to build function compositions is used by the Worker/Wrapper rewrites (Section 9.1) to generate appropriate pre-condition obligations.

4.5.5 Navigation

HERMIT, via KURE, uses sequences of crumbs to denote a path in the AST. Crumbs are also exposed by the Shell as a primitive means of navigation, allowing the user to descend into child nodes. Navigating solely with crumbs is both verbose and tedious, however. Scripts which rely heavily on crumbs for navigation tend to be extremely sensitive to changes in the program.

To offer navigation at a higher level of abstraction, HERMIT offers many transformations for generating paths in the tree. Like other transformations in HERMIT, path-finding transformations are defined using KURE.

$bindingOfT :: (Var \rightarrow Bool) \rightarrow TransformH\ CoreTC\ (Path\ Crumb)$

Path-finding transformations such as $bindingOfT$, which finds a path to the binding whose binder satisfies the given predicate, are usually defined in terms of KURE's generic path-finding functions. KURE has generic support for finding a single path, the shortest path, the longest path, or all paths that satisfy a given predicate. HERMIT's path-finding transformations are typically defined over the universe type, so that they generate valid paths when applied to any of the member types of the universe.

The paths generated by applying these transformations can be used to focus a subsequent transformation at the given path via one of KURE's focusing combinators. When used in the Shell, the resulting path is appended to the current focus, changing the focused expression.

4.5.6 Debugging

When developing large composite rewrites, it is often helpful to understand which of the component rewrites is succeeding, and when. If the overall composite rewrite simply fails, it is usually helpful to be able to see intermediate results, to understand where the rewrite is failing. This is especially true when the composite rewrite involves an iteration strategy, which may apply a large number of component rewrites in a non-obvious order.

It is also helpful to be able to typecheck the expressions generated by a rewrite. A subtle mistake in expression construction can lead to an ill-typed expression, which only becomes obvious when future rewrites that depend on correct type information behave strangely, or when GHC itself aborts compilation due to type errors.

To aid in debugging rewrites, HERMIT provides three helpful combinators. The first, $traceR$, is an identity rewrite which prints a given message and a count of the number of times the message has been printed as a side effect. The second, $observeR$, is an identity rewrite which prints a given message and the current expression as a side effect. The third, $bracketR$, calls $observeR$ before and after a given rewrite, but only if that rewrite succeeds.

$traceR :: String \to RewriteH\ a$
$observeR :: String \to RewriteH\ a$
$bracketR :: String \to RewriteH\ a \to RewriteH\ a$

Note that $traceR$ and $observeR$ never fail, so care must be used when including them in iteration strategies which require (eventual) failure to terminate. $bracketR$ succeeds only when its argument rewrite succeeds. Together, these combinators have been found useful for debugging large composite rewrites, especially in automated domain-specific optimization passes which may involve thousands of iterations of a particular strategy.

To check that expressions resulting from rewrites are well-typed, HERMIT lifts GHC's *Core Lint* [Hudak et al., 2007] functionality into KURE transformations. Core Lint is both a type checking pass and a lint checking pass for GHC Core. The lint checking pass checks for various invariants expected on the GHC Core program, such as correct liveness annotations on binders and variable occurrences.

HERMIT's lifted versions of Core Lint can be applied to expressions or the entire module. The latter pass detects more lint errors, as it has more information available. Internally, HERMIT uses these transformations as a sanity check, applying them after rewriting steps during proof, for instance. A special mode of the shell runs Core Lint on the module after every rewrite, which is useful for catching errors early. As no type inference is necessary, Core Lint is quite fast compared to type checking Haskell source. Even so, automatic linting is disabled by default for performance reasons.

4.5.7 Composite Transformations

When rewriting real Haskell programs, the key steps are often decisions about when and where to unfold a definition, or abstract an expression, or apply a GHC rewrite rule. In between are many simplification and reduction steps which are usually glossed over in pen-and-paper derivations. Rather than require the user to be explicit about all these steps, HERMIT provides rewrites which attempt to simplify automatically. This allows the user to focus on the key steps, and leave the tedious manipulation to HERMIT.

4.5.7.1 Simplify

The gentle simplification rewrite *simplifyR* attempts to clean up an expression without drastically altering it. It applies β-reduction, case reduction, dead-let elimination, and non-work-duplicating let-substitution anywhere they apply in the expression, to exhaustion. It also unfolds a limited set of Haskell functions which tend to be used by programs written in a point-free style. Currently, these are: (\$), ($\circ$), *id*, *flip*, *const*, *fst*, *snd*, *curry*, and *uncurry*. Unfolding these combinators often reduces expression size, leaving an expression which involves explicit abstraction and application, associated in a consistent way. Eliminating this syntactic noise generally makes it easier for subsequent rewrites to target the expression.

4.5.7.2 Smash and Bash

More aggressive than *simplifyR* are *smashR* and *bashR*. Both rewrites apply a wide range of local transformations which either evaluate or eliminate expressions in some way, or float let- and case-expressions outward. This includes an collection of rewrites for eliminating casts by floating and merging them. Generally, these rewrites are used interactively as a crude means of normalizing a term.

bashR is careful not to duplicate work, making it more suited to program transformation and domain-specific optimizations. *smashR* is less limited, freely duplicating expressions by inlining let bindings and performing other evaluation, and is intended for use during proof, where performance of the resulting expression is not a concern.

It is sometimes helpful to understand the sequence of rewrites applied by both *smashR* and *bashR*, so corresponding debugging versions are also defined. The debugging versions wrap each component rewrite in *bracketR*, generating a log of successful rewrites, with before and after expressions, as they happen. The debugging versions also check each intermediate result using Core Lint, to aide debugging HERMIT itself, so they tend to be slower.

Chapter 5

Proof

HERMIT is designed to support proving program properties and performing equational reasoning transformations. Properties may be stated directly by the user, or arise during transformation, capturing necessary pre-conditions for the transformation to succeed. HERMIT refers to such properties as *lemmas*.

This chapter presents the design of HERMIT's capabilities for proving lemmas. It starts with an example of proving a property interactively in order to give a flavor for proof in HERMIT. Subsequent sections consider of the design implications of HERMIT's lemmas in detail.

It is important to note that HERMIT does not encode proofs using a formal logic, as is done in a formal theorem prover. Rather, proof in HERMIT is akin to the systematic application of informal logic found in traditional mathematical proofs. Reasoning in such proofs is typically performed using natural (rather than formal, symbolic) language, meaning some ambiguity is involved. Likewise, HERMIT's notion of equivalence (Section 5.3) is defined in terms of available transformations which may have varying degrees of correctness guarantees. Nevertheless, this style of proof still offers a higher assurance of correctness than testing alone, falling somewhere on a spectrum between testing and formal proof. Throughout this dissertation, the term "proof" is used to refer to this notion of making a systematic argument about correctness using an informal logic.

```
module Tree where
data Tree a = Node (Tree a) a (Tree a)
            | Leaf a
treeMap :: (a → b) → Tree a → Tree b
treeMap f (Leaf x)        = Leaf (f x)
treeMap f (Node t1 x t2) = Node (treeMap f t1) (f x) (treeMap f t2)
{-# RULES "treeMapId" [~] treeMap id = id #-}
```

Figure 5.1: Tree.hs: Haskell Source for the Tree Example.

5.1 Example

Proofs, in HERMIT, proceed in the style of natural deduction [Smith, 2012]. The user rewrites a property until a primitive truth value is reached. This is done in the Shell, which has support for rewriting properties in addition to transforming the underlying GHC Core program.

To demonstrate HERMIT's interactive proof capabilities, this section seeks to prove a property about the implementation of the *treeMap* function in Figure 5.1. The property to be proven is the familiar first functor law, namely that mapping the identity function over the tree is itself an identity operation. As in the previous example in Section 4.1, the reader is encouraged to install HERMIT and follow along.

The property in question has been stated as a GHC rewrite rule (Section 2.2.3), named `treeMapId`. GHC rewrite rules are currently the primary means of stating properties. The rule will be parsed by GHC, along with the rest of the program, and made available to HERMIT. To begin the proof, we invoke HERMIT on the source file.

```
$ hermit Tree.hs
```

```
module main:Tree where
  treeMap :: ∀ a b . (a → b) → Tree a → Tree b
```

As before, HERMIT displays a summary of the module. In this case, we are not concerned with transforming the module itself. To begin the proof, we first need to instruct HERMIT to turn the `treeMapId` rule into a HERMIT lemma. HERMIT will create a lemma with the same name, which can then be proven.

```
hermit> rule-to-lemma treeMapId
hermit> prove-lemma treeMapId

Goal:
  ∀ △. treeMap ▲ ▲ (id ▲) ≡ id ▲
```

The `prove-lemma` command instructs the Shell to enter proof mode, where rewrites target the lemma we are trying to prove instead of the underlying GHC Core program. The goal in proof mode is to rewrite the lemma to a primitive truth value. Once this is accomplished, HERMIT will exit proof mode and mark the original version of the lemma as proven.

Once again, types are displayed as green symbols by HERMIT's default pretty-printer. For clarity, we will instruct HERMIT to hide them.

```
hermit> set-pp-type Omit

Goal:
  treeMap id ≡ id
```

This proof will require structural induction, but the rule (and thus the resulting lemma) was stated in a point-free style. In order to have a variable to induct on, we need to apply an extensionality rewrite to the lemma. The argument to `extensionality` is a name for the new universal quantifier. The type of the quantifier is inferred from the type of the expressions which make up the equivalence.

```
proof> extensionality 't

Goal:
  ∀ t. treeMap id t ≡ id t
```

In this case, the type of t is inferred to be $Tree\ a$, where a is a new universally quantified type variable that is currently hidden by the decision to not display types. Now we can perform structural induction on t.

```
proof> induction 't

Goal:
  (treeMap id undefined ≡ id undefined)
  ∧
  ((∀ a b c.
    (treeMap id a ≡ id a)
    ⇒
    ((treeMap id c ≡ id c)
     ⇒
     (treeMap id (Node a b c) ≡ id (Node a b c))))
   ∧
   (∀ a. treeMap id (Leaf a) ≡ id (Leaf a)))
```

Structural induction (Section 5.10.5) is not a special built-in proof technique. It is a transformation like any other, rewriting the lemma into a conjunction of the two base cases and the inductive case. Note that, due to the order of the constructors in the data definition of $Tree$ in the source file, the inductive case is the second case of the three in the conjuction. Two inductive hypotheses are generated for the inductive case (because the Node constructor has two components of type $Tree\ a$) and made available via implication. Implications in HERMIT lemmas are like non-recursive let-expressions in GHC Core programs. While focused on the consequent of the implication, the antecedent is assumed, and available as a rewrite; similar to how a let-binding is in-scope in the body of the let-expression, and available for inlining.

The two base cases will be easy to prove, so let us first focus on the inductive case. To do so, we navigate using crumbs, similiar to the way we navigate in GHC Core expressions. As before, the open brace ({) pushes the current focus on a stack, then each crumb changes the focus path. The semi-colon is a statement separator.

```
proof> { forall-body ; conj-rhs ; conj-lhs

Goal:
  ∀ a b c.
  (treeMap id a ≡ id a)
  ⇒
  ((treeMap id c ≡ id c)
   ⇒
   (treeMap id (Node a b c) ≡ id (Node a b c)))
```

We are now focused on the inductive case, which is comprised of two implications, one for each inductive hypothesis. To prove the implication requires rewriting the consequent until it is true, so we will once again use crumbs to navigate to it.

```
proof> forall-body ; consequent ; consequent

Assumed lemmas:
ind-hyp-0 (Built In)
  treeMap id a ≡ id a
ind-hyp-1 (Built In)
```

```
  treeMap id c ≡ id c
Goal:
  treeMap id (Node a b c) ≡ id (Node a b c)
```

Notice that by navigating into the consequent of the implications, the antecedents are in scope as assumed lemmas. The HERMIT Shell helpfully displays any in-scope local lemmas such as these above the goal.

Since *treeMap* and *id* are applied to explicit Node constructors, it makes sense to perform a bit of evaluation. Rather than do this step-by-step, we instruct HERMIT to unfold any function call, then apply the powerful `smash` rewrite to the result (Section 4.5.7.2).

```
proof> any-call unfold ; smash

Assumed lemmas:
ind-hyp-0 (Built In)
  treeMap id a ≡ id a
ind-hyp-1 (Built In)
  treeMap id c ≡ id c
Goal:
  Node (treeMap id a) b (treeMap id c) ≡ Node a b c
```

Notice that the first argument to the Node constructor on the left-hand side is an instance of the left-hand side of the `ind-hyp-0` lemma. We can use that lemma as a rewrite, applying it left-to-right (or 'forward').

```
proof> one-td (lemma-forward ind-hyp-0)

Assumed lemmas:
ind-hyp-0 (Built In)
  treeMap id a ≡ id a
ind-hyp-1 (Built In)
  treeMap id c ≡ id c
Goal:
  Node (id a) b (treeMap id c) ≡ Node a b c
```

The same can be done for the third argument to Node, using the other inductive hypothesis.

```
proof> one-td (lemma-forward ind-hyp-1)

Assumed lemmas:
ind-hyp-0 (Built In)
  treeMap id a ≡ id a
ind-hyp-1 (Built In)
  treeMap id c ≡ id c
Goal:
  Node (id a) b (id c) ≡ Node a b c
```

It is obvious that the two sides are equivalent at this point. We could manually unfold the calls to *id* and invoke HERMIT's `reflexivity` command to rewrite the entire equality to true, but we will call `smash` eventually, which will do this for us. Instead, we will pop the scope using (}) to return to the top of the lemma.

```
proof> }

Goal:
  (treeMap id undefined ≡ id undefined)
  ∧
  ((∀ a b c.
    (treeMap id a ≡ id a)
    ⇒
    ((treeMap id c ≡ id c) ⇒ (Node (id a) b (id c) ≡ Node a b c)))
   ∧
   (∀ a. treeMap id (Leaf a) ≡ id (Leaf a)))
```

The base cases are now in view again. They are simple to prove, only requiring us to unfold the calls to *treeMap* and *id* and `smash` the result.

```
proof> any-call unfold ; smash

Goal:
  true
```

In fact, `smash` has rewritten the entire lemma to the primitive truth value. This is because `smash` includes `reflexivity` as one of its rewrites, along with a host of lemma simplification rewrites which apply the usual boolean identity laws (Section 5.10.2). All that remains is to end the proof.

```
proof> end-proof

Successfully proven: treeMapId
```

HERMIT now marks the lemma as proven, meaning it can be used as a bi-directional rewrite, or as an auxiliary lemma during another proof. To display a list of available lemmas, we can use the `show-lemmas` command.

```
hermit> show-lemmas

treeMapId (Proven)
  treeMap id ≡ id
```

5.2 Lemmas

The remainder of this chapter is concerned with detailing the design of HERMIT's proof capabilities. This discussion primarily centers around the *Lemma* type, and operations defined on lemmas. A lemma is principally a clause, along with some status information indicating whether the lemma has been proven and whether it has been used.

```
data Lemma = Lemma Clause Proven Used
```

The *Clause* type encodes the actual property which the lemma embodies. *Primitive* clauses are CTrue, the primitive truth clause, and Equiv, which states an equivalence between two GHC Core expressions. *Composite* clauses combine other clauses via conjuction, disjunction, or implication. Implication clauses carry a lemma name which allows transformations to refer to the antecedent when it is in scope (Section 5.6). Any clause may reference universally quantified variables which are introduced by an enclosing Forall clause.

```
data Clause = CTrue                                   -- truth
            | Equiv CoreExpr     CoreExpr             -- alpha-equivalence
            | Conj  Clause       Clause               -- conjunction
            | Disj  Clause       Clause               -- disjunction
            | Impl  LemmaName    Clause Clause        -- implication
            | Forall [CoreBndr]  Clause               -- quantification
```

An example of a primitive lemma is the map fusion law:

$$\forall\ a\ b\ c\ (f :: b \rightarrow c)\ (g :: a \rightarrow b).\ \ \mathit{map}\ f \circ \mathit{map}\ g\ \equiv\ \mathit{map}\ (f \circ g)$$

This lemma is an equivalence between the expressions $\mathit{map}\ f \circ \mathit{map}\ g$ and $\mathit{map}\ (f \circ g)$, along with the universal quantifiers a, b, c, f, and g. The quantifiers a, b, and c quantify *types*, whereas f and g quantify values. Quantifiers scope over other quantifiers to their right, just as lambda-binders do, allowing b to appear in the type of f, for instance. Note that occurrences of the type quantifiers have been elided in the two expressions, for clarity.

An example of a composite lemma is the foldr fusion law:

$$\begin{array}{l}
\forall\ f\ g\ h\ a\ b. \\
(f\ \mathit{undefined}\ \equiv\ \mathit{undefined})\ \wedge\ (f\ a\ \equiv\ b)\ \wedge\ (\forall\ x\ y.\ f\ (g\ x\ y)\ \equiv\ h\ x\ (f\ y)) \\
\ \Rightarrow \\
f \circ \mathit{foldr}\ g\ a\ \equiv\ \mathit{foldr}\ h\ b
\end{array}$$

This lemma is an implication whose antecedent is a conjunction of three clauses. The clause $f\ \mathit{undefined} = \mathit{undefined}$ states that f is expected to be strict. Note the third clause in the conjuction, which has its own quantifiers which scope only over that particular clause. Again, the type quantifiers are elided for clarity.

The proven and used status of a given lemma is encoded by the following two types.

$\mathbf{data}\ \mathit{Proven} = \mathsf{NotProven} \mid \mathsf{Assumed} \mid \mathsf{BuiltIn} \mid \mathsf{Proven}$
$\mathbf{data}\ \mathit{Used} = \mathsf{Obligation} \mid \mathsf{UnsafeUsed} \mid \mathsf{NotUsed}$

Lemmas marked Assumed have been assumed by the user, whereas BuiltIn lemmas are assumed by a lemma library (Section 5.9), or by HERMIT itself. This distinction is important mainly to the Shell, which has different safety modes that may warn about or disallow assumed lemmas.

The Obligation and UnsafeUsed tags both indicate a lemma has been used, and that the validity of the current transformation or proof depends on the validity of the lemma. Again, the distinction is for the benefit of the Shell. By default, all lemmas are marked Obligation when they are used. Special unsafe commands in the Shell use the UnsafeUsed status, allowing proofs to be put off, or ignored altogether. Such unsafe commands are only available when HERMIT is explicitly run in unsafe mode, but are useful for rapid exploration of a transformation.

5.3 Equivalence

As HERMIT's lemmas are concerned with stating equivalences, it is important to be clear about the notion of equivalence. Two GHC Core expressions are considered to be equal if one can be transformed into the other, modulo α-equality. Thus equivalence is dependent on the set of primitive transformations that HERMIT provides, which include transformations that can change the strictness properties of a program. Consequently, HERMIT's equivalence relation only guarantees *partial correctness*: the output produced by a program after transformation can be either more or less defined, but when the output is defined it does not differ in value.

Partial correctness in this sense is still valuable, however. HERMIT is not intended to supplant automated formal theorem provers. Instead if seeks to improve the currently common practice of semi-formal reasoning by mechanizing it. In that light, even this partial correctness represents an increase in formality. Some commonly used transformations already introduce their strictness conditions as lemma obligations (Section 5.7). Future work on classifying HERMIT's transformations with a full accounting of pre-conditions and correctness criteria could strengthen the equivalence guarantee.

Similiarly, two clauses are equivalent if one can be transformed into the other, modulo α-equality. A proof in HERMIT consists of demonstrating a clause is equivalent to the primitive CTrue clause. The validity of a given transformation depends on the context in which is used. Intuitively, proving a specific case does not prove the general case. Thus, a transformation which weakens a clause by making it more specific is not valid during proof. However, a transformation which strengthens a clause by making it more abstract is valid. Conversely, it is not valid to apply a strengthening transformation to an already proven clause, but it is valid to apply a weakening transformation.

As the current primary means of proving lemmas is via the HERMIT's Shell, validity is ensured by selectively enabling or disabling certain transformations during proof. Fully classifying transformations as strengthening/weakening and enforcing validity at the KURE level remains future work.

5.4 Creating Lemmas

There are two methods of introducing lemmas into HERMIT. The first method is to state the lemma as a GHC RULES pragma, then convert it to lemma using the *ruleToLemmaT* transformation, which inserts the resulting *Lemma* into HERMIT's lemma store.

GHC RULES are convenient because they can be stated in Haskell syntax, rather than GHC Core. They are parsed, type-checked, and desugared into GHC Core by GHC's front end, allowing HERMIT to reuse these front-end capabilities. Most Haskell programmers will be familiar with RULES, as they are commonly used to influence the behavior of GHC's optimizer. As an example, the map fusion lemma can be stated as the following rule:

{-# RULES "map-fusion" [~] **forall** $f\ g.\ map\ f \circ map\ g = map\ (f \circ g)$ #-}

As GHC rules are intended to alter the behavior of the optimizer, they carry a *phase annotation*, directing GHC to only apply the rule during a specific phase of simplification. This rule is intended only for use as a HERMIT lemma, so it has been given the special `[~]` annotation, indicating it is never active during optimization.

GHC rules are equivalences between expressions, so they can only be used to specify primitive equivalence lemmas. Additionally, GHC imposes syntactic restrictions on the expressions in the pragma. One of these restrictions is that the head of each side of the rule must be a function application of an in-scope function, meaning only a subset of all possible primitive lemmas can be expressed. However, in practice, many useful properties can be specified using RULES, including most of the familiar type-class laws, which are the target of the case study in Chapter 6.

RULES pragmas intended as lemmas can be specified in the target program as inactive rules, or they may be contained in another module or package entirely. If the $\mathit{RuleName}$ argument to $\mathit{ruleToLemmaT}$ includes a fully-qualified module name, that module will be dynamically loaded in order to find the rule, even if the target program does not depend on it. When dynamically loading in this fashion, no rules are actually introduced into the target program.

Many interesting properties, such as $foldr$'s fusion law, cannot be specified as GHC rules. One means of constructing these properties is to merge separate rule-based lemmas. While HERMIT does supply transformations for this task, it can be tedious. A more direct solution is to construct the desired lemma entirely in KURE, using HERMIT's combinators for finding names (Section 4.5.3) and constructing expressions (Section 4.5.4). A lemma constructed this way can then be used by dynamically loading it from a lemma library (Section 5.9).

Ideally, GHC's parser could be modified to parse composite lemmas directly, or HERMIT itself could implement a parser for GHC Core and allow lemmas to be specified via the Shell or in scripts. In practice, both are likely useful, as some properties may require more explicit control of the resulting GHC Core than a Haskell-source-to-Core translation would allow. These capabilities are left as future work.

5.5 Primitive Operations

HERMIT implements a number of primitive operations on the Clause type which parallel those available for GHC Core expressions. These include capture-avoiding substitution, syntactic and α-equality, free variable calculation, redundant binder elimination, lint/type checking, unshadowing, and instantiation. This section presents redundant binder elimination and instantiation in detail, as both are specific to lemmas.

5.5.1 Redundant Binder Elimination

When lemmas are created from GHC RULES pragmas, constraints in the types of the expressions appear as explicit dictionary binders. For instance, the following rule states the bind associativity law expected of any monad instance.

{-# RULES “bind-assoc” $[\sim]$ **forall** $j\ k\ l.\ (j \gg\!= k) \gg\!= l = j \gg\!= (\lambda x \rightarrow k\ x \gg\!= l)$ #-}

Once desugared into GHC Core, the following property is given to HERMIT for conversion into a lemma. Note the explicit type binders (m, b, a, and c) and dictionary binders ($\$\mathit{dMonad1}$ and $\$\mathit{dMonad2}$).

$$
\begin{array}{l}
\forall\ m\ b\ a\ c\ \$\mathit{dMonad1}\ \$\mathit{dMonad2}\ j\ k\ l.\\
(\gg\!=)\ m\ \$\mathit{dMonad1}\ \ c\ b\ ((\gg\!=)\ m\ \$\mathit{dMonad2}\ \ a\ c\ j\ k)\ l\\
\ =\\
(\gg\!=)\ m\ \$\mathit{dMonad2}\ \ a\ b\ j\ (\lambda x \rightarrow (\gg\!=)\ m\ \$\mathit{dMonad2}\ \ c\ b\ (k\ x)\ l)
\end{array}
$$

Here, the class method $\gg\!=$ has been desugared into a selector function which is applied to the monad type m and a dictionary for that monad $\$\mathit{dMonad1}$ (or $\$\mathit{dMonad2}$). This selector function projects the actual implementation of $\gg\!=$ for this specific monad from the dictionary. The implementation function will be applied to the remaining arguments.

Inspection of the types of $\$\mathit{dMonad1}$ and $\$\mathit{dMonad2}$ will reveal they both have type $Monad\ m$. GHC’s typechecker guarantees that only one instance for a given type is in scope at a time, so these binders will necessarily always have the same concrete dictionary implementation. Two such binders are created in order that the rewrite rule liberally matches two different expressions which will eventually evaluate to the same dictionary.

However, having two such binders as universal quantifiers in a lemma can be problematic. Mildly inconvenient, they must both be instantiated separately when proving, even though they will get the same concrete value. More seriously, *occurrences* of the two binders are considered distinct, because the two binders are in fact unique. The fact that they must have the same concrete value eventually is not evident to GHC or HERMIT.

This is problematic when transforming the lemma itself because it prevents a rewrite which expects identical expressions for the dictionaries from matching. Consider transforming the left-hand side of the `bind-assoc` lemma above with the following hypothetical rewrite.

$$\begin{array}{l}\forall\ m\ a\ b\ c\ \$dMonad\ j\ k\ l\,.\\ (\gg\!=)\ m\ \$dMonad\ c\ a\ ((\gg\!=)\ m\ \$dMonad\ b\ c\ j\ k)\ l = foo\ m\ \$dMonad\ a\ b\ c\ j\ k\ l\end{array}$$

This rewrite is only parameterized over a single $\$dMonad$ dictionary, and thus, to match, the dictionary arguments to the two $\gg\!=$s must be the same. However, in `bind-assoc`, they are not! In short, redundant binders are never a problem when *applying* a lemma as a rewrite, but are problematic when rewriting the lemma itself (and lemmas must be rewritten to be proven).

In order to avoid this, redundant dictionary binders are eliminated at lemma construction. This is accomplished by substituting the first binder of a given dictionary type for any subsequent binders of the same type. Nominally, this means lemmas will successfully apply less often than rules, because the two dictionary expressions are required to be structurally equivalent expressions. In practice, this is rare, and can be worked around by using HERMIT to transform the two dictionary expressions to make them equivalent before applying the lemma.

5.5.2 Instantiation

Whereas substitution is concerned with substituting for a free variable occurrence in a clause, instantiation is more akin to β-reduction, weakening a clause by making the value of a universal quantifier concrete. However, unlike β-reduction, HERMIT allows quantifiers to be instantiated even if they are under other quantifiers, meaning the instantiation of a given quantifier may specialize the quantifers above it.

For instance, consider the following clause, stating a polymorphic identity property.

$\forall\ t\ (x :: t)\,.\ \ id\ t\ x\ \equiv\ x$

Instantiating x to Just `"Hello"` fully determines the type of t to be $Maybe\ String$. Thus, the resulting clause would be:

$id\ (Maybe\ String)\ (\mathsf{Just}$ `"Hello"`$)\ \equiv\ \mathsf{Just}$ `"Hello"`

More subtly, instantiating a quantifier may introduce new quantifiers. If, for instance, x was instantiated to the constructor Just, which has type $Maybe\ a$, the following clause would result:

$\forall\ a\,.\ \ id\ (Maybe\ a)\ \mathsf{Just}\ \equiv\ \mathsf{Just}$

It is also possible for the quantifier's type to specialize the type of the expression for which it is being instantiated. Consider instantiating the following clause which states that f is monotonic.

$\forall\ (f :: Int \to Int)\ x\ y\,.\ \ (x < y\ \equiv\ \mathsf{True})\ \Rightarrow\ (f\ x <= f\ y\ \equiv\ \mathsf{True})$

Instantiating f to the identity function $id :: a \to a$ will specialize the type of id to $Int \to Int$.

Type specialization is accomplished by GHC's built-in type (or kind) unification. The type (or kind) of the quantifier and the expression to which it is being instantiated are unified, giving a type substitution which can be applied to both. New quantifiers are introduced by free variables in the instantiation expression which results from this substitution.

In general, instantiation is a weakening transformation, and thus not valid during proof. It is primarily used to make a weakened instance of a lemma which can then be proved. For example, a type class law can be stated as a lemma, but can only be proved for specific instances of the class. Obtaining the weakened lemma for the specific instance is accomplished by instantiating it to a specific type.

Note, however, that instantiating *dictionaries* does not weaken or strengthen a clause because the typechecker guarantees that only one dictionary instance per type is in scope. The fact that HERMIT runs after the typechecker means it can rely on this guarantee. Thus, dictionary instantiation is allowed during proof.

5.6 Lemma Universes

HERMIT defines two universe types for clauses which extend the *Core* and *CoreTC* universes defined in Section 4.2.1. The first, *LCore*, only descends into nodes which contain GHC Core expressions or clauses. The second, *LCoreTC*, additionally descends into nodes which contain types and coercions.

```
data LCore = LClause Clause | LCore Core
data LCoreTC = LTCCore LCore | LTCTyCo TyCo
```

The rest of this section will present *LCore* in detail. *LCoreTC* is entirely analogous.

Injection instances to *LCore* are provided for *Clause* and all the component types of *Core*, including *CoreExpr*. The *Walker* instance is defined in terms of congruence combinators for the constructors of the *Clause* type, as is conventional in KURE.

```
instance (AddBindings c, ExtendPath c Crumb
         , LemmaContext c, ReadPath c Crumb) ⇒ Walker c LCore where
  allR :: ∀ m. MonadCatch m ⇒ Rewrite c m LCore → Rewrite c m LCore
  allR r = prefixFailMsg "allR failed: " $
    rewrite $ λc e → case e of
                      LClause cl  → inject <$> applyT allRclause c cl
                      LCore core → inject <$> applyT (allR $ extractR r) c core
    where
      allRclause :: MonadCatch m ⇒ Rewrite c m Clause
      allRclause =  forallR idR (extractR r)
                 <+ conjAllR (extractR r) (extractR r)
                 <+ disjAllR  (extractR r) (extractR r)
                 <+ implAllR (extractR r) (extractR r)
                 <+ equivAllR (extractR r) (extractR r)
                 <+ ctrueR
```

This definition takes advantage of the fact that clauses do not occur inside GHC Core expressions, so once the LCore constructor is encountered, traversal can be entirely delegated to the *allR* defined by the *Walker* instance for the *Core* universe.

The only notable difference from the *Walker* instance for the *Core* universe is the *LemmaContext* constraint on the context. This constraint specifies that the context can accumulate *local lemmas* during traversals. This is similiar to how bindings are accumulated using the *AddBindings* class.

```
class LemmaContext c where
  addAntecedent :: LemmaName → Lemma → c → c
  getAntecedents :: c → Map LemmaName Lemma
```

```
implT :: (ExtendPath c Crumb, LemmaContext c, Monad m)
        ⇒ Transform c m Clause a1
        → Transform c m Clause a2
        → (LemmaName → a1 → a2 → b)
        → Transform c m Clause b
implT t1 t2 f =
  transform $ λc cl → case cl of
    Impl nm ante con →
      let l = Lemma ante BuiltIn NotUsed
      in f nm <$> applyT t1 (c @@ Impl_Lhs) ante
              ⊛ applyT t2 (addAntecedent nm l c @@ Impl_Rhs) con
    _ → fail "not an implication."
implAllR :: (ExtendPath c Crumb, LemmaContext c, Monad m)
           ⇒ Rewrite c m Clause → Rewrite c m Clause → Rewrite c m Clause
implAllR r1 r2 = implT r1 r2 Impl
```

Figure 5.2: Congruence Combinators for Implication *Clause*s.

The congruence combinators for the Impl constructor in Figure 5.2 make use of *addAntecedent* to bring the antecedent into scope as a local lemma while traversing the consequent. Accumulating local lemmas in this manner is key to proving implication lemmas in HERMIT. Local lemmas in scope can be used as rewrites like any other lemma. This is used most notably by HERMIT's structural induction scheme, where the induction hypothesis is the antecedent to the clause representing the inductive case (Section 5.10.5).

5.7 Pre-conditions

HERMIT lemmas offer a convenient means for transformations to record necessary pre-conditions. In general, HERMIT transformations do not require pre-conditions to be proven first. Instead, they are recorded as an unproven lemma obligation. These obligations can then be proven after the fact.

Consider, for example, the transformation which floats a case expression from its position as an argument to a function.

```
f (case scrut of    ⟹    case scrut of
     alts → rhs)                alts → f rhs
```

This transformation is only valid if *f* is strict in its argument. Otherwise, it may alter the termination properties of the program by evaluating *scrut* more often.

The transformation can be idiomatically defined using KURE to perform both the rewrite itself and introduce the strictness condition as an unproven lemma.

```
caseFloatArgR :: LemmaName → RewriteH CoreExpr
caseFloatArgR nm = do
  App f (Case s b ty alts) ← idR
  r ← ...   -- construct the actual result, checking for capture, etc.
  clause ← buildStrictnessT f
  verifyOrCreateT nm $ Lemma clause NotProven Obligation
  return r
```

In HERMIT, as a general pattern, transformations which introduce lemmas for pre-conditions accept a lemma name to assign to the generated lemma. This particular transformation makes use of an auxiliary transformation *buildStrictnessT* which constructs the expression *f undefined* = *undefined* at the proper types. It then uses the *verifyOrCreateT* transformation to either introduce or discharge the obligation.

verifyOrCreateT first attempts to find a lemma with the given name. If found, the lemma is compared against the generated obligation. If the existing lemma can be used to prove (Section 5.8) the generated obligation, then the obligation is discarded. If it cannot, or no lemma with the given name exists, the obligation is recorded with the given name.

This design has two main benefits. First, not requiring proof of pre-conditions up-front allows larger transformations to be more easily constructed from smaller transformations which have pre-conditions. If the proof was required up-front, a large transformation would need a proof for every one of its component transformations up-front, making its use unwieldy. Instead, a large transformation ends up generating several pre-condition lemmas as appropriate.

Second, the use of *verifyOrCreateT* avoids unnecessary duplication of proof effort. In many cases, a single general property can be proved once, then used to discharge several pre-conditions which are instances of the general property.

5.8 Lemma Strength

TrieMaps (Section 4.4) are defined for the $Clause$ type, meaning clauses can be folded in a manner similar to expressions. There is no notion of a variable at the clause level, and clauses do not appear within GHC Core expressions, so clauses cannot unify with holes. The resulting fold requires the clause structure of the pattern and target to be identical, modulo the antecedent names, which are discarded during matching. (As antecedent names are only bound and do not occur, the fold does not even have to ensure they are α-equivalent.) Expressions within Equiv clauses are folded using the TrieMaps defined for expressions, determining the value of any holes in the same way as a regular expression fold.

The ability to fold clauses in this way leads to a natural means of defining relative strength of clauses. A clause `D` is weaker than clause `C` if `C` can be used as a pattern to fold `D`, using `C`'s quantifiers as holes. Intuitively this follows from the idea that instantiation is a weakening transformation. If `C` can be successfully used to fold `D`, that is the same as saying that `C` can be transformed into `D` by exclusively using a series of weakening instantiations of `C`'s quantifiers. Thus, `C` is stronger than `D`, and a proven `C` subsumes the proof of `D`.

5.9 Lemma Libraries

HERMIT lemmas may be packaged up into a library, allowing for sharing and reuse. HERMIT exports the following type:

$\mathbf{type}\ LemmaLibrary = TransformH\ ()\ (Map\ LemmaName\ Lemma)$

A lemma library is a normal Haskell module which exports one or more top-level bindings with the type $LemmaLibrary$. Such a Haskell module can be packaged using Cabal and distributed independently of HERMIT itself. HERMIT provides a primitive transformation which dynamically loads the desired library, applies it in the current context, and inserts the lemmas defined by the library into the Kernel's lemma store.

$loadLemmaLibraryT :: HermitName \rightarrow TransformH\ a\ ()$

The *HermitName* argument should be the fully-qualified name of a binding with the type *LemmaLibrary*. For instance, given the following library module:

```
module HERMIT.FooLibrary where
lemmas :: LemmaLibrary
lemmas = do ...
```

The library can be loaded using:

```
loadLemmaLibraryT "HERMIT.FooLibrary.lemmas"
```

The dynamic loading of the library is done using GHC's built-in dynamic loading capabilites, meaning the target program does not need to depend on the library in any way. As the *LemmaLibrary* type synonym indicates, lemma libraries are themselves transformations, so they have access to the current context and GHC state when loaded. Thus, library definitions can potentially be quite sophisticated and context-dependent. GHC's Core Lint (Section 4.5.6) is applied to lemmas returned by the library before insertion into the lemma store.

5.10 Lemma Dictionary

Proof, in HERMIT, is the process of rewriting a lemma's clause until it is the primitive CTrue clause. This demonstrates that the lemma is equivalent to truth by a series of equational transformations. To facilitate this, HERMIT's dictionary includes a number of useful rewrites over clauses. Additionally, proven lemmas are an important source of rewrites for both expressions and clauses.

This section highlights the most interesting transformations in the dictionary which involve lemmas. The full dictionary contains many more transformations not listed here which do things such as looking up lemmas by name, marking them proved, pretty-printing them, etc. The intent of this section is to give a sense of HERMIT's capabilities regarding lemmas.

5.10.1 Lemmas As Rewrites

A proven lemma is itself a useful rewrite. A primitive lemma can be used to rewrite an expression by folding the expression using either side of the lemma as the fold pattern, and the quantifiers of the lemma as holes. The result of the fold is the other side of the lemma, with holes instantiated to their matching expressions. This is accomplished using the rewrites *lemmaForwardR* and *lemmaBackwardR*, which apply the lemma left-to-right and right-to-left, respectively.

$$\begin{aligned}&\mathit{lemmaForwardR} &&:: \mathit{LemmaName} \to \mathit{RewriteH}\ \mathit{CoreExpr}\\ &\mathit{lemmaBackwardR} &&:: \mathit{LemmaName} \to \mathit{RewriteH}\ \mathit{CoreExpr}\end{aligned}$$

A lemma can also be used to rewrite another lemma. For instance, in the course of proving a lemma by rewriting it, the user may be faced with a clause which is an instance of an already proven lemma. HERMIT's *lemmaR* rewrite allows the clause to be rewritten directly to truth.

$$\mathit{lemmaR} :: \mathit{LemmaName} \to \mathit{RewriteH}\ \mathit{Clause}$$

This rewrite works by using the named lemma as a fold pattern, with any top-level quantifiers as holes. If the targeted clause can be folded by the pattern, it is considered a more specific instance of the named lemma (Section 5.8), and thus true if the named lemma is true. This is the capability that allows HERMIT proofs to be modular and reusable.

The *lemmaR* rewrite is actually a special case of the more general *lemmaConsequentR*.

$$\mathit{lemmaConsequentR} :: \mathit{LemmaName} \to \mathit{RewriteH}\ \mathit{Clause}$$

This rewrite requires the named lemma to be an implication[1]. The consequent of the implication is used as a fold pattern, with the top-level quantifiers (those scoping over the entire implication) as holes. The result is the antecedent, instantiated with expressions which matched the holes.

As an example of *lemmaConsequentR* in action, consider trying to prove the following lemma, where f and g are concrete functions.

$$\mathit{filter}\ f \ \equiv\ \mathit{filter}\ f \circ \mathit{filter}\ g$$

Rather than proceeding directly, one could appeal to the following more general lemma, which has been proven separately.

```
filter-split
```

$$\forall\, p\ q\,.\ (\forall\, x\,.\ (q\ x\ \equiv\ \mathsf{False})\ \Rightarrow\ (p\ x\ \equiv\ \mathsf{False}))\ \Rightarrow\ (\mathit{filter}\ p\ \equiv\ \mathit{filter}\ p \circ \mathit{filter}\ q)$$

[1] Any clause cl can be turned into $\mathit{true}\ \Rightarrow\ cl$, thus *lemmaR* is a special case of *lemmaConsequentR*.

Using *lemmaConsequentR*, the concrete instance being proved can be folded using the consequent of `filter-split`. The result of the rewrite is the antecedent of `filter-split`, instantiated to the expressions which matched the holes in the consequent.

$\forall\, x\,.\ (g\ x\ \equiv\ \mathsf{False})\ \Rightarrow\ (f\ x\ \equiv\ \mathsf{False})$

Proof proceeds by proving this instantiated antecedent. This capability allows general properties such as `filter-split` to be used to prove specific instances of the property.

5.10.2 Simplification

Two important simplification rewrites are provided for clauses. The first is *reflexivityR*, which checks an Equiv clause for α-equivalence between its expressions, replacing it with a primitive truth clause. As a matter of convenience, reflexivity is applied automatically by the `end-proof` command in the Shell. It is also included in `smash`.

If a rewrite like *reflexivityR* or *lemmaR* introduce a primitive truth clause in part of a larger composite clause, it is important that it can be simplified away. A rewrite for this purpose, called *clauseIdentitiesR* implements the following standard boolean identity laws.

$$\begin{array}{ll}
\mathit{true}\ \wedge\ c & \equiv\ c \\
c\ \wedge\ \mathit{true} & \equiv\ c \\
\mathit{true}\ \vee\ c & \equiv\ \mathit{true} \\
c\ \vee\ \mathit{true} & \equiv\ \mathit{true} \\
\mathit{true}\ \Rightarrow\ c & \equiv\ c \\
c\ \Rightarrow\ \mathit{true} & \equiv\ \mathit{true} \\
\forall\,\overline{vs}\,.\ \mathit{true} & \equiv\ \mathit{true}
\end{array}$$

These simplifications are also included in `smash`. In cases where the user wishes to simplify the clause structure without affecting the expressions, HERMIT provides the *simplifyClauseR* rewrite, which iteratively applies reflexivity and these boolean identity simplifications to exhaustion.

5.10.3 Instantiation

Instantiating the universal quantifiers of a lemma is an important operation when proving a specific instance of a general property. This is central to the case study in Chaper 6, where a property for each type class law is stated in terms of the class, then instantiated to a specific instance of the class, and proved for that instance. HERMIT provides two means of instantiating lemmas.

The first instantiation rewrite is $\mathit{instClauseR}$, which allows the user to select an arbitrary quantifier and replace it with an expression (or type) whose type (or kind) unifies with that of the quantifier. This is the type of instantiation discussed in Section 5.5.2, lifted into a rewrite over clauses. Since instantiation is a weakening transformation, it is not allowed during proof.

The second instantiation rewrite is $\mathit{instDictionariesR}$, which automatically instantiates any dictionary binders which have concrete types. To obtain expressions for the dictionaries, GHC's typechecker and desugarer are invoked. Since the typechecker guarantees that only one class instance for a given type is in-scope, dictionary instantiation is considered non-weakening, and is allowed during proof.

5.10.4 Strengthening

It is also possible to strengthen, or increase the generality of, a lemma. HERMIT provides a rewrite called $\mathit{abstractForallR}$ which adds a universal quantifier to the lemma, then folds all occurrences of a given expression into occurrences of this quantifier. This operation is the dual to instantiation, and thus is allowed during proof. If a lemma is strengthened outside of proof, its proven status is reset.

5.10.5 Structural Induction

Haskell programs usually contain recursive functions defined over (co)inductive data types. Proving even simple properties of such programs often requires the use of an induction principle. For example, while $[\,] +\!\!+\ xs \ \equiv \ xs$ can be proved simply by unfolding the definition of $+\!\!+$, proving the similar property $xs +\!\!+ [\,] \ \equiv \ xs$ requires reasoning inductively about the structure of xs.

To formalize HERMIT's notion of structural induction requires some notation: $\overline{vs}$ denotes a sequence of variables. $\forall(\mathsf{C}\ \overline{vs} :: \mathsf{A})$ quantifies over all constructors C of the algebraic data type A, fully applied to a sequence $\overline{vs}$ of length matching the arity of C. $\mathbb{C} : \mathsf{A} \rightsquigarrow \mathsf{B}$ denotes that $\mathbb{C}$ is an expression context containing one or more holes of type A, having an overall type B. For any expression $a :: \mathsf{A}$, $\mathbb{C}[\![a]\!]$ is the context $\mathbb{C}$ with all holes filled with the expression a.

Given contexts $\mathbb{C}, \mathbb{D} : \mathsf{A} \rightsquigarrow \mathsf{B}$, for any algebraic data types A and B, then structural-induction provides the following inference rule:

$$\frac{\mathbb{C}[\![\bot]\!] \equiv \mathbb{D}[\![\bot]\!] \qquad \forall (\mathsf{C}\ \overline{vs} :: \mathsf{A})\,.\ \ (\forall (v \in \overline{vs}, v :: \mathsf{A})\,.\ \ \mathbb{C}[\![v]\!] \equiv \mathbb{D}[\![v]\!]) \Rightarrow (\mathbb{C}[\![\mathsf{C}\ \overline{vs}]\!] \equiv \mathbb{D}[\![\mathsf{C}\ \overline{vs}]\!])}{\forall (a :: \mathsf{A})\,.\ \ \mathbb{C}[\![a]\!] \equiv \mathbb{D}[\![a]\!]}$$

STRUCTURAL-INDUCTION

The conclusion is called the *induction hypothesis*. Informally, the premises of the judgement require that:

- the induction hypothesis holds for undefined values;
- the induction hypothesis holds for any fully applied constructor, given that it holds for any argument of that constructor (of the same type).

HERMIT implements this induction scheme as a rewrite over clauses named $\mathit{inductionR}$.

$$\mathit{inductionR} :: (\mathit{Id} \to \mathit{Bool}) \to \mathit{RewriteH}\ \mathit{Clause}$$

This rewrite accepts a predicate for selecting the quantifier over which to induct. It then transforms a clause matching the induction hypothesis into a conjuction of the premises, essentially applying the STRUCTURAL-INDUCTION judgement as a rewrite bottom-to-top.

As an example, applying $\mathit{inductionR}$ to the clause $\forall\ xs\,.\ \ xs +\!\!+ [\,] \equiv xs$ transforms it to:

$$(\bot +\!\!+ [\,] \ \equiv\ \bot)\ \wedge\ ([\,] +\!\!+ [\,] \ \equiv\ [\,])\ \wedge\ (\forall\ y\ ys\,.\ \ (ys +\!\!+ [\,] \ \equiv\ ys)\ \Rightarrow\ ((y : ys) +\!\!+ [\,] \ \equiv\ (y : ys)))$$

This resulting clause can now be further rewritten using standard unfolding and simplification rewrites. The implication antecedent generated in the third sub-clause will be key to rewriting the consequent. Recall that antecedents are available as local lemmas in the consequent (Section 5.6).

This form of structural induction is somewhat limited in that it only allows the induction hypothesis to be applied to a variable one constructor deep. Concretely, applying induction again on

ys above would yield an additional hypothesis which includes y, rather than the desired original hypothesis instantiated to a variable that is substructural to ys.

Consequently, there are some data types for which HERMIT's induction scheme is insufficient to prove properties. Generalizing the induction rewrite to handle recursive calls which are n constructors deep remains future work.

5.10.6 Remembered Definitions

Commonly, when reasoning about a function definition, a prior definition of that function is needed. If the definition has been transformed equationally, then any prior definition is transitively equivalent to the current definition, so any prior definition is a valid unfolding for the function.

In pen-and-paper reasoning, using a past definition is a matter of looking up the page for a version of the function which is useful for the current step. Since HERMIT maintains a revision history of all versions of the GHC Core program, it certainly has access to all versions of the function so far encountered. HERMIT could offer an interface such as the following, where $ASTId$ denotes a specific version of the program, from which it finds the definition of the function specified by $HermitName$.

$unfoldR :: (...) \Rightarrow HermitName \to ASTId \to RewriteH\ CoreExpr$
$foldR :: (...) \Rightarrow HermitName \to ASTId \to RewriteH\ CoreExpr$

However, a single reasoning session may involve dozens or hundreds of steps, making specifying the correct version of the function difficult for the user. Additionally, this interface makes scripted transformations both less clear and more brittle. $ASTId$ version numbers are not explicitly denoted in scripts, so the given step may seem to refer to a magic number. Modifying the script may create additional intermediate $ASTId$s, requiring all subsequent uses of the identifiers to be modified.

Instead, HERMIT builds upon lemmas to offer a facility for reasoning with 'remembered' definitions. This requires the user to tell HERMIT to explicitly 'remember' definitions with a user-specified name. Remembering a definition simply creates an assumed lemma whose left-hand side is the function name and whose right-hand side is the function body.

The transformation which remembers a definition is called *rememberT*. It applies to either a non-recursive binding or a single binding in a recursive binding group. A lemma is created from the binding using the supplied lemma name. Any lambda bindings at the head of the right-hand side of the binding become universally quantified variables of the lemma.

For example, applying *rememberT* `"initsumlength"` to the following binding:

```
sumlength :: [Int] → (Int, Int)
sumlength = λ xs → (,) Int Int (sum xs) (length xs)
```

generates this lemma:

```
initsumlength (Proven)
  ∀ xs. sumlength xs ≡ (,) Int Int (sum xs) (length xs)
```

The lemma is assumed proven so it can immediately be used as a rewrite.

Chapter 6

Case Study: Proving Type-Class Laws

The most prominent example of informal equational reasoning in Haskell is type-class laws. Type-class laws are properties of type-class methods that the class author expects any *instance* of the class to satisfy. However, these laws are typically written as comments in the source code, and are not enforced by a compiler; the onus is on the instance declarer to manually verify that the laws hold. For example, the following documentation for the $Functor$ class is included in the Haskell standard library.

> ```
> class Functor f where
>   fmap :: (a → b) → f a → f b
> ```
>
> Instances of $Functor$ should satisfy the following laws:
>
> ```
> fmap id == id
> fmap (f ∘ g) == fmap f ∘ fmap g
> ```
>
> The instances of $Functor$ for lists, $Maybe$ and IO satisfy these laws.

A similar situation arises regarding GHC's rewrite rules [Peyton Jones et al., 2001]. GHC applies these rules as optimisations at compile-time, without any check that they are semantically correct; the onus is again on the programmer to ensure their validity. This is a fragile situation: even if the laws (or rules) are correctly verified by hand, any change to the implementation of the

Monoid		
mempty-left	$\forall x\,.$	$mempty \diamond x \equiv x$
mempty-right	$\forall x\,.$	$x \diamond mempty \equiv x$
mappend-assoc	$\forall x\ y\ z\,.$	$(x \diamond y) \diamond z \equiv x \diamond (y \diamond z)$
Functor		
fmap-id		$fmap\ id \equiv id$
fmap-distrib	$\forall g\ h\,.$	$fmap\ (g \circ h) \equiv fmap\ g \circ fmap\ h$
Applicative		
identity	$\forall v\,.$	$pure\ id \circledast v \equiv v$
homomorphism	$\forall f\ x\,.$	$pure\ f \circledast pure\ x \equiv pure\ (f\ x)$
interchange	$\forall u\ y\,.$	$u \circledast pure\ y \equiv pure\ (\lambda f \rightarrow f\ y) \circledast u$
composition	$\forall u\ v\ w\,.$	$u \circledast (v \circledast w) \equiv pure\ (\circ) \circledast u \circledast v \circledast w$
fmap-pure	$\forall g\ x\,.$	$pure\ g \circledast x \equiv fmap\ g\ x$
Monad		
return-left	$\forall k\ x\,.$	$return\ x \gg\!= k \equiv k\ x$
return-right	$\forall k\,.$	$k \gg\!= return \equiv k$
bind-assoc	$\forall j\ k\ l\,.$	$(j \gg\!= k) \gg\!= l \equiv j \gg\!= (\lambda x \rightarrow k\ x \gg\!= l)$
fmap-liftm	$\forall f\ x\,.$	$liftM\ f\ x \equiv fmap\ f\ x$

Figure 6.1: Laws Proven in the Type-Class Laws Case Study.

involved functions requires that the proof be updated accordingly. Such proof revisions can easily be neglected, and, furthermore, even if the proof is up-to-date, a user cannot be sure of that without manually examining the proof herself. What is needed is a mechanical connection between the source code, the proof, and the compiled program.

This case study proves a number of type-class laws on common Haskell data types. These laws, listed in Figure 6.1, are expected to hold of any instance of the class. Types targeted include lists, *Maybe*, and the *Map* type from the `containers` package, as well as *Identity* and *Reader* from the `transformers` package. Both `containers` and `transformers` are core standard libraries for Haskell. Each law was stated as a GHC rewrite rule and loaded into HERMIT as a lemma. The laws were then instantiated for each type and proved, when possible. The results are summarised in Table 6.1.

Note that these laws were proven for the actual data types and class instances defined in the `base`, `containers`, and `transformers` packages. Occasionally these instance methods could be defined in a way that is more amenable to reasoning. For example, the *Applicative* instances are usually defined in terms of $Monad$, which complicates the proofs. This case study operates on the actual types and instances because it reflects proving laws for real code.

The study begins by providing a full example of proving a single law (Section 6.1). It then describes how to modify the `containers` Cabal file to cause the proofs to be automatically checked during compilation (Section 6.2) and discusses some practical issues when proving properties in GHC Core (Section 6.3). Finally, it concludes with reflection on the overall success of the case study (Section 6.4).

6.1 Example: return-left Monad Law for Lists

This section walks through a HERMIT proof for the return-left Monad Law for lists in order to give a flavor for the work involved in proving a type-class law. The steps in this proof involve more complex transformations than previous examples, demonstrating the advantages of using KURE's strategy combinators for directing transformations.

In order to observe the effect of instantiation on the types of the lemma quantifiers, HERMIT's pretty printer is first instructed to display detailed type information. The general law, which has already been loaded from a GHC RULES pragma, is then copied in preparation for instantiation.

```
hermit> set-pp-type Detailed
hermit> copy-lemma return-left return-left-list

return-left-list (Not Proven)
  ∀ (m :: ⋆ → ⋆)
    (a :: ⋆)
    (b :: ⋆)
    ($dMonad :: Monad m)
    (k :: a → m b)
    (x :: a).
  (>>=) m $dMonad a b (return m $dMonad a x) k ≡ k x
```

Next, the type variable m is instantiated to the list type constructor.

```
hermit> inst-lemma return-left-list 'm [| [] |]

return-left-list (Not Proven)
  ∀ (a :: ⋆) (b :: ⋆) ($dMonad :: Monad []) (k :: a → [b]) (x :: a).
  (>>=) [] $dMonad a b (return [] $dMonad a x) k ≡ k x
```

(The `[|  |]` syntax are delimiters enclosing manually written Core expressions, which HERMIT then parses and resolves.) Entering proof mode, the type of the dictionary binder has been fully determined, so it can also be instantiated.

```
hermit> prove-lemma return-left-list
hermit> inst-dictionaries

Goal:
  ∀ (a :: ⋆) (b :: ⋆) (k :: a → [b]) (x :: a).
  (>>=) [] $fMonad[] a b (return [] $fMonad[] a x) k ≡ k x
```

Next, note that the `(return [] $fMonad[] a x)` expression can be simplified to $[x]$. This can be achieved by unfolding `return`, which will expose a case expression which scrutinises the `$fMonad[]` dictionary. This will be simplified away by HERMIT's powerful `smash` rewrite, which performs a number of simplifying rewrites until exhaustion. This will leave the actual instance method defining $return$ for lists, which can also be unfolded. Rather than do this step by step, HERMIT is directed to focus on the application of `return` and repeatedly unfold and smash the expression.

```
proof> { application-of 'return ; repeat (unfold <+ smash) }

Goal:
  ∀ (a :: ⋆) (b :: ⋆) (k :: a → [b]) (x :: a).
  (>>=) [] $fMonad[] a b ((:) a x ([] a)) k ≡ k x
```

Now to simplify away the $\gg\!=$ applications. Unfolding $\gg\!=$ directly results in a locally defined recursive worker named go, in terms of which the list instance of $\gg\!=$ is defined. Proving in the context of this recursive worker is tedious and brittle. It is cleaner to prove the following pair of lemmas separately, then apply them as necessary during this proof:

$$\begin{array}{lllcl}
\textsf{bind-left-nil} & \forall\, k\,. & [\,] \gg\!= k & \equiv & [\,] \\
\textsf{bind-left-cons} & \forall\, x\ xs\ k\,. & (x : xs) \gg\!= k & \equiv & k\ x +\!\!+ (xs \gg\!= k)
\end{array}$$

```
proof> one-td (lemma-forward bind-left-cons)

Goal:
  ∀ (a :: ⋆) (b :: ⋆) (k :: a → [b]) (x :: a).
  (++) b (k x) ((>>=) [] $fMonad[] a b ([] a) k) ≡ k x
```

```
proof> one-td (lemma-forward bind-left-nil)

Goal:
  ∀ (a :: *) (b :: *) (k :: a → [b]) (x :: a).
  (++) b (k x) ([] b) ≡ k x
```

Appealing to another auxiliary lemma, which can be proved by straightforward induction, eliminates the list append.

append-right $\quad \forall\, xs\,.\;\; xs +\!\!+ [\,] \;\equiv\; xs$

```
proof> one-td (lemma-forward append-right)

Goal:
  ∀ (a :: *) (b :: *) (k :: a → [b]) (x :: a). k x ≡ k x
```

As both sides are α-equivalent, the proof can be concluded.

```
proof> end-proof

Successfully proven: return-left-list
```

6.2 Configuring Cabal

As a GHC plugin, HERMIT integrates with GHC's existing ecosystem, including Haskell's premier packaging system, Cabal. Cabal packages feature a single, per-package configuration file. This file describes how Cabal should build the package, including how to build test cases.

HERMIT co-opts Cabal's test feature to mechanically check the proofs whenever the package is rebuilt. As an example, a `laws/` directory can be added to the `containers` package which contains three files. The first is `Laws.hs`, which provides the RULES pragmas that define the desired laws. The other two files are the proof scripts. One for the *Functor* proofs, the other for *Monoid*.

A `Test-suite` section, seen in Figure 6.2, is then added to the Cabal configuration file for `containers`. This defines the target code for the test, which is `Laws.hs`, along with build dependencies. The build dependencies shown are those of the `containers` library, plus an additional dependency on `hermit`. Note that this additional dependency is only for the test case, and does not change the dependencies of the `containers` library itself.

```
Test-suite hermit-proofs
    hs-source-dirs: laws, .
    main-is: Laws.hs
    type: exitcode-stdio-1.0

    build-depends: base >= 4.2 && < 5, array,
                   deepseq >= 1.2 && < 1.4, ghc-prim
    ghc-options:
        -fexpose-all-unfoldings
        -fplugin=HERMIT
        -fplugin-opt=HERMIT:Main:laws/Functor.hec
        -fplugin-opt=HERMIT:Main:laws/Monoid.hec
        -fplugin-opt=HERMIT:Main:resume

    build-depends: hermit
```

Figure 6.2: Test Added to Cabal Configuration File for `containers`.

Cabal runs the proofs by providing GHC the required series of flags. The `-fexpose-all-unfoldings` flag is described in Section 6.3.3. The `-fplugin=HERMIT` flag enables HERMIT, and the remaining three flags direct HERMIT to target the *Main* module (found in `Laws.hs`) with two scripts, resuming compilation on successful completion.

The proof scripts should also be added to the configuration file's `extra-source-files` section, so they are included in source distributions. The normal Cabal testing work-flow can be used to check the proofs.

```
$ cabal configure -enable-tests
$ cabal build
```

Note that the executable generated by the test does not have to actually be run, as the proof checking is done at compile time.

6.3 Proving in GHC Core

Despite being a small, relatively stable, typed intermediate language, GHC Core is not designed with proof in mind. Consequently, there are a few practical concerns and limitations when proving properties in GHC.

6.3.1 Implications

Recall that the $Monoid$ instance for $Maybe\ a$ requires a $Monoid$ instance for the type a. Likewise, the proof of the mappend-assoc law for $Maybe\ a$ depends on the law also holding for a. Thus, the following modified mappend-assoc lemma for $Maybe$ must be proved instead:

mappend-assoc-impl

$$\begin{aligned}&\forall\, m.\ \ (\forall\,(x :: m)\ (y :: m)\ (z :: m).\ \ (x \diamond y) \diamond z \ \equiv\ x \diamond (y \diamond z))\\ &\qquad \Rightarrow\\ &\qquad (\forall\,(i :: Maybe\ m)\ (j :: Maybe\ m)\ (k :: Maybe\ m).\ \ (i \diamond j) \diamond k \ \equiv\ i \diamond (j \diamond k))\end{aligned}$$

The proof proceeds by rewriting the consequent of the implication using the antecedent. HERMIT cannot yet generate such an implication lemma from the original mappend-assoc lemma automatically. Though it can spot the superclass constraint, it has no way of knowing which laws are associated with a given type class, because GHC itself does not have this information. Instead, the implication is constructed and introduced directly using a purpose-built KURE transformation.

6.3.2 Newtypes

GHC's **newtype** declaration offers the benefits of type abstraction with no runtime overhead [Breitner et al., 2014b, Vytiniotis and Peyton Jones, 2013]. To accomplish this, GHC Core implements newtype constructors as casts around the enclosed expression, rather than as normal algebraic data constructors. These casts are erased before code generation.

Proofs in the presence of **newtype**s must deal with these casts explicitly. HERMIT's `smash` rewrite is effective at floating and eliminating casts in all the examples in this study. In some cases `smash` could not eliminate all the casts, but still produced alpha-equivalent expressions, allowing the proof to be completed without further cast elimination.

6.3.3 Missing Unfoldings

Equational reasoning often involves *fold/unfold* transformations [Burstall and Darlington, 1977]. One consequence of the choice to target GHC Core and work within GHC, is that in order to unfold functions defined in previously compiled modules, HERMIT relies on the unfolding information present in the interface files generated by GHC. Unfortunately, GHC does not normally include unfoldings for recursive functions in the interface files for their defining modules. This prevents HERMIT from unfolding those functions. This includes, for example, the $+\!\!+$ and map functions. There are three work-arounds to this issue.

The first option is to recompile the defining packages with GHC's `-fexpose-all-unfoldings` flag. This is the prefered solution when the needed unfoldings are defined in the current package. For example, when proving the monoid and functor laws for `containers`, the test suite can compile the library with this flag to get the unfoldings HERMIT needs. However, in the case of $+\!\!+$ and map, this would mean recompiling the `base` package, which is not possible without reinstalling GHC itself.

The second option is to redefine the function with a new name, and use that function instead of the library function. For example:

```
myAppend :: [a] → [a] → [a]
myAppend []       ys = ys
myAppend (x : xs) ys = x : myAppend xs ys
```

However, this is not an option when reasoning about pre-existing code that uses the library version of the function.

A third option is to define a GHC rewrite rule to convert calls to the library function into calls to the new function, and then use this rule to transform the program before beginning to reason about it. For example:

```
{-# RULES "my-append" [~]     (++) = myAppend #-}
```

This is the solution used by this case study to get unfoldings for $+\!\!+$ and map. None of these work-arounds is ideal, and finding a cleaner solution remains as future work. Possibilities include making the `-fexpose-all-unfoldings` behavior the default for GHC, or implementing a systematic

Law (see Fig. 6.1)	Data type				
	List	Maybe	Map	Identity	Reader
mempty-left	7	5	5	N/A	
mempty-right	9	5	5		
mappend-assoc	9	16	-		
fmap-id	10	7	15	5	10
fmap-distrib	11	8	20	5	10
identity	7	7	N/A	5	15
homomorphism	8	5		5	15
interchange	17	5		5	15
composition	-	5		5	15
fmap-pure	18	5		5	15
return-left	7	5		5	12
return-right	10	5		5	12
bind-assoc	11	5		5	12

Numbers represent length of proof script, including instantiation steps.

Table 6.1: Summary of Proven Type-Class Laws.

means of compiling alternative, parallel versions of libraries including all unfolding information in a manner similiar to GHC's existing profiling and dynamic-linking compilation modes.

6.4 Reflections

Results for the case study are listed in Table 6.1, and the complete set of HERMIT proof scripts are available online [Farmer et al., 2015]. The numbers in the table represent the number of lines in the proof script, including instantiation steps. Overall, proving type-class laws in GHC Core appears to be viable with the simple reasoning techniques offered by HERMIT.

In general, the proofs were brief, and predominantly consisted of unfolding definitions and simplification, with relatively simple reasoning steps. Once this is done, any required inductive proofs tend to be short and straightforward.

Unsurprisingly, proving auxiliary lemmas for use in larger proofs helped to manage complexity. Proving the larger lemmas directly required working at a lower level, and led to a substantial amount of duplicated effort. This was especially true of the *Applicative* laws, as the *Applicative* instances were often defined in terms of their $Monad$ counterparts. Unfolding a single $\circledast$ results in several calls to $\gg\!=$. In the case of lists, naively unfolding $\gg\!=$ results in a local recursive worker function. Proving equalities in the presence of such workers requires many tedious unfolding and let-floating transformations. Using proven auxiliary lemmas about $\gg\!=$ avoided this tedium.

No attempt was made to quantify the robustness of the proof scripts to changes in the underlying code. The types and instances for which the laws were proven are relatively stable over time. As most of these proofs were fairly heavy on unfolding and simplification, they are expected to be sensitive to changes. However, HERMIT's interactive proof mode does allow the user to stop a proof script midway, lowering the burden of amending existing proofs.

Configuring a Cabal package to check proofs on recompilation is straightforward, requiring a single additional section to a package's Cabal configuration file. Proofs can be checked at any time by enabling the package tests. End users of the package can still build and install the package exactly as before.

Finally, note that while this case study focused on type-class laws, the approach outlined here could be used to provide proofs to accompany the GHC RULES pragmas commonly included in Haskell libraries.

Chapter 7

Case Study: `concatMap`

This chapter presents a case study in developing a domain-specific optimization using HERMIT. The optimization itself is described in detail, along with a simplification algorithm which is required to enable the key transformation in practice. This simplification algorithm was developed by using the HERMIT Shell to interactively apply the transformation to example programs. From this, an intuition was developed for directing key steps, such as unfolding, that create the conditions necessary for the transformation to succeed.

The primary benefit of the optimization is that it allows programmers to express higher-order sequence computations at a high level of abstraction, but still achieve the performance of a hand-fused loop. Avoiding the need to write this low-level loop code mitigates many possible bugs. Thus, a program which is more "obviously correct" can also be fast.

7.1 Introduction

In functional languages, it is natural to implement sequence-processing pipelines by gluing together reusable combinators, such as $foldr$ and zip. These combinators communicate their results to the next function in the pipeline by means of intermediate data structures, such as lists. If these pipelines are compiled in a straightforward way, the intermediate structures adversely affect performance as they must be allocated, traversed, and subsequently garbage collected.

Many techniques, collectively known as *deforestation* [Wadler, 1988, Hinze et al., 2011] or *fusion*, exist to transform such programs to eliminate these intermediate structures. Intuitively, rather than allow each combinator to transform the entire sequence in turn, the resulting code processes sequence elements in an assembly-line fashion. In many cases, after fusion, no sequence structures need to be allocated at all.

Shortcut (or algebraic) fusion works by expressing sequence computations using a set of primitive producer and consumer combinators, along with rewrite rules that combine, or fuse, consumers and producers. The three most well-known shortcut fusion systems, foldr/build [Gill et al., 1993], its dual unfoldr/destroy [Svenningsson, 2002], and Stream Fusion [Coutts et al., 2007], each choose a different set of primitive combinators and fusion rules.

This choice determines which sequence combinators can be fused by each system[1]. The trade-offs are briefly summarized here, though an excellent and thorough overview of the three systems can be found in Coutts [2010].

The foldr/build system cannot fuse zip-like combinators which consume more than one sequence. It also cannot fuse consumers which make use of accumulating parameters, such as $foldl$, without a subsequent non-trivial arity-raising transformation [Gill, 1996]. Despite these shortcomings, GHC has used foldr/build to fuse list computations for 20 years in part because it performs well on nested sequence computations, such as $concatMap$, which are common in list-heavy code.

The unfoldr/destroy system fuses zip and $foldl$, but cannot fuse $filter$ or $concatMap$. Stream Fusion improves on unfoldr/destroy by fusing $filter$, but it still cannot fuse $concatMap$. Stream Fusion is currently the system of choice for array computations, which tend to heavily use zip, $foldl$, and $filter$.

This case study enhances Stream Fusion so that it fuses $concatMap$. This enhancement removes a significant limitation which prevents Stream Fusion from replacing foldr/build as the fusion system of choice for GHC. This is accomplished by using HERMIT to transforms calls to

[1] There is a distinction between "fusion" and "fusion that results in an optimization". Fusion is only an optimization if it reduces allocation. Fusion may occur, but result in a function which allocates an internal structure equivalent to the eliminated sequence. In this case study, only fusion that results in an optimization is relavent, and this is the meaning intended when saying a particular system "can fuse" a given combinator.

$concatMap$ into calls to a similar combinator, $flatten$, which is more amenable to fusion. GHC's current user-directed rewriting system, GHC RULES, cannot express this transformation. Thus, while the transformation has been proposed previously, it has never been implemented in practice.

This case study explores the practicality and payoff of implementing such a transformation using HERMIT and applying it to real Haskell programs. There are many details, especially regarding simplification and desugaring, that were not obvious at the outset.

Section 7.4 describes a transformation from $concatMap$ to $flatten$ which enables fusion. This is extended to monadic streams in Section 7.4.2 so that it may be applied to vector fusion. The HERMIT implementation of the transformations includes a necessary simplification algorithm to enable the core transformation in practice (Section 7.5). The resulting system is applied to the **nofib** [Partain, 1993] suite of benchmark programs, demonstrating its advantage over foldr/build in list-heavy code (Section 7.6.2). It is also applied to the ADPfusion [Höner zu Siederdissen, 2012] library, which is used to write CYK-style parsers [Grune and Jacobs, 2008, Chapter 4.2] for single- and multi-tape grammars. The library makes heavy use of nested vector computations that need to be fused to achieve high performance and previously made extensive use of $flatten$. Applying the transformation with HERMIT simplifies the implementation of ADPfusion with no loss of performance (Section 7.7).

7.2 Stream Fusion

This section summarizes the Stream Fusion technique. Readers familiar with the topic may safely skip ahead, as none of this material is new. More detail can be found in Coutts et al. [2007] and Coutts [2010].

The key idea of Stream Fusion is to transform a pipeline of recursive sequence processing functions into a pipeline of non-recursive stream processing functions, terminated by a single recursive function which "runs" the pipeline. The non-recursive functions are known as *producers*, if they produce a stream, or *transformers*, if they transform one stream into another. The recursive function at the end of the pipeline is known as the *consumer*.

The benefit of this transformation is that it enables subsequent local transformations such as inlining and constructor specialization, which are generally useful and thus implemented by the compiler, to *fuse* the producers and transformers into the body of the consumer, yielding a single recursive function which produces no intermediate data structures. Stream Fusion relies on a data type which makes explicit the computation required to generate each element of a given sequence:

```
data Stream a where
  Stream :: (s → Step a s) → s → Stream a
data Step a s = Yield a s | Skip s | Done
```

A *Stream* is a pair of a *generator function* ($s \to Step\ a\ s$) and an existentially-quantified state (s). When applied to the state, the generator may give one of three possible responses, embodied in the *Step* type. Yield returns a single element of the sequence, along with a new state. Skip provides a new state without yielding an element. Done indicates that there are no more elements in the sequence. Generator functions are non-recursive, which allows them to be easily combined by GHC's optimizer.

Conversion to and from this *Stream* representation is done using a pair of representation-changing functions. This section uses Haskell lists as the sequence type, but the same technique works for other sequence types, such as arrays. The *stream* function is a producer that converts a list to a *Stream*:

```
stream :: [a] → Stream a
stream xs = Stream uncons xs
  where uncons :: [a] → Step a [a]
        uncons []       = Done
        uncons (x : xs) = Yield x xs
```

The state of *stream* is the list of values to which it is applied. The generator function yields the head of the list, returning the tail of the list as the new state.

The *unstream* function is a consumer that repeatedly applies the generator function to obtain the elements of the list:

```
unstream :: Stream a → [a]
unstream (Stream g s) = go s
  where go s = case g s of
                 Done       → []
                 Skip s′    → go s′
                 Yield x s′ → x : go s′
```

Using *stream* and *unstream*, list combinators can now be redefined in terms of their *Stream* counterparts. Consider *map*:

```
map :: (a → b) → [a] → [b]
map f = unstream ∘ mapS f ∘ stream
mapS :: (a → b) → Stream a → Stream b
mapS f (Stream g s0) = Stream mapStep s0
  where mapStep s = case g s of
                      Done       → Done
                      Skip s′    → Skip s′
                      Yield x s′ → Yield (f x) s′
```

Note that *stream* and *mapS*, as producer and transformer, respectively, are both non-recursive. Rather than traverse a sequence, *mapS* simply modifies the generator function. Wherever the original stream would have produced an element x, the new stream produces the value $f\ x$ instead. Subsequent inlining and case reduction will fuse the two generators into a single non-recursive function.

The final, crucial, ingredient is the following GHC rewrite rule, the proof of which can be found in Coutts [2010]:

$$stream \circ unstream \;\equiv\; id$$

As an example of Stream Fusion in action, consider a simple pipeline consisting of two calls to *map*.

```
map f ∘ map g
```

Unfolding *map* yields the underlying stream combinators.

```
unstream ∘ mapS f ∘ stream ∘ unstream ∘ mapS g ∘ stream
```

Applying the rewrite rule eliminates the intermediate conversion.

```
unstream ∘ mapS f ∘ mapS g ∘ stream
```

Inlining the remaining functions, along with their generators, and performing standard local transformations such as case reduction and the case-of-case transformation [Santos, 1995] results in the following recursive function, which produces no intermediate lists.

```
let go []       = []
    go (x : xs) = f (g x) : go xs
in go
```

In this case, Stream Fusion has effectively implemented the $map\ f \circ map\ g \equiv map\ (f \circ g)$ transformation.

7.3 Fusing Nested Streams

The *concatMap* combinator is a means of expressing nested list computations. It accepts a higher-order argument f and a list, referred to as the *outer* list. It maps f over each element of the outer list, inducing a list of *inner* lists. It returns the concatenation of the inner lists as its result. Similiar to *map* in the previous section, *concatMap* can be implemented in terms of its stream counterpart, *concatMapS*.

```
concatMap :: (a → [b]) → [a] → [b]
concatMap f = unstream ∘ concatMapS (stream ∘ f) ∘ stream
```

The *concatMapS* function is a non-recursive transformer with a somewhat complicated generator function.

```
concatMapS :: (a → Stream b) → Stream a → Stream b
concatMapS f (Stream g s) = Stream g′ (s, Nothing)
  where
    g′ (s, Nothing) =
        case g s of
          Done       → Done
          Skip s′    → Skip (s′, Nothing)
          Yield x s′ → Skip (s′, Just (f x))
    g′ (s, Just (Stream g″ s″)) =
        case g″ s″ of
          Done       → Skip    (s, Nothing)
          Skip s′    → Skip    (s, Just (Stream g″ s′))
          Yield x s′ → Yield x (s, Just (Stream g″ s′))
```

The state of the resulting stream is a tuple, containing as its first component the state of the outer stream (the second argument to *concatMap*). Its second component is optionally an inner stream.

The generator function g' operates in two modes, determined by whether the inner stream is present in the state (Just) or absent (Nothing). When the inner stream is absent, g' applies the generator for the outer stream to the first component of the state. When this results in a value x, it constructs a new state by applying f to x to obtain the inner stream.

Subsequent applications of g' will see the Just constructor and operate in the second mode, which applies the generator for the inner stream to its state. When the inner stream is exhausted, it switches back to the first mode by discarding the inner stream state.

Optimizing *concatMapS*, GHC will use *call-pattern specialization* [Peyton Jones, 2007] to eliminate the *Maybe* type, yielding two mutually recursive functions, one for each mode. Unfortunately optimization stops before all *Step* constructors are fused away.

```
go1 acc s         = ... go2 acc s' g'' s'' ...
go2 acc s g'' s'' = case g'' s'' of
                      Done       → go1 acc s
                      Skip s'    → go2 acc       s g'' s'
                      Yield x s' → go2 (acc + x) s g'' s'
```

The problem is that the generator for the inner stream g'' is an argument to *go2*, and therefore not statically known in the body of *go2*. Indeed, this follows from the original definition of *concatMapS* above, where g'' is bound by pattern matching on the tuple of states. The fact that g'' is not statically known in *go2* means it cannot be inlined, thwarting case reduction, which would have eliminated the *Step* constructors.

The code for g'' is statically known in *go1*. Additionally, *go2* always repasses g'' unmodified on recursive calls. The *static-argument transformation* (SAT) [Santos, 1995] could be applied to *go2* and the resulting wrapper could be inlined into *go1*. This would make the code for g'' statically known at its call site, enabling full fusion.

This approach was suggested in the original Stream Fusion paper [Coutts et al., 2007], but it involves a delicate interaction between call-pattern specialization and the SAT that is difficult to control. Aggressively applying the SAT can have detrimental effects on performance, so GHC is quite conservative in its use. In this case, GHC will not apply the SAT to *go2* automatically. Even if GHC had a means of targeting the SAT via source annotation, the fact that *go2* is generated by call-pattern specialization, at compile time, with an auto-generated name, means there is nothing in the source to annotate. Despite considerable effort by GHC developers, successfully applying this solution in the general case has remained elusive.

Stepping back, note that this is a consequence of the power of *concatMapS* itself. The inner stream, including its generator function, is created by applying a function to a value of the outer stream *at runtime*. That function could potentially pick from arbitrarily many *different* inner streams based on the value it is applied to. Each of these streams may have an entirely different generator function. In fact, since the type of the state in a *Stream* is existentially quantified, the returned streams may not even have the same state type.

A less powerful alternative to *concatMapS* is *flatten*. The type of *flatten* makes explicit that the generator, and the type of the state, of the inner stream are always the same, regardless of the value present in the outer stream. This means that *flatten* cannot express the choice of inner streams possible with *concatMapS*, but it is readily fused by GHC.

```
flatten :: (a → s)              -- initial state constructor
        → (s → Step b s)        -- generator
        → Stream a → Stream b
```

In the overwhelming majority of cases found in real code, the extra power of *concatMapS* is unnecessary, meaning *flatten* can be used instead. The disadvantage is that *flatten* is more difficult to use, as it breaks the *Stream* abstraction by exposing the user to the *Step* type. Whereas the rest of the Stream Fusion system hides the complexity of state and generator functions from the programmer, providing familiar sequence combinators, *flatten* requires one to think in terms of generator functions and state. A call to *concatMap* with a complicated inner stream pipeline can make use of existing stream combinators, while *flatten* requires the programmer to write a hand-fused, potentially complex generator function.

7.4 Transforming `concatMap` to `flatten`

In his dissertation, Coutts [2010] proposes the following transformation for optimizing common uses of $concatMap$ by transforming them into calls to $flatten$. The advantage of such a transformation is its specificity. Rather than manage a brittle interaction between two general program transformations with potential negative performance consequences, one specific transformation is performed which is known to be advantageous. This is exactly the motivation for GHC rewrite rules.

$$\forall\ g\ s\ .\ concatMapS\ (\lambda x \to \mathsf{Stream}\ g\ s) \Longrightarrow flatten\ (\lambda x \to s)\ g$$

This transformation is only valid if the state type and generator function of the inner streams are independent of the runtime values of the outer stream. That is, the state type and generator function are the same for each inner stream, and statically known. This restriction is exactly what allows the stream to be expressed in terms of $flatten$, and doing so makes this independence explicit.

While this transformation enforces the essential restriction that the value of x does not determine *which* generator and state is selected, it has the undesirable side condition that x cannot be free in g. This side condition severely limits the applicability of the transformation in practice. To see why this is a problem, consider this simple nested enumeration.

$$concatMapS\ (\lambda x \to enumFromToS\ 1\ x)\ (enumFromToS\ 1\ n)$$

As traditionally written, the generator for the inner $enumFromToS$ will necessarily depend on x in order to know when to stop generating additional values. The proposed transformation would fail to apply in this situation.

This could be worked around by carefully defining $enumFromToS$ such that it stores its arguments in the stream state. That is, an additional invariant could be placed on generator functions that they have no free variables that are not also free in their enclosing stream combinator definition. From a practical perspective, this complicates all stream combinator definitions for the benefit of $concatMap$. More complicated state types are required, which results in higher arity functions after call-pattern specialization, even when $concatMap$ is not present.

This section defines a more sophisticated transformation which separates these concerns, permitting g to use x to compute its result, without allowing x to determine which g is selected, and without requiring all stream combinators to be redefined with the additional invariant on their generators. Unfortunately, the GHC RULES system is incapable of expressing such a transformation.

7.4.1 Non-Constant Inner Streams

The principal limitation to the proposed transformation is the free variable check on the generator function. For any interesting use of $concatMapS$, this will fail. To lift this restriction, the transformation is altered to extend the state with the value of the outer stream. The generator function then has access to the value of the outer stream by way of the state. Note this transformation makes intentional use of variable capture (when x is free in g).

$$\begin{array}{l}\forall\ g\ s\,.\ concatMapS\ (\lambda x \rightarrow \mathsf{Stream}\ g\ s)\\ \qquad\Downarrow\\ \qquad flatten\ (\lambda x \rightarrow (x, s))\ (\lambda (x, s) \rightarrow fixStep\ x\ (g\ s))\end{array}$$

Notice this changes the type of the inner stream state. The original state can be projected out of the extended state in order to apply the original generator, getting a $Step$ result which contains a possible value and new state. This new state is of the *original* state type. A state of the extended type must be returned. To do this, an auxiliary $fixStep$ combinator is used to place x back into the state held by the $Step$ result, thereby lifting it to the extended state type.

$$\begin{array}{l}fixStep :: a \rightarrow Step\ b\ s \rightarrow Step\ b\ (a, s)\\ fixStep\ \_\ \mathsf{Done} \qquad\quad = \mathsf{Done}\\ fixStep\ a\ (\mathsf{Skip}\ s) \qquad = \mathsf{Skip}\ (a, s)\\ fixStep\ a\ (\mathsf{Yield}\ b\ s) = \mathsf{Yield}\ b\ (a, s)\end{array}$$

This improved transformation cannot be implemented as a GHC rewrite rule because it requires manipulating syntactic language constructs such as case expressions. More practically, it is rare that the body of the function argument is in Head-Normal Form (i.e. starting with an explicit Stream constructor). Often the body will involve a call to another stream combinator instead. A custom simplification algorithm (developed in Section 7.5) is used to expose the constructor.

7.4.2 Monadic Streams

So far, the transformation works on pure streams. The vector streams targeted in Section 7.7 are parameterized by a monad, permitting generator and state construction functions to perform monadic effects. This leads to the following definition of the stream datatypes.

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Stream} :: (* \to *) \to * \to * \mathbf{where} \\
\quad \mathsf{Stream} :: (s \to m\ (\mathit{Step}\ s\ a)) \to s \to \mathit{Stream}\ m\ a \\
\mathit{concatMapM}\ ::\ \mathit{Monad}\ m \\
\qquad\qquad\quad \Rightarrow (a \to m\ (\mathit{Stream}\ m\ b)) \\
\qquad\qquad\quad \to \mathit{Stream}\ m\ a \to \mathit{Stream}\ m\ b \\
\mathit{flattenM}\ ::\ \mathit{Monad}\ m \\
\qquad\qquad \Rightarrow (a \to m\ s) \\
\qquad\qquad \to (s \to m\ (\mathit{Step}\ s\ b)) \\
\qquad\qquad \to \mathit{Stream}\ m\ a \to \mathit{Stream}\ m\ b
\end{array}
$$

The Stream constructor of the inner stream is now wrapped in a monadic context. The simplest such context is return.

$$\mathit{concatMapM}\ (\lambda x \to \mathit{return}\ (\mathsf{Stream}\ g\ s))$$

However, the monadic context may also have an arbitrary number of binds which scope over the inner stream. The transformation must collect the bound values and store them in the state, like it does for the outer stream binder x. Here the monadic context is denoted as $\mathcal{M}\ll\ldots\gg$. Since the context is executed once per element of the outer stream, it can safely be moved to the state construction function of $\mathit{flattenM}$.

$$
\begin{array}{l}
\forall\ g\ s\ .\ \mathit{concatMapM}\ (\lambda x \to \mathcal{M}\ll\mathsf{Stream}\ g\ s\gg) \\
\qquad\quad \Downarrow \\
\qquad\quad \mathit{flattenM}\ (\lambda x \to \mathcal{M}\ll((x, b_1\ldots b_n), s)\gg) \\
\qquad\qquad\qquad\quad (\lambda((x, b_1\ldots b_n), s) \to \mathit{liftM}\ (\mathit{fixStep}\ (x, b_1\ldots b_n))\ (g\ s))
\end{array}
$$

Instead of storing x in the state, an n-ary tuple of x and the other binders is stored. The projection is modified to project out of this tuple. As a minor optimization, only those binders which appear free in the generator function are stored. Finally, $\mathit{fixStep}$ is lifted over the monadic result of the generator in the normal way.

7.5 Implementation

In practice, the body of the function passed to $concatMapS$ is not an explicit Stream constructor. The body must be simplified with Algorithm 1 in order to expose the constructor. This simplification is done by HERMIT when it tries to apply the transformation. While the HERMIT implementation necessarily operates on GHC Core, the code in this section is presented in Haskell syntax for clarity.

Algorithm 1 Simplification Algorithm **SIMPLIFY**

```
In order, repeatedly apply the first transformation that succeeds.

  1. Gentle simplification
  2. Once, in a top-down manner:
      (a) Apply the stream/unstream rule.
      (b) Float a let inwards.
      (c) Eliminate a case.
      (d) Reduce a case on an inner stream.
      (e) Float a case inwards.
  3. Unfold an application.
```

1. Gentle simplification Performs dead-let elimination, case reduction, β-reduction, limited (non-work-duplicating) let substitution, and unfolds Haskell's operators for function composition ($\circ$) and application (\$), as well as the identity function (id).

2a. Apply the `stream/unstream` rule. Recall the definition of $concatMap$ in terms of $concatMapS$ from Section 7.2.

$$concatMap :: (a \to [b]) \to [a] \to [b]$$
$$concatMap\ f = unstream \circ concatMapS\ (stream \circ f) \circ stream$$

After transformation, f will be composed of stream combinators wrapped in an $unstream$ which turns the stream back into a list. When f is inlined, this $unstream$ will unite with $stream$, enabling the elimination of both. If the $stream$ application were to instead be unfolded in search of an explicit Stream constructor that would be a commitment to having an intermediate list. (Recall that the state type of a $Stream$ produced by $stream$ is a list.)

2b. Float a let inwards. The inner stream will often be wrapped in let bindings, especially if a stream combinator has been unfolded. These bindings will scope over the entire Stream constructor, and may or may not depend on the value of the outer stream, so they cannot be reliably floated outwards. Observe, however, that they will never capture the Stream constructor itself, so they can be reliably floated inwards past the constructor.

$$\textbf{let}\ b = e\ \textbf{in}\ \mathsf{Stream}\ g\ s \Longrightarrow \mathsf{Stream}\ (\textbf{let}\ b = e\ \textbf{in}\ g)\ (\textbf{let}\ b = e\ \textbf{in}\ s)$$

Floating inwards necessarily duplicates let bindings. This loss of sharing could result in duplicated allocation and computation. In practice, it appears rare that a let binding is used in both the generator and the state (the two arguments to the Stream constructor), so at least one will usually be eliminated by the next gentle simplification step, which, according to Algorithm 1, will occur directly after this transformation. In addition to applications, lets are floated into case expressions and lambdas.

2c. Eliminate a case. This step eliminates a case with a single alternative when none of the binders of the case are free in the right-hand side of the alternative.

$$\frac{\forall v\ \in \overline{vs}.\ \ v\ \notin \mathit{freeVars}\ \mathit{rhs}}{\begin{array}{l}\textbf{case}\ e\ \textbf{of} \\ \quad \mathsf{C}\ \overline{vs} \to \mathit{rhs}\end{array}} \quad \Longrightarrow \mathit{rhs}$$

One might question the necessity of this rule. This situation most often arises from simplification step 2e (float a case inwards), which itself is primarily caused by strictness annotations. Strictness annotations, and use of the Haskell seq function, are desugared to case expressions. (Operationally, case expressions in GHC Core perform computation.) When a case is floated inwards in 2e, it is necessarily duplicated (just as when lets are floated inwards). Some of the duplicates may bind values that are never used.

2d. Reduce a case on an inner stream. Nested streams will result in a case expression on the inner stream. Consider:

$$
\begin{array}{l}
\mathit{concatMapS}\ (\lambda x \to \mathit{concatMapS}\ (\lambda y \to \mathit{enumFromToS}\ 1\ y) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{enumFromToS}\ 1\ x)) \\
\qquad\qquad (\mathit{enumFromToS}\ 1\ n)
\end{array}
$$

When transforming the outermost $concatMapS$, the body of the function argument will eventually be:

$$
\begin{array}{l}
\lambda x \to \textbf{case}\ \mathit{flatten}\ \mathit{mkS}\ g''\ (\mathit{enumFromTo}\ 1\ x)\ \textbf{of} \\
\qquad\quad \mathsf{Stream}\ g\ s \to \ldots\ \mathsf{Stream}\ g'\ s'\ \ldots
\end{array}
$$

GHC would have unfolded $flatten$ eventually. However, in order to transform the outer $concatMapS$ to $flatten$, it needs to be unfolded *now* so as to expose the Stream constructor in the right-hand side of the alternative.

A wrinkle arises if the head of the scrutinee is not a stream combinator. It may be a case expression, let expression, or application; or, more subtly, the $stream$ combinator itself.

$$
\begin{array}{l}
\lambda x \to \textbf{case}\ \mathit{stream}\ (\mathit{unstream}\ (\mathit{flatten}...))\ \textbf{of} \\
\qquad\quad \mathsf{Stream}\ g\ s \to \ldots\ \mathsf{Stream}\ g'\ s'\ \ldots
\end{array}
$$

If the algorithm were to unfold $stream$, the case would reduce, but it would fall prey to the same problem mentioned in step `2a`. That is, a commitment would be made to a state type that includes a list, missing a fusion opportunity.

These issues are all ones Algorithm 1 is designed to handle, so it is recursively applied to the scrutinee. When finished, it will yield an explicit Stream constructor, which will be reduced by the next gentle simplification step.

$$
\frac{e \Longrightarrow e'}{\begin{array}{lcl}
\textbf{case}\ e\ \textbf{of} & \Longrightarrow & \textbf{case}\ e'\ \textbf{of} \\
\quad \mathsf{Stream}\ g\ s \to \mathit{rhs} & & \quad \mathsf{Stream}\ g\ s \to \mathit{rhs}
\end{array}}
$$

2e. Float a case inwards. Strictness annotations result in case expressions with a single, always matching default alternative. The right-hand side of the alternative may refer to the bound result of the evaluated scrutinee. For the same reasons that lets cannot always be floated outward, the only option is to eliminate the case or float it inwards.

Note, however, that it cannot be eliminated wholesale, or step `2c` would have done so. Eliminating it involves either changing the case to a let, which would make the program more lazy, or substituting the scrutinee into the right-hand side of the alternative, which could duplicate work. Instead, the algorithm floats it inwards.

$$\mathbf{case}\ e\ \mathbf{of}\ v \to \mathsf{Stream}\ g\ s \implies \mathsf{Stream}\ (\mathbf{case}\ e\ \mathbf{of}\ v \to g)\ (\mathbf{case}\ e\ \mathbf{of}\ v \to s)$$

In most situations, one of these cases is eliminated, avoiding duplicated work, though how often this happens in practice has not been explicitly quantified. Within the context of the overall transformation, strictness remains unaltered. Nominally, if e is $\bot$, it is now possible to force the entire expression to the Stream constructor, where before it would have diverged. However, forcing just the Stream constructor without forcing one of its arguments provides no useful information, so only a pathological stream combinator would do so.

The other situation which requires this rule is the desugaring of pattern binders for list comprehensions (Section 7.5.2). The desugaring results in a case with a single alternative to bind the components of the pattern.

3. Unfold an application. The last resort is to unfold a function application. It is crucial that this step be tried last, for the reasons mentioned in steps `2a` and `2d`. That is, the algorithm should avoid unfolding the $stream$ combinator when possible, because it commits the program to a list state type, resulting in an intermediate list.

7.5.1 Multiple Inner Streams

A slightly more general version of the transformation in step `2e` as actually implemented. This more general version floats a case with multiple alternatives inwards if all alternatives share the same constructor and type.

$$
\begin{array}{ll}
\textbf{case}\ e\ \textbf{of} & \Longrightarrow \mathsf{Stream}\ (\textbf{case}\ e\ \textbf{of} \\
\quad \mathsf{P} \rightarrow \mathsf{Stream}\ g_1\ s_1 & \qquad\qquad\quad \mathsf{P} \rightarrow g_1 \\
\quad \mathsf{Q} \rightarrow \mathsf{Stream}\ g_2\ s_2 & \qquad\qquad\quad \mathsf{Q} \rightarrow g_2) \\
 & \qquad\qquad (\textbf{case}\ e\ \textbf{of} \\
 & \qquad\qquad\quad \mathsf{P} \rightarrow s_1 \\
 & \qquad\qquad\quad \mathsf{Q} \rightarrow s_2)
\end{array}
$$

This has the effect of merging two streams whose state type is the same. The strictness argument is the same as for step `2e`, though the resulting cases are less likely to be eliminated, so work duplication is an issue. It is easy to construct a small example which would benefit from this rule:

$$
\begin{array}{l}
concatMapS\ (\lambda x \rightarrow \textbf{case}\ even\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad \mathsf{True}\ \ \rightarrow enumFromToS\ 1\ x \\
\qquad\qquad\qquad\qquad\quad \mathsf{False} \rightarrow enumFromToS\ 1\ (x+1))
\end{array}
$$

It remains unclear how often this happens in a real program. How often the transformation has this effect has not been quantitatively measured. It does, however, suggest possible future work on a more involved means of merging streams.

7.5.2 List Comprehensions

Haskell offers convenient syntax for nested list computations in the form of list comprehensions. A list comprehension has a body and a series of clauses. A clause can be a generator, a guard, or a let expression. GHC desugars list comprehensions in two different ways. Haskell 98 desugaring [Peyton Jones, 2003] is used when optimization is disabled or parallel list comprehensions are present. With standard optimizations enabled, comprehensions are desugared to applications of $foldr$ and $build$ [Gill, 1996].

Neither scheme will result in good fusion with this extended Stream Fusion system. When desugaring guards and pattern-match failures, both will produce branching case expressions inside the function argument to $concatMap$. These case expressions cannot be merged because their state type differs, blocking the transformation.

Thus, in the paper on which this case study is based, a third means of desugaring comprehensions is provided. The translation is a novel extension of Haskell 98 desugaring, and is required to fuse list comprehensions with this system. Altering the behavior of GHC's desugarer is not possible with a GHC plugin, so is outside the scope of HERMIT's capabilities. Since this study is intended to focus on HERMIT's role in the optimization, the desugaring details are not presented here. The interested reader is referred to Farmer et al. [2014]. Note, however, that the results presented in Section 7.6.2 were generated with this desugaring enabled because many of the benchmark programs feature list comprehensions, rather than explicit calls to list combinators.

7.5.3 Call-Pattern Specialization

Stream Fusion depends crucially on call-pattern specialization to eliminate the constructors in the stream state. GHC allows a data type to be annotated to indicate that it should try to aggressively specialize functions with arguments of the annotated type. In the following example, *sPEC* is the dummy argument of such an annotated type, and should force GHC to specialize *go*. Unfortunately GHC tends to bind deeply nested constructors and float them outwards, defeating specialization.

```
let s1 = (,) (Left (Left (Left (Just (I# 3))))) Nothing
    go = ... go ...
in  go sPEC (I# 0) s1
```

To counter this, immediately before call-pattern specialization runs, a HERMIT transformation collects all the non-recursive bindings in the program whose heads are non-recursive data constructors and inlines them unconditionally.

To do so, a *nonRecDataConT* transformation is defined which matches only on constructor applications. It then checks that none of the arguments to the constructor have the same type as the constructor's result type, using two functions supplied by the GHC API to get this information. This check is to avoid inlining recursive constructors, such as the (:) constructor for lists. These recursive constructors cannot be specialized by call-pattern specialization anyway.

The *nonRecDataConT* transformation is defined in such a way that it only succeeds if expression is of the desired form. Using this, it can be lifted using the *letT* and *nonRecT* congruence combinators to only succeed on non-recursive let expressions where the right-hand side of the binding is an expression on which *nonRecDataConT* succeeds. This acts as a guard which only succeeds on let expressions which should be inlined. The actual inlining is done by the *letNonRecSubstR* transformation from the HERMIT Dictionary. The whole rewrite is applied wherever possible, in a top-down manner, using the *anytdR* traversal provided by KURE.

```
inlineConstructorsR :: RewriteH Core
inlineConstructorsR = anytdR
                      $ promoteExprR
                      $ do letT (nonRecT successT nonRecDataConT const) successT const
                             letNonRecSubstR
  where nonRecDataConT = do
          (dc, _tys, _args) ← callDataConT
          let dcResTy  = dataConOrigResTy dc
              dcArgTys = dataConOrigArgTys dc
          guardMsg (not (any (eqType dcResTy) dcArgTys)) "constructor is recursive"
```

In practice, this simple heuristic offers a huge improvement in the form of decreased allocation, because specialization completely eliminates the constructors.

7.5.4 The Plugin

A custom plugin which applies the transformation is defined using the Plugin DSL. The plugin consists of two modules.

The first module provides the definitions of the Stream Fusion combinators and the *fixStep* function (Section 7.4.1), along with GHC RULES which transform list combinators into their Stream Fusion counterparts. A dependency on this module will be injected into the target module when the transformation runs, so code targeted by the transformation need not be altered in any way.

The plugin follows the standard Stream Fusion approach to inlining and forcing call-pattern specialization. Where it differs from tradition is the rewrite rules it provides for list combinators. Traditional Stream Fusion makes use of an alternative list prelude, in which list combinators were defined directly in terms of stream combinators. Instead, the plugin provides GHC RULES such as this one:

```
{-# RULES “map-mapS” [~] forall f . map f ≡ unstream ∘ mapS f ∘ stream #-}
```

```
module HERMIT.Optimization.StreamFusion where
import HERMIT
plugin :: Plugin
plugin = hermitPlugin $ λ_opts → do
  pass 0 $ apply (Always "-- apply all rules")
        $ do f ← compileRulesT allRules
             tryR (repeatR (anytdR (promoteExprR $ runFoldR f) <+ simplifyR))
  apply (Always "-- concatmap -> flatten")
    $ tryR $ repeatR $ anyCallR $ promoteExprR concatMapSR
  before SpecConstr $ apply (Always "-- inline constructors")
                    $ tryR inlineConstructorsR
```

Figure 7.1: Stream Fusion HERMIT Plugin

The motivation for this approach is two-fold. First, there is no need to hide or redefine the list combinators in the Prelude. This would be less onerous if Stream Fusion were the default list implementation, but redefining Prelude functions makes selective use more difficult.

Second, and more importantly, the desugaring of list comprehensions requires GHC to assign globally unique identifiers to the combinators it generates, so it can generate proper names even if the combinators themselves are not visible to the compiler. Desugaring directly to stream combinators, or even the alternative prelude combinators, would require the combinator and module names to be hard-coded into GHC itself. A change to the Stream Fusion library would require a corresponding change to GHC. On the other hand, the standard list combinators are stable and not likely to change, and are already assigned unique names by GHC. Desugaring to these standard combinators, then rewriting with rules, minimizes changes to GHC itself.

The second module provides the actual GHC plugin, defined using HERMIT's Plugin DSL. This module is the one specified to GHC using the `-fplugin` flag when compiling the target program. Figure 7.1 presents the plugin definition.

The fact that list combinators already have GHC rewrite rules defined for foldr/build is a complicating factor. To sidestep this, the plugin applies all of its rules to exhaustion before the first optimization pass runs. This transforms all the list combinators before their own rules get a chance to fire.

Then, before and after every optimization pass, the $concatMap$->$flatten$ transformation is applied whenever possible. Finally, before the call-pattern specialization pass, non-recursive constructor bindings are aggressively inlined, as discussed in Section 7.5.3.

7.6 Performance

To evaluate the performance benefits of the transformation, it is applied to both Haskell lists and vectors. First, Section 7.6.1 benchmarks the results of the optimization on several micro-benchmarks that exercise different aspects of the transformation. Next, Section 7.6.2 applies the transformation to GHC's `nofib` benchmark suite. Finally, Section 7.6.3 illustrates how $concatMap$ can sometimes lead to *better* performance than $flatten$.

All measurements were performed on a 64-bit 1.7Ghz Intel Core^{TM} i7-4650U, with 8GB RAM, running OS X 10.9.5 and GHC 7.8.3 (modified with the additional desugaring scheme).

7.6.1 Micro-benchmarks

In order to illustrate the performance gap between $flatten$ and $concatMap$, and thus characterize the potential benefit of the transformation, it is applied to the micro-benchmarks listed below. Note that these particular benchmarks characterize *best case* improvements, as they are designed to result in tight loops on unboxed integers.

The graph in Figure 7.2 summarizes the results. Each benchmark provides the following measurements, where appropriate:

- **concatMap** Use the `vector` library's $concatMap$ combinator. This represents the current status quo for Stream Fusion.
- **flatten** Use `vector`'s $flatten$ combinator and hand-written generator functions.
- **Optimized** Apply the transformation to **concatMap**.
- **List** Use lists and apply foldr/build.

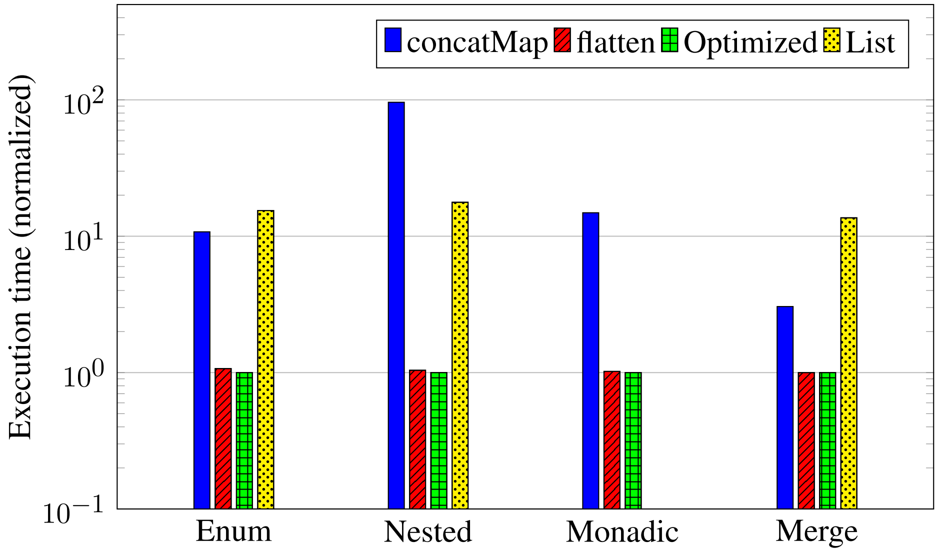


Figure 7.2: Micro-benchmark Performance Results

Enum This benchmark characterizes the potential speedup gained by using $flatten$ instead of $concatMap$, and demonstrates that the optimization can fully close the gap.

$$f\ n = foldl'\ (+)\ 0\ (concatMap\ (enumFromTo\ 1)\ (enumFromTo\ 1\ n))$$

Nested This benchmark demonstrates the advantage foldr/build normally has on nested list computations. Note that lists outperform vectors despite being slower in **Enum**. In this case, superior fusion for lists is overcoming vector's normally superior data structure.

$$\begin{aligned}f\ n = foldl'\ (+)\ 0\ (&concatMap\ (\lambda x \rightarrow concatMap\ (\lambda y \rightarrow enumFromTo\ y\ x)\\ &\qquad\qquad\qquad\qquad\qquad (enumFromTo\ 1\ x))\\ &\quad (enumFromTo\ 1\ n))\end{aligned}$$

Monadic This benchmark exercises the monadic stream transformation. As lists are not parameterized over a monad, they are absent from this comparison.

$$\begin{aligned}f\ n = runST\ (\mathbf{do}\ &vec \leftarrow getVector\\ &foldl'\ (+)\ 0\ (concatMapM\ (\lambda x \rightarrow \mathbf{do}\ z \leftarrow readVector\ vec\ x\\ &\qquad\qquad\qquad\qquad\qquad\qquad return\ (enumFromTo\ 1\ z))\\ &\qquad\qquad\qquad (enumFromTo\ 1\ n)))\end{aligned}$$

Negative figures indicate an improvement over foldr/build.

Program	Allocs	Runtime	Elapsed	TotalMem
bernouilli	-6.4%	+0.0%	+0.0%	+0.0%
exp3_8	+0.0%	-1.3%	-1.2%	+0.0%
gen_regexps	-44.7%	-45.4%	-45.4%	-56.7%
integrate	-61.3%	-48.2%	-42.6%	+0.0%
kahan	+0.0%	+1.2%	1.4%	+0.0%
paraffins	+129.9%	-11.6%	-12.6%	-29.1%
primes	+2.2%	-15.2%	-15.9%	-33.3%
queens	-17.1%	-7.5%	-6.9%	+0.0%
rfib	+0.0%	+0.0%	+0.0%	+0.0%
tak	+0.0%	-4.7%	-4.7%	+0.0%
wheel-sieve1	+195.2%	+4.6%	+4.7%	-29.6%
wheel-sieve2	-0.6%	-1.2%	-2.4%	+0.0%
x2n1	-77.4%	+0.0%	+0.0%	+0.0%
Min	-77.4%	-48.2%	-45.4%	-56.7%
Max	+195.2%	+4.6%	+4.7%	+0.0%
Geom. Mean	-9.9%	-15.2%	-14.5%	-13.8%

Figure 7.3: Nofib Performance Comparison between foldr/build and Stream Fusion with the $concatMap$->$flatten$ Transformation

Merge This benchmark involves merging two streams with the same state type, as discussed in Section 7.5.1.

$$\begin{array}{l}
f\ n = \mathit{foldl'}\ (+)\ 0\ (\mathit{concatMap}\ (\lambda x \rightarrow \textbf{case}\ (\mathit{odd}\ x)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{True} \rightarrow \mathit{enumFromTo}\ 1\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{False} \rightarrow \mathit{enumFromTo}\ 2\ x) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{enumFromTo}\ 1\ n))
\end{array}$$

7.6.2 Nofib Suite

In order to evaluate the transformation on real Haskell programs, it is applied to a subset of GHC's **nofib** benchmarking suite [Partain, 1993]. The **nofib** suite is the standard by which other GHC optimizations are measured before inclusion in the compiler.

The "imaginary" subset of the suite is targeted. This limits the number of necessary stream combinator implementations (aside from $concatMapS$ and *flattenS*) to those used by this subset. This is a practical engineering issue that can be solved in a more mature fusion system, and is unrelated to the $concatMap$ transformation itself.

Figure 7.3 summarizes the results for the selected programs. In summary, programs experience $\approx 15\%$ speedup over foldr/build on average, with an $\approx 10\%$ reduction in allocation. In the following discussion, the Stream Fusion system which includes the $concatMap$->$flatten$ transformation developed in this case study is referred to as HERMITSF, to distinguish it from existing Stream Fusion.

The gains for `bernouilli`, `integrate`, and `x2n1` are due to Stream Fusion itself, and HERMITSF provides no additional benefit. The `integrate` program features no calls to $concatMap$ at all, but makes heavy use of the $zipWith$ combinator. Similarly, `x2n1` is a true micro-benchmark, consisting of a single mapping operation inside a strict left fold. Though the results are not presented here for brevity, Stream Fusion performs significantly worse than foldr/build on each of the other programs. Allocation increases by 45.7% on average, and 836.7% in the worst case. Runtime is equivalent on average, but increases by 66.1% in the worst case. This is the penalty Stream Fusion pays on $concatMap$-heavy code. HERMITSF always outperforms Stream Fusion alone.

The `gen_regexps` program is an ideal case for HERMITSF, as it makes heavy use of both $foldl$ and $concatMap$. Previous systems could fuse one of these combinators, but HERMITSF can fuse both.

Programs which make use of explicit recursion on lists, rather than combinators, tend to be slowed by HERMITSF. Both `paraffins` and `wheel-sieve1` make use of functions which explicitly accept and return lists, and are also recursive. These recursive functions will not be inlined by GHC, preventing the $stream$ and $unstream$ combinators from coming together and being eliminated. This results in many extra conversions between lists and streams. The resulting allocation is high, even if gains elsewhere improve runtime. To solve this, Stream Fusion systems typically include extra RULES to “back out” unfused stream combinators at the end of the optimization process, converting them back to their list counterparts. HERMITSF currently does not focus on doing this.

These initial results are promising, and demonstrate the viability of the approach. Applying the optimization to the full `nofib` suite remains future work. Compilation time, currently 1.5-10× that of `ghc -O2`, could also be improved.

7.6.3 Performance Advantages of `concatMap`

Somewhat surprisingly, *concatMap* can provide performance advantages over *flatten*. When using *flatten*, the programmer must carefully consider low-level performance issues because they are, in essence, writing a hand-fused inner-loop. By using *concatMap*, the inner loop can instead be constructed from existing stream combinators, which presumably are already efficiently implemented. In this case, modularity makes it easier to get good performance.

To illustrate, this section benchmarks a pair of equivalent `vector` pipelines (one using *concatMap*, the other *flatten*) and examines the resulting Core. Using list combinators, the specification of the pipeline is:

```
spec :: Int → Int
spec n = foldl (+) 0 [i | x ← [1 .. n], i ← [1 .. x]]
```

The `vector` code is morally equivalent, though with added strictness and use of `vector`'s *enumFromStepN* in place of *enumFromTo* in order to reflect the sort of code a user of the `vector` library would write in practice.

```
cmap :: Int → Int
cmap n = foldl′ (+) 0 (concatMap (λ(!x) → enumFromStepN 1 1 x)
                                  (enumFromStepN 1 1 n))
flat :: Int → Int
flat !n = foldl′ (+) 0 (flatten mkS stp Unknown (enumFromStepN 1 1 n))
  where
    mkS !x = (1, x)
    stp (!i, !max)
      | i <= max  = Yield i (i + 1, max)
      | otherwise = Done
```

These functions are first benchmarked with standard Stream Fusion optimizations using `criterion` [O'Sullivan] to establish a baseline. As can be seen in Figure 7.4, *cmap* is consistently 10x slower than *flat*. Examining the Core reveals that *flat* results in a single wrapper

n	*flat*	*cmap*			
		SF	ratio	HERMITSF	ratio
5000	9.7ms	100.3ms	10.3x	8.8ms	0.91x
10000	39.3ms	395.6ms	10.1x	35.0ms	0.89x
20000	155.9ms	1603.8ms	10.3x	139.3ms	0.89x

Figure 7.4: Optimizing Equivalent Stream Pipelines for the `vector` Package.

function to unbox the input and box the result, along with a tight recursive worker on unboxed integers.

Generally, uses of *concatMap* result in residual *Step* constructors, indicating fusion is incomplete. GHC actually manages to fuse away the *Step* constructors in *cmap*, as this code is very simple. However, the resulting inner loop involves both boxed integers and a tuple argument.

Applying the *concatMap* transformation results in nearly equivalent performance. In fact, *cmap* is now consistently 11% *faster* than *flat*! Examining the Core for the inner loop of each function reveals why. The bodies of the loops consist only of tail calls on unboxed integers, but the bounds test in the inner loop is different. In *cmap*, the iterator is compared to zero, whereas in *flat*, the iterator is compared to a max bound.

Indeed, this behavior is exactly what the generator function for *flat* implements. Changing this generator to implement the same algorithm as *enumFromStepN*, results in comparisons to zero.

```
step (!i, !max)
  | max > 0   = Yield i (i + 1, max − 1)
  | otherwise = Done
```

However, this actually makes *flat* $18\times$ slower! Examining the Core again, the inner loop now involves *Integer* arguments, a clue to what is happening. The result type of *flat* only constrains the type of i. Since the state of the stream is existentially quantified, and *max* is no longer compared to i, the type of *max* is defaulted to *Integer* rather than *Int*. Ascribing a type fixes the problem, resulting in *flat* being as fast as *cmap*.

```
step (!i, !max)
  | max > (0 :: Int) = Yield i (i + 1, max − 1)
  | otherwise        = Done
```

The fact that the programmer must consider such performance issues each time she writes a generator function is exactly what makes *flatten* burdensome to use. Using *concatMap* (properly optimized) allows them to take advantage of the hard work done by the `vector` library writers, who have heavily optimized their enumerating stream producer. Thus, in practice, transforming $concatMap$ to $flatten$ can result in better performance than direct uses of $flatten$ itself.[2]

7.7 ADPfusion

ADPfusion [Höner zu Siederdissen, 2012] is a library designed to simplify the implementation of complex dynamic programming algorithms in Haskell. It targets both single- and multi-tape linear and context-free grammars. ADPfusion can be used to implement complex parsing problems, such as weakly synchronized grammars for machine translation [Burkett et al., 2010] in computational linguistics or interacting ribonucleic acid structure prediction as considered by Huang et al. [2009]. As ADPfusion employs CYK-style parsing [Grune and Jacobs, 2008] that lends itself well to a low-level "table-filling" implementation, the resulting programs will perform close to equivalent `C` code, while being implemented at a much higher level of abstraction.

ADPfusion makes index calculations implicit. Production rules are combined by a small set of combinators, producing a parsing function from an index to a set of (co-)optimal parses. Lifting index calculations from explicit manipulations by the user to combinators makes it less likely that bugs appear, while the type system keeps evaluation functions and production rules in sync.

Questions of how to develop algorithms like these lead to the development of an algebraic framework that allows users to "multiply" dynamic programming algorithms in a meaningful way [Höner zu Siederdissen et al., 2013]. The resulting algorithms (grammars) are naturally of the multi-tape variety and the grammar definitions call for an automated embedding in an efficient framework. ADPfusion is used as the target DSL in this case to give efficient code.

[2] A more recent build of GHC eliminates the 11% disparity in the original code. This is the result of new boolean primitive operations which were added after this section was written. While this specific example is no longer strictly true, it is illustrative of the general problem with optimizing `flatten` by hand. This issue accounts for the performance gains made by `concatMap` relative to `flatten` in Section 7.7.

Using `concatMap` instead of `flatten`

The availability of $concatMap$ helps control the complexity of ADPfusion's underlying Stream Fusion framework by simplifying the design of specialized (non-)terminal symbols for formal grammars.

To understand how ADPfusion uses $flatten$, consider the parses for the production rule $S \rightarrow SS$, given in set notation:

$$
\begin{array}{lll}
{}_iS_j \rightarrow \{ (S^{ik}, S^{lj}) \mid & k & \leftarrow \{ i \ \ldots \ j \} \\
 & , S^{ik} & \leftarrow {}_iS_k \\
 & , l & \leftarrow \{ k \} \\
 & , S^{lj} & \leftarrow {}_lS_j \}
\end{array}
$$

That is, partial parses are generated from left to right for each production rule. All parses, except the final one, make use of the $flatten$ combinator to extend the current stream of partial parses and current index state with the parses for the current symbol. As all indices are already fixed when considering the right-most symbol in a production rule, only a single parse is generated in such a case (denoted $l \leftarrow \{k\}$).

This explanation assumes that, for fixed indices (say ${}_iS_k$), a non-terminal produces only a single parse. When only a single optimal result is required, this is actually the desired behaviour. In cases where co- or sub-optimal parses are required, non-terminals produce multiple results, thereby requiring an *additional* $flatten$ operation for each non-terminal, leading to the full notation above.

Thus, the $flatten$ function is used extensively in ADPfusion. Each new (non-)terminal requires up to two $flatten$ operations. Symbols on the right-hand sides of production rules admit multiple parses. Nesting further, in multi-tape settings, $flatten$ is used to combine parses from individual tapes.

ADPfusion must handle a fixed, but arbitrary, number of input tapes and allow the user to integrate new (non-) terminal parsers easily with the existing library. The ability to fuse applications of $concatMap$, instead of having to rely on $flatten$, allows for the replacement of the complex system of recursive calls to $flatten$ with simpler calls to $concatMap$.

As an example, the following (simplified) code is used for multi-tape indices. A Subword $(i{:}.j)$ denotes the substring currently parsed. The highest subword index is removed from the index stack, followed by a recursive call to *tableIx* to calculate inner indices. Using $flatten$, the set of indices is expanded and the index stack extended (with a payload z and a temporary stack a).

```
class TableIx i where
    tableIx :: i → Stream (z, a, i) → Stream (z, a, i)
instance TableIx is ⇒ TableIx (is :. Subword) where
    tableIx (is :. Subword (i :. j))
        = flatten mk step ∘ tableIx is ∘ map (λ(z, a, (ns :. n)) → (z, (a :. n), ns))
  where
    mk (z, a :. Subword (k :. l), ns) = (z, a, ns, l, l)
    step (z, a, ns, k, l)
        | l > j       = Done
        | otherwise = Yield (z, a, (ns :. Subword (k :. l))) (z, a, ns, k, l + 1)
```

A fusable version of $concatMap$ simplifies the implementation.

```
instance TableIx is ⇒ TableIx (is :. Subword) where
    tableIx (is :. Subword (i :. j))
        = concatMap f ∘ tableIx ∘ map (λ(z, a, (ns :. n)) → (z, (a :. n), ns))
  where
    f (z, a :. Subword (k :. l), ns) =
        map (λm → (z, a, ns :. Subword (m :. j))) (enumFromStepN l 1 (j − l + 1))
```

This simplicity becomes more pronounced as *TableIx* instances statically track additional boundary conditions, maximal yield sizes, and special table conditions which have been omitted here, for clarity.

Performance of ADPfusion

ADPfusion was re-implemented using $concatMap$ in order to test the performance of the $concatMap$ transformation on real code. Since ADPfusion is built upon a finite, fixed set of functions (mainly the stream-generating *MkStream* type class), HERMIT optimizations can be targeted to exactly the offending calls.

Table 7.1 summarizise these results for various input lengths. All applications of $concatMap$ are rewritten, the *Step* data constructors are successfully eliminated, and unboxing (especially of loop counters) of all variables occurs. The HERMIT-optimized version ($\texttt{ADPfusion}_{\texttt{hermit}}$) is on par with the version using $flatten$. Using $concatMap$ without HERMIT optimization leads to a slowdown of ≈6-8× compared to both optimized versions. Runtimes for the `C` reference

input length	400	600	800	1 000
ADPfusion$_{\texttt{hermit}}$	0.03s	0.10s	0.22s	0.41s
ADPfusion$_{\texttt{flatten}}$	0.03s	0.10s	0.22s	0.44s
ADPfusion$_{\texttt{concatMap}}$	0.19s	0.64s	1.56s	3.15s
C	0.01s	0.04s	0.09s	0.20s

Table 7.1: Runtime in Seconds for the `Nussinov78` Algorithm using `ADPfusion` and `C`.

implementation are included for comparision. (Options used: `ghc -O2 -fllvm`, resp. `gcc -O3`, measurements performed on an Intel Core i5-3570K).

7.8 Conclusions and Future Work

This case study uses HERMIT to specify a custom GHC plugin which implements a transformation for fusing Stream Fusion's $concatMap$. A key benefit of HERMIT is that it lowers the barrier to implementing such transformations. HERMIT allows a user to rapidly prototype an optimization, interactively exploring the transformation in action *during compilation*.

The transformation extends one originally proposed by Coutts [2010]. The value of the outer stream is stored in a modified inner-stream state so it is available to the inner-stream generator. The transformation is also extended to monadic streams. These extensions require the manipulation of syntactic constructs of the GHC Core representation of the program, something that is currently inexpressible by GHC's RULES rewrite system.

Several subtleties were uncovered by implementing the transformation itself. This led to the specific simplification heuristics in Algorithm 1, which are required to enable the transformation in practice. Each step in this algorithm, and their specific ordering, was discovered by exploring the optimization interactively with HERMIT.

Aggressive call-pattern specialization is crucial for Stream Fusion. The inlining heuristic in Section 7.5.3 was key to achieving fusion. This suggests that modifying GHC's implementation of specialization to look through let bindings may be profitable in general, and is worth pursuing as future work. However, if such a modification is not found to be generally useful, it can continue to be applied by HERMIT for this specific optimization.

A number of steps in the transformation have the potential to duplicate work (or allocation). The impact of this duplication remains unquantified, though the performance measurements in Section 7.6 show the transformation is generally an optimization. Speedups of micro-benchmarks targeted by the transformation are considerable. This is in no small part due to the fact that existing Stream Fusion frameworks perform so poorly on $concatMap$. As Figure 7.2 illustrates, lists often perform better than vectors in $concatMap$-heavy code because foldr/build is so good at fusing $concatMap$.

Results from the nofib benchmark suite are mostly positive with speedup of $\approx 15\%$ and a $\approx 10\%$ reduction in allocation, on average. Some programs experience large speedups or slowdowns. The slowdowns are the result of the limited set of stream combinators implemented, along with other practical implementation issues, rather than the $concatMap$ transformation itself. Making a production quality system that can be "always on" remains future work. Nevertheless, the HERMIT-based implementation presented in this study can be selectively enabled when it is determined to be beneficial.

This performance is already available to users of $flatten$, but at considerable cost in implementation complexity. This complexity often leads to sub-optimal uses of $flatten$. Users of a fully fused $concatMap$ can more readily take advantage of the hard work of library writers.

The ADPfusion library relies on Stream Fusion on vectors to achieve good performance. The preliminary results we presented in Sec. 7.7 suggest that using HERMIT to optimize $concatMap$ will reduce the implementation complexity of the library considerably, without unduly sacrificing performance.

Chapter 8

Case Study: Making a Century

To assess how well HERMIT supports calculational programming, this case study mechanizes a program derivation from the chapter *Making a Century* in *Pearls of Functional Algorithm Design* [Bird, 2010, Chapter 6]. The book is entirely dedicated to reasoning about Haskell programs, with each chapter calculating an efficient program from an inefficient specification program. Thus, unlike the case study in Chapter 6, the goal is *program transformation*, not proving properties. However, many of the transformations used have preconditions, and thus there are several proof obligations along the way, making this case study more than just an optimization pass like the one in Chapter 7.

The program in *Making a Century* computes the list of all ways the addition and multiplication operators can be inserted into the list of digits $[1\,..\,9]$, such that the resultant expression evaluates to 100. For example, one possible solution is:

$$12 + 34 + 5 \times 6 + 7 + 8 + 9 = 100$$

The details of the program are not overly important to the case study, and the interested reader should consult the textbook for details [Bird, 2010, Chapter 6]. What is important is that the derivation of an efficient program involves a substantial amount of equational reasoning, and the use of a variety of proof techniques, including fold/unfold transformation [Burstall and Darling-

Fold Fusion	$\forall f\ g\ h\ a\ b\,.$	$(f \bot \equiv \bot \ \wedge \ f\ a \equiv b \ \wedge \ \forall x\ y\,.\ f\ (g\ x\ y) \equiv h\ x\ (f\ y))$ $\Rightarrow$ $f \circ foldr\ g\ a \equiv foldr\ h\ b$
Lemma 6.2		$filter\ (good \circ value) \equiv filter\ (good \circ value) \circ filter\ (ok \circ value)$
Lemma 6.3	$\forall x\,.$	$filter\ (ok \circ value) \circ extend\ x$ $\equiv$ $filter\ (ok \circ value) \circ extend\ x \circ filter\ (ok \circ value)$
Lemma 6.4	$\forall x\,.$	$map\ value \circ extend\ x \equiv modify \circ map\ value$
Lemma 6.5	$\forall f\ g\,.$	$fst \circ fork\ (f, g) \equiv f \ \wedge \ snd \circ fork\ (f, g) \equiv g$
Lemma 6.6	$\forall f\ g\ h\,.$	$fork\ (f, g) \circ h \equiv fork\ (f \circ h, g \circ h)$
Lemma 6.7	$\forall f\ g\ h\ k\,.$	$fork\ (f \circ h, g \circ k) \equiv cross\ (f, g) \circ fork\ (h, k)$
Lemma 6.8	$\forall f\ g\,.$	$fork\ (map\ f, map\ g) \equiv unzip \circ map\ (fork\ (f, g))$
Lemma 6.9	$\forall f\ g\,.$	$map\ (fork\ (f, g)) \equiv zip \circ fork\ (map\ f, map\ g)$
Lemma 6.10	$\forall f\ g\ p\,.$	$map\ (fork\ (f, g)) \circ filter\ (p \circ g)$ $\equiv$ $filter\ (p \circ snd) \circ map\ (fork\ (f, g))$

Figure 8.1: Main Lemmas in the ‘Making a Century’ Case Study.

ton, 1977], structural induction (Section 5.10.5), fold fusion [Meijer et al., 1991], and numerous auxiliary lemmas.

Rather than present every proof in detail, this case study focuses on a representative extract, and then discuss the aspects of the mechanization that proved challenging. The HERMIT scripts for the complete case study are available online [Farmer et al., 2015].

8.1 HERMIT Scripts

After creating a Haskell file containing the function definitions from the textbook, the next task is to introduce the lemmas used in the equational-reasoning steps. The main lemmas (specifically, those that are named in the textbook) are displayed in Figure 8.1, giving a rough idea of their complexity. The majority of these lemmas are equivalences between expressions and can be introduced via (inactive) rewrite rules in the Haskell source file (see Section 5.4). The notable exception is the fold-fusion law, which is constructed and introduced using a custom KURE transformation. Lemma 6.5 is also a composite lemma, but it was more convenient to introduce a pair of lemmas rather than constructing an explicit conjunction.

$$\begin{array}{ll}
 & \mathit{unzip} \cdot \mathit{map}\ (\mathit{fork}\ (f, g)) \\
= & \quad \{\text{definition of } \mathit{unzip}\ \} \\
 & \mathit{fork}\ (\mathit{map}\ \mathit{fst}, \mathit{map}\ \mathit{snd}) \cdot \mathit{map}\ (\mathit{fork}\ (f, g)) \\
= & \quad \{(6.6) \text{ and } \mathit{map}\ (f \cdot g) = \mathit{map}\ f \cdot \mathit{map}\ g\ \} \\
 & \mathit{fork}\ (\mathit{map}\ (\mathit{fst} \cdot \mathit{fork}\ (f, g)), \mathit{map}\ (\mathit{snd} \cdot \mathit{fork}\ (f, g))) \\
= & \quad \{(6.5)\ \} \\
 & \mathit{fork}\ (\mathit{map}\ f, \mathit{map}\ g)
\end{array}$$

(a) Textbook extract.

```
-- ∀ f g . fork ((,) (map f) (map g)) = (.) unzip (map (fork ((,) f g)))
    forall-body ; eq-rhs
-- (.) unzip (map (fork ((,) f g)))
    one-td (unfold 'unzip)
-- (.) (fork ((,) (map fst) (map snd))) (map (fork ((,) f g)))
    lemma-forward "6.6" ; any-td (lemma-forward "map-fusion")
-- fork ((,) (map ((.) fst (fork ((,) f g)))) (map ((.) snd (fork ((,) f g)))))
    one-td (lemma-forward "6.5a") ; one-td (lemma-forward "6.5b")
-- fork ((,) (map f) (map g))
```

(b) HERMIT script.

Figure 8.2: Comparing the Textbook Calculation with the HERMIT Script for Lemma 6.8.

Irrespective of how they were introduced, the same approach was taken to proving each lemma: the proof was performed in HERMIT's interactive mode until successful and then the final proof was saved as a script that could be invoked thereafter. Finally, the main program transformation ($\mathit{solutions}$) was developed interactively, invoking the saved auxiliary proof scripts as necessary. Roughly half of the proofs in this case study were transliterations of proofs from the textbook, and half were proofs that were not included in the text and had to be developed in HERMIT (see Section 8.3). Both sets of proofs proceeded in a similar manner, but with more experimentation and backtracking during the interactive phase for the latter set.

As an example, compare the proofs of Lemma 6.8. Figure 8.2a presents the proof extracted verbatim from the textbook [Bird, 2010, Page 36], and Figure 8.2b presents the corresponding HERMIT script. Note that lines beginning "--" in a HERMIT script are *comments*, and for readability have been typeset differently to the `monospace` HERMIT code. These comments represent the current expression between transformation steps, and correspond to the output of the HERMIT

REPL when performing the proof interactively. When generating a HERMIT proof script after an interactive session, HERMIT can automatically insert these comments if desired. The content of the comments can be configured by HERMIT's various pretty-printer modes — in this case they omit the type arguments (as in Section 4.1) to make the correspondence with the textbook extract clearer.

The main difference between the two calculations is that, in HERMIT, one must specify where in the term to apply a rewrite, and in which direction lemmas are applied. In contrast, in the textbook the lemmas to be used or functions to be unfolded are merely named, relying on the reader to be able to deduce how it was applied.

In this proof, and most others in the case study, the HERMIT scripts are about as clear, and not much more verbose, than the textbook calculations. There is one exception though: manipulating terms containing adjacent occurrences of the function-composition operator.

8.2 Associative Operators

On paper, associative binary operators such as function composition are typically written without parentheses. However, in GHC Core, operators are represented by nested application nodes in an abstract syntax tree, with no special representation for associative operators. Terms that are equivalent semantically because of associativity properties can thus be represented by different trees. Consequently, it is sometimes necessary to perform a tedious restructuring of the abstract syntax tree before a transformation can match the term.

For function composition, one way to avoid this problem is to work with eta-expanded terms and unfold all occurrences of the composition operator, as this always produces an abstract syntax tree consisting of a left-nested sequence of applications. However, the goal of this case study was to match the textbook proofs, which are written in a point-free style, as closely as possible. Thus, this unfolding was not performed.

comp-id-L	$\forall f\,.$	$id \circ f \equiv f$
comp-id-R	$\forall f\,.$	$f \circ id \equiv f$
comp-assoc	$\forall f\ g\ h\,.$	$(f \circ g) \circ h \equiv f \circ (g \circ h)$
comp-assoc4	$\forall f\ g\ h\ k\ l\,.$	$f \circ (g \circ (h \circ (k \circ l))) \equiv (f \circ (g \circ (h \circ k))) \circ l$
map-id		$map\ id \equiv id$
map-fusion	$\forall f\ g\,.$	$map\ (f \circ g) \equiv map\ f \circ map\ g$
map-strict	$\forall f\,.$	$map\ f\ undefined \equiv undefined$
zip-unzip		$zip \circ unzip \equiv id$
filter-strict	$\forall f\,.$	$filter\ f\ undefined \equiv undefined$
filter-split	$\forall p\ q\,.$	$(\forall x\,.\ q\ x \equiv \mathsf{False} \Rightarrow p\ x \equiv \mathsf{False}) \Rightarrow filter\ p \equiv filter\ p \circ filter\ q$

Figure 8.3: Auxiliary Lemmas Proved in HERMIT during the ‘Making a Century’ Case Study.

More generally, rewriting terms containing associative (and commutative) operators is a well-studied problem [e.g. Dershowitz et al., 1983, Kirchner and Moreau, 2001, Braibant and Pous, 2011], and it remains as future work to provide better support for manipulating such operators in HERMIT.

8.3 Assumed Lemmas in the Textbook

As is common with pen-and-paper reasoning, several properties that are used in the textbook are assumed without a proof being given. This included some of the named lemmas from Figure 8.1, as well as several auxiliary properties, some explicit and some implicit (Figure 8.3). While performing reasoning beyond that presented in the textbook was not intended to be part of the case study, proofs of these properties were nevertheless attempted in HERMIT.

Of the assumed named lemmas, the fold-fusion law has a straightforward inductive proof, which can be encoded fairly directly using HERMIT’s built-in structural induction. Lemmas 6.5, 6.6, 6.7 and 6.10 are properties of basic function combinators, and proving them mostly they consisted of inlining definitions and simplifying the resultant expressions, with the occasional basic use of induction. The same was true for the auxiliary lemmas, which are listed in Figure 8.3. Systematic proofs such as these are ripe for mechanization, and HERMIT provides several strategies that perform a suite of basic simplifications to help with this. Consequently, the proof scripts were short and concise.

Lemmas 6.2, 6.3 and 6.4 were more challenging. For Lemma 6.2, it was helpful to introduce and prove the `filter-split` auxiliary lemma (Figure 8.3), which captures the essence of the key optimization in the case study. After this, the proof was fairly straightforward.

Lemmas 6.3 and 6.4 appear to be non-trivial properties without obvious proofs, so they were not proven in HERMIT. This did not inhibit the rest of the case study however, as HERMIT allows a lemma to be taken as an assumption, which can then be used without being proved. If such assumed lemmas are used in a calculation, HERMIT will issue a compiler warning. This ability to assume lemmas can be disabled by a HERMIT option, allowing the user to ensure that only proved lemmas are used.

Finally, the simplification of the definition of expand is stated in the textbook without presenting any intermediate transformation steps [Bird, 2010, Page 40]. It is not obvious what those intermediate transformation steps would be, and thus this simplification was not encoded in HERMIT.

8.4 Constructive Calculation

There was one proof technique used in the textbook that HERMIT does not directly support: calculating the definition of a function from an *indirect* specification. Specifically, the textbook postulates the existence of an auxiliary function (expand), uses that function in the conclusion of the fold-fusion rule, and then calculates a definition for that function from the indirect specification given by the fold-fusion pre-conditions. HERMIT is based around transforming (and proving properties of) existing definitions, and does not support this style of reasoning; so this calculation could not be replicated. However, the calculation was *verified* by working in reverse: starting from the definition of expand, the use of the fold-fusion law could be validated by proving the corresponding pre-conditions.

Calculation	Textbook Lines	HERMIT Commands	
		Transformation	Navigation
Fold Fusion	assumed	24	28
Lemma 6.2	assumed	11	7
Lemma 6.3	assumed	assumed	
Lemma 6.4	assumed	assumed	
Lemma 6.5	assumed	4	4
Lemma 6.6	assumed	2	2
Lemma 6.7	assumed	2	1
Lemma 6.8	7	5	6
Lemma 6.9	1	4	4
Lemma 6.10	assumed	23	18
solutions	16	6	8
expand	19	18	18

Table 8.1: Comparison of Calculation Sizes in ‘Making a Century’.

8.5 Calculation Sizes

As demonstrated by Figure 8.2, the HERMIT proof scripts are roughly the same size as the textbook calculations. It is difficult to give a precise comparison, as the textbook uses both formal calculation and natural language. Table 8.1 presents some statistics, but this is only intended to give a rough approximation of the scale of the proofs. The size of the two main calculations (transforming *solutions* and deriving *expand*) are given, as well as the sizes of the proofs of the named auxiliary lemmas. Textbook lines measures lines of natural language reasoning as well as lines of formal calculation, but not definitions, statement of lemmas, or surrounding discussion. In the HERMIT scripts, the number of transformations applied, and the number of navigation and strategy combinators used to direct the transformations to the desired location in the term are measured. HERMIT commands for stating lemmas, loading files, switching between transformation and proof mode, or similar, are not measured, as they are considered comparable to the surrounding discussion in the textbook. To get a feel for the scale of the numbers given, compare Lemma 6.8 in Table 8.1 to the calculation in Figure 8.2.

8.6 Reflections

Overall, mechanizing the textbook calculations was fairly straightforward, and it was pleasing that most of the steps in the textbook translate directly into an equivalent HERMIT command. The only annoyance was the need to occasionally manually apply associativity lemmas (see Section 8.2) so that the structure of the term would match the transformation being applied.

While having to specify where in a term each lemma must be applied does result in more complicated proof scripts than in the textbook, this is not necessarily detrimental. Rather, a pen-and-paper proof that does not specify the location is passing on the work to the reader, who must determine for herself where, and in which direction, the lemma is intended to be applied. Furthermore, when desired, strategic combinators such as `any-td` can be used to avoid specifying precisely which sub-term the lemma should be applied to by applying it anywhere it matches.

This case study also uncovered one error in the textbook. Specifically, the inferred type of the *modify* function [Bird, 2010, Page 39] does not match its usage in the program. The definition of *modify* should include a $concatMap$, which would correct the type mismatch and give the program its intended semantics. This is the version the case study uses. However, this cannot be claimed as HERMIT detecting an error in a pen-and-paper proof, as it was caught by GHC's type checker, not by HERMIT.

Chapter 9

Applications

HERMIT has been applied to a number of projects in addition to the large case studies in this dissertation. The tool support it provides for working within GHC make it valuable for prototyping optimization passes and building other experimental compiler features. HERMIT's reasoning features support the exploration of equational reasoning techniques on real programs. This chapter surveys some of the projects which make use of HERMIT, reflecting on HERMIT's role.

9.1 Worker/Wrapper Transformation

The Worker/Wrapper transformation [Gill and Hutton, 2009, Peyton Jones and Launchbury, 1991] changes how a (typically recursive) function operates using only locally verifiable assumptions. The transformation is a general form of data-refinement [Hoare, 1972]. Worker/wrapper has been used as a mechanism to implement strictness analysis optimizations [Jones and Partain, 1993], argument transposition [Launchbury and Sheard, 1995], constructor specialization [Peyton Jones, 2007], memoization [Michie, 1968], and CPS translation [Sussman and Steele, 1975, Appel, 1992].

The Worker/Wrapper transformation is illustrated in Figure 9.1. Solid arrows signify function calls, and dashed arrows signify the application of specified refinements. The boxes are computations that have a type, given as an incoming arrow, and require the use of a second computation,

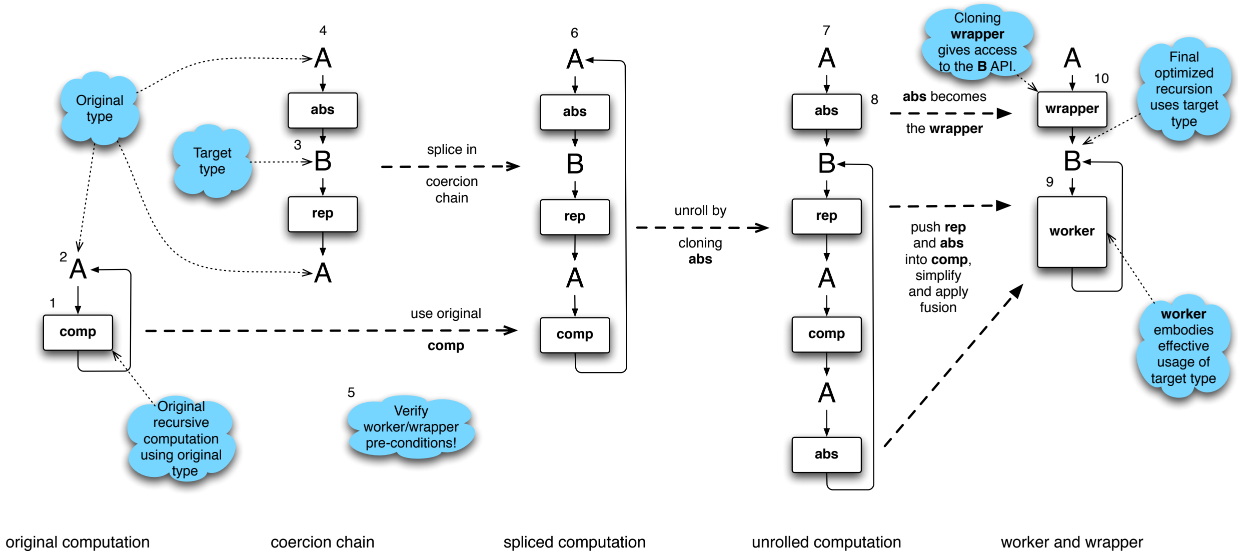

Original type
Target type
Original recursive computation using original type
Verify worker/wrapper pre-conditions!
Cloning wrapper gives access to the B API.
Final optimized recursion uses target type
worker embodies effective usage of target type
1
2
3
4
5
6
7
8
9
10
A
B
comp
abs
rep
wrapper
worker
splice in coercion chain
use original comp
unroll by cloning abs
abs becomes the wrapper
push rep and abs into comp, simplify and apply fusion
original computation
coercion chain
spliced computation
unrolled computation
worker and wrapper

Figure 9.1: The Worker/Wrapper Transformation

given as an outgoing arrow. As such, these computational boxes can be considered as representing functions with higher-order arguments. **A** and **B** should be read as the type of a computation.

The **original computation**[1] is a recursive function, notated with an arrow back to the type of the computation, **A**[2]. (Superscripts in this paragraph refer to Figure 9.1.) The Worker/Wrapper transformation operates as follows: A target type[3] for the recursion is selected, here called **B**. A **coercion chain**[4] is constructed from two higher-order functions, that coerce from **A** to **B**, and back to **A**, using the standard data-refinement terminology of **abs** and **rep** [Hoare, 1972]. The Worker/Wrapper local preconditions are verified[5]; one example is when **abs** and **rep** form an identity over **A**. The coercion-chain can now be spliced into the original recursion back-edge[6], replacing the original computation. The **abs** definition can now be unrolled a single step[7], which exposes **abs** as a non-recursive component of the overall computation[8]. This new formulation can be transformed and optimized using local, well understood algebraic transformations and fusion rules into an efficient *worker*[9] that operates over **B**, and a *wrapper*[10] that allows original users of **A**'s API to call the new function.

The three possible pre-conditions for Worker/Wrapper are:

$$\begin{array}{lll}\text{(A)}\ \mathit{abs} \circ \mathit{rep} & \equiv & \mathit{id}_A \\ \text{(B)}\ \mathit{abs} \circ \mathit{rep} \circ \mathit{comp} & \equiv & \mathit{comp} \\ \text{(C)}\ \mathit{fix}\ (\mathit{abs} \circ \mathit{rep} \circ \mathit{comp}) & \equiv & \mathit{fix}\ \mathit{comp}\end{array}$$

All three follow from the formalization, with each pre-condition directly implying the subsequent, weaker pre-condition. The third pre-condition can actually be observed inside Figure 9.1, where the splicing is a use of this final rule, applied right-to-left. The intuition behind all the pre-conditions is that they ensure, specifically for the function `comp`, that the alternative type **B** can safely be used.

Following from these pre-conditions, and other well understood equational reasoning principles, is a factorization transformation which states how the wrapper is defined in terms of the new worker, along with a set of equations for calculating the definition of the worker itself. In these equations, $\mathit{fix}\ f$ is the original computation, and $\mathit{fix}\ g$ is the new worker, where fix is the function for definining the least fixed point of another function.

$$\begin{array}{lll}
Factorization: & & \\
\quad \mathit{fix}\ f & \equiv & \mathit{abs}\ (\mathit{fix}\ g) \\
(1\alpha)\ g & \equiv & \mathit{rep} \circ f \circ \mathit{abs} \\
(1\beta)\ \mathit{fix}\ g & \equiv & \mathit{fix}\ (\mathit{rep} \circ f \circ \mathit{abs}) \\
(2\alpha)\ \mathit{rep} \circ f & \equiv & g \circ \mathit{rep}\ \wedge\ \textbf{strict}\ \ \mathit{rep} \\
(2\beta)\ \mathit{fix}\ g & \equiv & \mathit{rep}\ (\mathit{fix}\ f) \\
(3\alpha)\ \mathit{abs} \circ g & \equiv & f \circ \mathit{abs}
\end{array}$$

Part of the appeal of Worker/Wrapper is that the factorization equation is simple, and that the derivation relies on a single pre-condition which can itself be verified by simple equational reasoning. The transformation can thus be performed semi-formally, usually by hand, to justify a data refinement, or even calculate a previously unknown definition for the worker.

This simplicity makes Worker/Wrapper an obvious target for HERMIT. In fact, the transformation was the original motivating example of the HERMIT project. The two primary tasks required to perform Worker/Wrapper are to verify one of the pre-conditions and to simplify away the representation-changing functions (abs and rep) after factorization. Both tasks are supported by HERMIT's mechanized reasoning and transformation capabilities.

HERMIT provides two rewrites for the factorization step. The first, called `split-1-beta` implements the factorization using rule (1β) above. The second, called `split-2-beta` implements the factorization using rule (2β). In each case, a lemma corresponding to pre-condition (C) is generated as a proof obligation. If the user has previously proven pre-conditions (A) or (B), these lemmas can be used to prove (C).

The other rules are not implemented because they either follow directly from these two, or are not definitional in g, requiring constructive calculation (as defined in Section 8.4). Constructive calculation is a style of reasoning not supported by HERMIT. The fact that these other rules are not implemented does not reduce the power of Worker/Wrapper in HERMIT. A particular derivation may begin more conveniently using a certain rule, but they are equivalent.

All of the known Worker/Wrapper derivations from the literature have been mechanized using HERMIT. The derivations are included as examples with the HERMIT package. In his thesis, Torrence [2015] uses HERMIT to mechanize a series of Worker/Wrapper transformations which systematically refine an executable specification of Conway's Game of Life [Adamatzky, 2010] into an efficient implementation. This is the first time Worker/Wrapper has been mechanized on a complete program.

HERMIT currently only provides specialized support for the factorization step, relying on its other general purpose transformations and the user for proving the pre-condition and fusing away the representation changing functions. It is conceivable that patterns in both the proof and simplification could be exploited to offer an even higher level of abstraction, though this remains future work.

9.2 Optimizing SYB is Easy!

In Adams et al. [2014], HERMIT is used to implement a domain-specific optimization pass which removes the overhead of generic traversals for the Scrap Your Boilerplate library [Lämmel and Peyton Jones, 2003]. Scrap Your Boilerplate (SYB) is the most widely used generic-programming library in the Haskell community, permitting the concise expression of data-type generic traversals. It also happens to be the slowest. The runtime type comparison necessary during SYB traversals makes them an order-of-magnitude slower than equivalent non-generic, hand-written traversals [Adams and DuBuisson, 2012, Sculthorpe et al., 2014]. Previous optimization techniques were unable to eliminate this performance penalty.

The optimization which HERMIT implements takes advantage of domain-specific knowledge about the structure of SYB traversals to direct a limited form of partial evaluation [Jones et al., 1993]. The work itself focuses on formalizing the optimization technique and evaluating it empirically on a selection of benchmarks. HERMIT was used to actually implement the optimization in order to perform the evaluation, and this section reflects on HERMIT's role in its development, as well as the project's effect on HERMIT itself.

The intuition for this optimization is that the evaluation of the run-time type comparisons can actually be performed at compile-time when the type of the structure being traversed is statically known. The structure type is most often known at the application site of the traversal, though simply ascribing a concrete type to a traversal is enough. Performing this evaluation is a matter of unfolding SYB combinators and performing other local transformations such as case reduction, augmented with rewrites for statically evaluating certain primitive operations that compare run-time type representations.

HERMIT's interactive mode was used to explore these necessary evaluation steps on several examples. Eventually, a pattern emerged. Run-time type comparisons always occurred in computations whose types featured certain 'undesirable' types. These types include the *Data* and *Typeable* dictionaries used to implement SYB's type-safe cast interface, as well as the types related to run-time type representations. Directing evaluation to focus on terms with these 'undesirable' types allowed it to proceed automatically.

A key step in the algorithm is memoizing prior specialization results. If the traversal has already been specialized to a particular set of arguments, all future applications to the same set of arguments can be replaced by a call to the specialized version. This is the step that "ties the knot" of recursion in the algorithm, preventing unbounded unfolding and evaluation.

To implement this step, a call to an unspecialized traversal is first let-bound to a new name, and the resulting binding is maximally floated. Any further occurrence of the same unspecialized traversal is folded (Section 4.4) to this new name. All unfolding and evaluation occurs in the right-hand side of the let-binding, so all the calls share the results of the specialization, including recursive calls.

Since a binding is generated for every combination of traversal and type and dictionary arguments, a large number of fold patterns will be generated for non-trivial examples. Each of these patterns must be checked at every traversal application site. This is the single most time-consuming aspect of the optimization. HERMIT's original naive implementation of the fold rewrite could only check a single pattern at a time. The poor performance of this memoization step motivated the current implementation of fold in terms of TrieMaps, which greatly improved performance of the optimization by allowing all the patterns to effectively be checked simultaneously.

The SYB optimization and the Stream Fusion optimization in Chapter 7 were both implemented by first gaining an intuition for the optimization through interactive transformation. This approach appears to be fruitful, exposing many of the unforeseen (until implementation) details of the optimization early, and providing fast development iteration. HERMIT is the key enabling technology for this methodology.

9.3 Haskell-to-Hardware

HERMIT has also been used to build a prototype compiler backend which compiles Haskell programs to configurations suitable for a programmable logic device, such as an Field-Programmable Gate Array (FPGA). The dominant means of programming these devices is via specialized Hardware Description Languages (HDLs) such as Verilog [Thomas and Moorby, 1998] or VHDL [Armstrong and Gray, 1993]. Another approach which is currently being actively explored by the functional languages community is to reify a functional computation as a structure which can then be translated into a circuit description [Bjesse et al., 1998, Gill et al., 2013]. This approach is motivated by the desire to raise the level of abstraction for describing circuits, and to take advantage of advanced type system features offered by modern functional languages.

Typically, these functional HDLs are implemented as embedded domain-specific languages (eDSLs), using features of the host language such as type-classes to interpret computations both as regular programs (simulation) and data structures (reification). Despite continued progress, there are limits to the computations which can be reified this way [Mainland and Morrisett, 2010]. Notoriously tricky are issues of sharing, both of bindings and abstractions, and higher-order computations, which are common in functional programs.

The `lambda-ccc` compiler is implemented as a custom HERMIT plugin which observes the GHC Core representation of the program at compile time. The program is already an explicit structure at this point, bypassing many reification issues that source-level eDSLs face. Additionally, GHC Core is syntactically smaller, and more stable, than the full Haskell language, making it a more suitable target for compilation. In effect, `lambda-ccc` is an alternative back end for GHC which generates hardware descriptions, and it can take advantage of GHC's front end parsing, typechecking, and desugaring capabilities.

The plugin works by rewriting the GHC Core program to an alternative program which, when run, will generate a hardware description which implements the computation embodied by the original program. This hardware description can then be synthesized using standard tools.

Reflecting on the `lambda-ccc` prototype: "HERMIT made it relatively easy to experiment with different reification strategies. It was very helpful to have a high-level tool for writing individual transformations as well as repetition and traversal strategies." [Elliott, 2015]. While the choice to use HERMIT has been beneficial for prototyping, especially by leveraging HERMIT's interactive capabilities, one drawback is compilation speed, which is slower than desired. (HERMIT itself has had relatively little performance tuning.) It is likely that the ideas developed using the HERMIT prototype will be used to build a dedicated GHC plugin pass in the future.

HERMIT's use for a project such as `lambda-ccc` was unexpected, but beneficial to HERMIT's development. The project was the primary motivator for developing the ability to call the GHC typechecker from within HERMIT to generate dictionary expressions. This feature was later leveraged to create the dictionary instantiation transformation necessary to perform the type-class law proofs in Chapter 6. Additionally, many of HERMIT's practical features, especially in the Shell, were motivated by the needs of viewing and navigating the relatively large programs generated by the `lambda-ccc` translation.

Chapter 10

Related Work

There have been three main approaches taken to verifying properties of Haskell programs: testing, automated theorem proving, and semi-formal methods, such as equational reasoning. Testing tends to be lightweight, with good support for integrating into existing development workflows. Tests are adept at finding small counterexamples to properties which are not true, but tests do not constitute a proof. Automated theorem provers can provide formal, machine-checked proofs, but typically require considerable time and expertise. Using an automated prover often requires a radical departure from typical development workflows. Semi-formal methods offer a middle ground, with more assurance than testing, but considerably less effort than using a formal theorem prover.

10.1 Testing

The most prominent testing tool in the Haskell community is QuickCheck [Claessen and Hughes, 2000]. QuickCheck automatically generates random inputs to the property being tested, in search of a counterexample. A related tool, SmallCheck [Runciman et al., 2008], exhaustively generates test values of increasing size in order to find minimal counter examples. This has been extended by Lazy SmallCheck [Runciman et al., 2008, Reich et al., 2013], which also tests partial values. Jeuring et al. [2012] develops infrastructure which uses QuickCheck to test type-class laws or the individual steps of a user-supplied equational-reasoning proof in order to locate errors in such proofs. QuickCheck itself has been replicated in other languages, including Erlang [Arts et al., 2008], and C [Arts and Castro, 2011].

10.2 Automated Proof

Several tools exist which attempt to automatically prove properties of Haskell programs by interfacing with an automated theorem prover. The general approach taken by these tools is to translate the Haskell program, via GHC Core, into a first-order logic. Program properties are then checked by passing them to an external theorem prover for verification. Most of these tools provide their own automated induction principle(s), invoking the external theorem prover as required.

While this overlaps with some of HERMIT's functionality, the aim of these tools is different. They seek to generate proofs for program properties (and in some cases, the properties themselves) without user guidance. HERMIT's goal is broader, seeking to mechanize semi-formal reasoning for both proof and program transformation, lowering the burden of performing the sort of reasoning that is typically done by hand. In some cases, HERMIT could profitably incorporate these tools as decision procedures for its own proofs.

Zeno [Sonnex et al., 2012] is an automated prover which attempts to solve the general problem of determining if two Haskell terms are equivalent. Zeno targets the GHC Core representation of the program, but translates it to its own intermediate language which lacks the first class type equality (coercion evidence) features of GHC Core. Thus, Zeno targets Haskell 98 plus language extensions which are purely syntactic, not full GHC-extended Haskell. It automatically proves user-specified properties, including implications and any auxiliary lemmas it generates. Proofs are output to Isabelle/HOL theory files, which can then be checked. The language for specifying properties is similar to that of QuickCheck, and properties are included in the program. Zeno needs access to the definitons of all functions involved in a proof, meaning it cannot reason about library functions, including those from the Haskell Prelude. This is more restrictive than HERMIT, which can reason about library functions as long as the unfoldings for those functions are included in the interface file for their defining module. As Zeno has access to GHC Core, this appears to be a limitation of the implementation, and not fundamental. Zeno also cannot target programs which feature local recursive definitons or functions which are non-terminating, such as corecursive definitions. Zeno can only reason about inductive data types, meaning built-in types such as $Integer$, Int, and $Char$ cannot be used in targeted functions.

HALO [Vytiniotis et al., 2013] is another system which translates Haskell to first-order logic, via GHC Core. HALO's distinguishing feature is its translation, which allows it to reason is the presence of partial and infinite values. Additionally, the authors prove that if a property is proven for the translated version of the program, then it actually holds in the underlying program, a property of the translation which is often assumed. Properties can be specified using user-supplied predicates written in Haskell directly, in the target program. HALO relies on existing first-order logic provers, such as Z3 De Moura and Bjørner [2008] or Vampire Hoder et al. [2010]. The project appears to be not be maintained.

The Haskell Inductive Prover (Hip) [Rosén, 2012] is another tool which translates Haskell to first-order logic, focusing on automating induction proofs. Unlike the previous tools, Hip translates Haskell source, instead of GHC Core. A limitation of Hip is that it only attempts to apply induction to user-specified conjectures, not to any intermediate lemmas that may be needed to complete the proof. HipSpec [Claessen et al., 2013] is built on top of Hip. HipSpec uses QuickSpec [Claessen et al., 2010] to exhaustively generate conjectures (up to a fixed term size) about the functions of the target program. Hip is used to prove these conjectures first, using those that are successfully proven when attempting the main proof. The main novelty of HipSpec is that it infers suites of properties about programs from their definitions in a bottom-up fashion, rather than taking the goal-directed approach of the aforementioned tools which start from the user-stated program properties and seek to prove them. In fact, using HipSpec, user-specified properties are entirely optional.

HaskHOL [Austin, 2011] is an effort to implement a theorem prover for higher-order logic as a Haskell-hosted domain-specific language, providing Haskell libraries and tools access to proving-as-a-library. Work currently under peer review implements a HERMIT plugin which acts as a translation layer, providing program definitions to HaskHOL, though integration is in early stages.

One can also use proof assistants, such as Coq [Bertot and Castéran, 2004] or Agda [Norell, 2007], directly to interactively mechanize program transformations. This requires modeling the syntax and semantics of the target language, and then encoding the program in that model. If the goal is to *transform* the program (rather than just verifying properties), then the resulting program must be transliterated back into the target language before it can be compiled and executed. To perform this kind of reasoning on Haskell programs requires modeling Haskell's domain-theoretic setting of continuous functions over pointed ω-complete partial orders [Schmidt, 1986]. This is true even for programs written in a subset of the Haskell language, and is due to the fact that Haskell's partial values and lazy semantics are a poor fit for the total languages provided by such proof assistants. One of the aims of the HERMIT project is to make transforming and reasoning about Haskell programs easy for the user: familiarity with domain theory and proof assistants should not be prerequisites.

10.3 Semi-formal Tools

Semi-formal reasoning can be used both to prove properties of Haskell programs and to validate the correctness of program transformations. There are numerous examples of semi-formal reasoning being performed manually in the literature [e.g. Gibbons and Hutton, 2005, Bird, 2010, Danielsson and Jansson, 2004, Gill and Hutton, 2009]. Several tools have attempted to mechanize semi-formal reasoning on Haskell programs, though most are not currently maintained. The majority of these tools operate on Haskell source code (or some variant thereof), and do not attempt to support the full Haskell language, including extensions, which is implemented by GHC. HERMIT's decision to operate on GHC Core, during compilation, is the primary difference between it and these tools.

Closely related to HERMIT is the Programming Assistant for Transforming Haskell (PATH) [Tullsen, 2002]. Both are designed to be user directed, rather than fully automated, and are targeted at regular Haskell programmers, without advanced knowledge of language semantics and formal theorem proving tools. The significant difference is the choice of target language for transformations. PATH first translates Haskell into PATH-L, a Haskell-like functional language with explicit recursion and unlifted tuples, performs its transformations, then converts PATH-L back to Haskell, which is then compiled.

The Ulm Transformation System (Ultra) [Guttmann et al., 2003] is similar to PATH, although its underlying semantics are based on CIP [Bauer et al., 1988], whereas PATH develops its own formalism. A distinguishing feature of Ultra is that it operates on a subset of Haskell extended with some non-deterministic operators, thereby allowing concise non-executable specifications to be expressed and then transformed into executable programs.

Another tool similar to HERMIT is the Haskell Refactorer (HaRe) [Li et al., 2005, Brown, 2008, Li and Thompson, 2008a, Thompson and Li, 2013], which supports user-guided refactoring of Haskell programs. The objective of HaRe is slightly different, as refactoring is concerned with program transformation, whereas HERMIT supports both transformation and proof. HaRe targets Haskell source, providing a graphical user interface and built-in transformations for manipulating the program. The original version of HaRe targeted Haskell 98 source code, using Programatica [Hallgren et al., 2004] to transform Haskell source code into an AST. Work has recently begun on a re-implementation of HaRe which targets GHC-extended Haskell.

HERMIT is a direct descendant of HERA [Gill, 2006]. HERA operated on Haskell syntax using Template Haskell [Sheard and Peyton Jones, 2002]. An (unpublished) conclusion from HERA was that transformations such as the Worker/Wrapper transformation need typing information. The local availability of explicit type information was the original motivation for choosing GHC Core as the target language for HERMIT. HERA can be considered as an early prototype of HERMIT, now completely subsumed.

More broadly, there are a wide variety of refactoring tools for other languages. However, unlike HERMIT, most do not support higher-order commands and the scripting of composite refactorings [Li and Thompson, 2012]. One exception is Wrangler [Li and Thompson, 2008b], a refactoring tool for Erlang, which has recently added such support [Li and Thompson, 2012].

10.4 Stream Fusion

Starting with deforestation work by Wadler [1988] a number of approaches to deforestation in Haskell have been developed, including foldr/build [Gill et al., 1993], unfoldr/destroy [Svenningsson, 2002], and Stream Fusion [Coutts et al., 2007]. Each approach is limited both theoretically and practically. The inability of foldr/build to fuse zip is a theoretical limitation due to the nature of the primitive consumer ($foldr$), which can only traverse a single list at a time. On the other hand, fusing $foldl$ is only a practical limitation to foldr/build, and Gill [1996] proposes an *arity-raising* transformation to lift this limitation. As the dual to foldr/build, the unfoldr/destroy approach cannot fuse $unzip$, because the primitive producer ($unfoldr$) cannot produce more than a single list at a time. Stream Fusion extends unfoldr/destroy, overcoming a practical limitation when fusing $filter$. The case study in Chapter 7 addresses another practical limitation common to unfoldr/destroy and Stream Fusion, fusing $concatMap$. Further details, including an algorithm for list comprehension desugaring which is required to get good fusion, and further related work, can be found in Farmer et al. [2014].

10.5 Design

There are a wide variety of approaches to formalizing program transformation, such as fold/unfold [Burstall and Darlington, 1977], expression procedures [Scherlis, 1980, Sands, 1995], the CIP system [Bauer et al., 1988], and the Bird-Meertens Formalism [Meijer et al., 1991, Bird and de Moor, 1997]. These systems vary in their expressive power, often trading correctness for expressiveness. For example, fold/unfold is more expressive than expression procedures, but expression procedures ensures total correctness whereas fold/unfold allows transformations that introduce non-termination [Tullsen, 2002]. HERMIT implements rewrites to support fold/unfold reasoning, but sidesteps termination issues by relying on the user to direct application of the rewrites and to ensure termination.

HERMIT uses the Kansas University Rewrite Engine (KURE) [Sculthorpe et al., 2014] as its language for specifying transformations on GHC Core. Other strategic programming languages include Stratego [Bravenboer et al., 2008], which grew out of work on a strategy language to translate RML [Visser et al., 1998], and drew inspiration from ELAN [Borovanský et al., 2001]. StrategyLib [Lämmel and Visser, 2002] is the system most similar to KURE, and many aspects of the KURE design were drawn from it. Visser [2005] surveys the strategic programming discipline.

The combinators of Ltac [Delahaye, 2000], the tactics language used by the proof assistant Coq [Bertot and Castéran, 2004], are reminiscent of KURE's strategic programming combinators. The key differences are that Ltac tactics operate on proof obligations rather than tree-structured data, and that they return a *set* of sub-goals.

There has been significant work in safely handling bindings while working with abstract-syntax trees, a notoriously thorny problem. *Unbound* [Weirich et al., 2011], a Haskell-hosted DSL for specifying binding structure, is a possible solution. HERMIT uses congruence combinators for this task, which are a general mechanism for encapsulating the maintenance of *any* sort of contextual information, including bindings.

Chapter 11

Conclusion

Writing a program which is both correct and fast is difficult. One approach is to actually write two programs: one which is correct, the other which is fast, and then prove that they are equivalent. Establishing this equivalence formally requires considerable effort and expertise, meaning it is not appropriate for most projects. However, at least when both programs are written in functional languages, this equivalence can be established semi-formally by reasoning about the properties of pure functions. In some cases, the second program does not need to be written explicitly, and can be derived from the first, via program transformation.

While this workflow is popular in the functional languages community, it is typically performed manually, either by modifying the source code or by using pen and paper. This can be tedious and error-prone, resulting in obfuscated code or separate proof artifacts which must be manually updated as the program changes over time. This dissertation investigates the idea of mechanizing semi-formal reasoning during compilation as a means of addressing these concerns. It does so by developing such a system, called HERMIT, and evaluating its effectiveness on reasoning tasks which are actually performed by the Haskell community.

HERMIT is built to enable programmers to reason about their programs in a familiar, semi-formal style. It is important that reasoning in HERMIT can be performed at a similar level of abstraction to pen-and-paper reasoning, because semi-formal reasoning's simplicity is a large part of its appeal. HERMIT succeeds on this point, as the first and third case studies, along with other examples in this dissertation, demonstrate. Scripts in HERMIT closely correspond to their pen-and-paper counterparts in both form and length.

The individual steps in a script also closely resemble those of by-hand reasoning. The primary difference is that the HERMIT script must specify *where* to apply the transformation, in addition to *what* the transformation is. This may at first appear to be more work, but it can be argued that not doing so is just passing this work onto the reader. Experience refactoring examples as HERMIT developed shows that explicitly targeting transformations leads to scripts which are more legible and easier to update when the program changes.

Mechanical support unburdens the programmer from handling the details of large syntactic manipulations, which in turn allows semi-formal reasoning to be applied to real programs. HERMIT's composite rewrites are effective at performing many tedious reductions automatically, allowing the user to focus on key steps such as whether and when to fold or unfold a particular definition.

The fact that this mechanically supported reasoning occurs at compile time is what enables HERMIT's support for the entire Haskell language, including language extensions. HERMIT is the first such system to do so. Other benefits of reasoning at compile time include access to explicit, accurate, and local information about types; a syntactically smaller and more stable target language; and integration with the existing Haskell tool ecosystem.

With access to GHC internals, HERMIT is also an effective tool for prototyping domain-specific optimizations. Using domain-specific knowledge which the compiler is unable to discover on its own, such optimizations improve the performance of code written at a high level of abstraction. Such code is generally more concise, and more likely to be "obviously correct". The second case study uses HERMIT to develop one such optimization, leveraging HERMIT's interactive mode to gain intuition about the optimization by applying it to examples.

11.1 Reflections

Access to typing information was the original motivation for the decision to operate during compilation. This was largely due to Gill's experience while implementing HERA [Gill, 2006], the predecessor to HERMIT which operated on Haskell source code. Transforming at the source level required a large amount of syntactic transformation and a correspondingly large dictionary of transformations. One of HERA's goals was to mechanize the Worker/Wrapper transformation, which needs access to typing information. This information was difficult to gather at the source level, relying on type inference and type ascription.

That HERMIT benefits from GHC's election to use a small, strongly-typed intermediate language such as GHC Core should not be overlooked. The decision to structure GHC's optimizer around GHC Core was fortuitous, and in large part motivated by the desire that GHC's optimizer itself be series of program transformations. In a way, HERMIT is a natural extension of this thinking, allowing the programmer to direct transformation of GHC Core, rather than relying only on what can be done automatically.

There are drawbacks to reasoning at compile-time. Foremost, the programmer must reason about GHC Core, instead of the Haskell they are already familiar with. GHC Core is generally more verbose because it is syntactically simpler. It also features concepts, such as explicit type and dictionary expressions and explicit coercion evidence, which are not part of Haskell. However, these differences are not extreme, and GHC Core is sufficiently similar to Haskell that the same general reasoning tactics still apply.

From the beginning, HERMIT's development has been driven by example. In many cases, this means capabilities developed to handle one example were later merged with or subsumed by capabilities developed in the course of another example. In this sense, many aspects of HERMIT's design and implementation were *discovered*, rather than planned.

To provide one example, HERMIT's structural induction capability was at one point a special mode of the Shell. Reasoning in the presence of implications required yet another Shell mode, which assumed the antecedent. Only after the first version of the Century case study in Chapter 8 was completed did the (obvious, in retrospect) means of handling these features using KURE transformations and the context become clear.

One aspect of HERMIT that was definitely planned was the choice to operate during compilation, within GHC. Fortuitously, GHC developed its plugin capability around the time the HERMIT project got underway. This saved valuable time, allowing an early version of HERMIT to appear quickly. It also shaped HERMIT's architecture, determining the context in which HERMIT runs.

Early on, it was not clear whether targeting GHC Core, instead of Haskell directly, would be problematic. The worry was that GHC Core's additional syntactic verbosity would make visualizing and rewriting large programs difficult. Additionally, it was feared that GHC Core would constrain reasoning at a low level, filled with too many operational details. So far, this has not proved to be the case. The ability to control information displayed by the pretty-printer, combined with HERMIT's suite of transformations for focusing on bindings, applications, and other expressions, has mitigated the worry about verbosity. Composite rewrites, such as `bash` and `smash`, and KURE's library of traversal strategies have raised the level of abstraction during transformation.

Another concern was that HERMIT's dictionary would be ever-growing, requiring a KURE transformation to be defined for every new task. This too has, so far, not been the case. The dictionary grew rapidly during the first third of the project, but has been relatively stable since then. This is due to the development of lemmas, and specifically two features of lemmas. First, lemmas can be used as bidirectional rewrites, meaning a large class of local transformations can be recast as equational lemmas, assumed if necessary, and applied as transformations. Second, lemmas can be generated by other transformations as proof obligations to be proven post-hoc.

This was particularly beneficial for the Worker/Wrapper transformations, which had been implemented to that point as a large suite of specialized KURE transformations. The pre-conditions necessary to apply Worker/Wrapper vary based on the problem at hand, and the original system required them to be proven up-front, using rewrites-over-lemmas which were passed in as arguments. This meant that performing the Worker/Wrapper factorization required selecting from a large number of slightly different transformations, then providing a rewrite to prove the generated pre-condition non-interactively. The current means of factorization, described in Section 9.1 is much simpler.

The ability to fold expressions, as outlined in Section 4.4, went through several iterations. The initial implementation was an ad-hoc twin traversal of two expressions which was both slow and had several subtle bugs which resulted in ill-typed expressions. Fixing these bugs only worsened performance, to the point that running the SYB optimization discussed in Section 9.2 on large examples took *weeks*.

Early in HERMIT's development, Peyton Jones pointed out GHC's TrieMaps as a fast means of comparing expressions [Peyton Jones, 2013]. Extending them to support matching in the presence of holes (that is, to be a fast means of comparing expression *contexts*) paid off immediately in terms of performance. The SYB optimizations which were taking weeks could now complete in less than eight hours. This is largely due to the fact that multiple patterns can be checked simultaneously. The surprising consequence of this fold implementation was that it provided the key functionality needed to rewrite lemmas using other lemmas, which enables HERMIT's natural deduction style of proof.

As a means of interfacing with GHC's optimizer, HERMIT is a valuable means of prototyping new optimization techniques. This use case was not an original motivation for HERMIT, but became apparent as HERMIT developed. The interactive interface allows for rapid design iteration based on observing the optimization as it happens, rather than working out behavior post-mortem. The effectiveness of these prototypes can be evaluated on real Haskell programs without many of the burdens of modifying the compiler itself, including both the typical long build times inherent in a piece of software as large as a compiler and the need to ensure internal invariants are respected.

Some of the features required by HERMIT were not present in GHC because there was no need for them. This includes running the typechecker from within the optimization pipeline to generate dictionary expressions and declaring GHC rewrite rules which are not automatically applied. Thankfully, in these cases the GHC developers were kind enough to answer questions and accept patches. HERMIT also uses many GHC features in ways very different to how they are used in the compiler. This has led to the discovery of memory leaks and other subtle bugs which only manifest in HERMIT's use cases.

11.2 Future Work

HERMIT presents many avenues for possible future work. In general these fall into two categories: improving HERMIT itself and applying HERMIT to interesting reasoning tasks.

Improving HERMIT In terms of usability, the single biggest improvement that could be made to HERMIT would be to implement a proper parser for both GHC Core and HERMIT's lemmas. As GHC rewrite rules are the primary means of introducing lemmas, properties that do not conform to the syntactic restrictions on RULES pragmas must be constructed directly using special-purpose KURE transformations. This includes lemmas featuring expressions which are not function applications, and all composite lemmas. Lifting this restriction would allow many of the transformations in the HERMIT Dictionary to be recast as lemmas. It would also allow lemmas to be stated in scripts, and potentially alongside Haskell code, rather than riding on the coattails of RULES. Attempts have been made by others to provide a parser for GHC Core in the past [Tolmach et al., 2009], but this capability has since been removed from GHC because it was not maintained. Ideally, this parser would accept output produced by HERMIT's pretty printer.

The dynamic language for specifying commands in the HERMIT Shell is also creaking under its own weight. The requirement that types in the Shell be monomorphic clashes with KURE's flexibility regarding multiple universe types and traversal strategies, but is necessary due to the reliance on Haskell's $Dynamic$ type. Other features not present in the language, such as abstraction, limit the ability to reuse or programmatically direct transformations in the Shell. Replacing the Shell with a Haskell interpreter, akin to GHCi, is planned, but itself is a substantial project.

HERMIT's principle of structural induction is limited to data types where the recursion is only one level deep. This is not a fundamental limitation, and generalizing induction to arbitrarily-nested recursion would usefully extend the number of proofs that could be performed in practice.

To adhere to the idea of induction-as-a-rewrite, this likely requires enriching the data type used to represent lemmas.

Lemma libraries are a feature that allow HERMIT to import lemmas which have been packaged as an independent Haskell library. This allows proof efforts in HERMIT to be shared and reused, though so far this potential remains mostly untapped. HERMIT includes a library of assumed lemmas about basic types such as integers mostly as a proof-of-concept. Enriching these libraries is a possible future project. More interesting would be to use this feature to generate lemmas from an external source, such as the OpenTheory Project.

Proof in HERMIT occurs by rewriting lemmas, in the style of natural deduction. HERMIT is designed for rewriting, and this style of proof has proven effective thus far, but the goal of HERMIT is not to implement yet another theorem prover. Connecting HERMIT to existing provers, either for performing interactive proof or by treating them as decision procedures, would fruitfully expand HERMIT's proof capabilities without duplicating effort. Integration with HipSpec [Claessen et al., 2013] sounds especially promising, and would allow HERMIT to discharge many obligations automatically.

Applying HERMIT The rewrites for performing the Worker/Wrapper factorization presented in Section 9.1 have been used to mechanize most of the known Worker/Wrapper derivations found in the literature [Jones and Partain, 1993, Launchbury and Sheard, 1995, Peyton Jones, 2007, Michie, 1968, Sussman and Steele, 1975, Appel, 1992] as well as a large case study [Torrence, 2015]. These derivations are included as examples in the HERMIT software distribution. However, the simplification steps required to eliminate the representation-changing functions after the factorization step must still be worked out manually. This has been found to often be more difficult than proving the precondition for factorization. Devising a means of automating this simplification, or at least directing it at higher level, would allow Worker/Wrapper to be applied to ever-larger programs. HERMIT is certainly an appropriate tool for such a project. Indeed, Worker/Wrapper was the original motivating example for HERMIT.

HERMIT's TrieMaps, developed to implement the fold operation in Section 4.4, could be applied to other pattern matching problems. Their ability to check multiple patterns simultaneously yielded orders of magnitude speedup on the SYB optimization in Section 9.2. They could, for instance, be used to encode a route matcher for a web application framework, pattern matching incoming request URIs, extracting URI components using holes, and passing the matches to an appropriate handler.

HERMIT is potentially a valuable tool for teaching students equational reasoning techniques. Mechanization would lower the burden of missteps while the student is exploring a transformation or proof. The dictionary of existing rewrites would allow the student to focus on the strategy of the transformation, rather than the tedious details. This would likely require a richer user-interface than the current Shell; one that includes better guidance on the large collection of commands, or allows transformations to be expressed by direct manipulation, such as dragging expressions around with a mouse.

HERMIT's value for prototyping compiler optimizations stems from its interactive capabilities and its assistance for constructing compiler plugins. There are likely many potential domain-specific optimizations which cannot currently be expressed to GHC but could be implemented using HERMIT. Even general purpose optimizations can benefit from this approach, allowing their effectiveness on real programs to be tested before investing effort in modifying the compiler itself. That *interactivity* is a selling point of a system named HERMIT is certainly ironic.

References

A. Adamatzky. *Game of Life Cellular Automata*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 1849962162, 9781849962162.

M. D. Adams and T. M. DuBuisson. Template your boilerplate: Using template haskell for efficient generic programming. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 13–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364509. URL `http://doi.acm.org/10.1145/2364506.2364509`.

M. D. Adams, A. Farmer, and J. P. Magalhães. Optimizing SYB Is Easy! In *Proceedings of the 2014 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '14. ACM, 2014.

A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

J. R. Armstrong and F. G. Gray. *Structured Logic Design with VHDL*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-855206-1.

T. Arts and L. M. Castro. Model-based testing of data types with side effects. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 30–38, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0859-5. doi: 10.1145/2034654.2034662. URL `http://doi.acm.org/10.1145/2034654.2034662`.

T. Arts, L. M. Castro, and J. Hughes. Testing erlang data types with quviq quickcheck. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ERLANG '08, pages 1–8, New

York, NY, USA, 2008. ACM. ISBN 978-1-60558-065-4. doi: 10.1145/1411273.1411275. URL `http://doi.acm.org/10.1145/1411273.1411275`.

E. Austin. HaskHOL: A Haskell Hosted Domain Specific Language for Higher-Order Logic Theorem Proving. Master's thesis, University of Kansas, 2011.

F. L. Bauer, H. Ehler, A. Horsch, B. Moeller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP*. Springer-Verlag, 1988.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

R. Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.

R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.

R. S. Bird and L. G. L. T. Meertens. Nested datatypes. In *Proceedings of the Mathematics of Program Construction*, MPC '98, pages 52–67, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64591-8. URL `http://dl.acm.org/citation.cfm?id=648084.747162`.

P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184, Baltimore, Maryland, United States, 1998.

P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–98, 2001.

T. Braibant and D. Pous. Tactics for reasoning modulo AC in Coq. In *International Conference on Certified Programs and Proofs*, volume 7086 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2011.

M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52–70, 2008.

J. Breitner, R. A. Eisenberg, S. Peyton Jones, and S. Weirich. Safe zero-cost coercions for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 189–202, New York, NY, USA, 2014a. ACM. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628141. URL `http://doi.acm.org/10.1145/2628136.2628141`.

J. Breitner, R. A. Eisenberg, S. Peyton Jones, and S. Weirich. Safe zero-cost coercions for Haskell. In *International Conference on Functional Programming*, pages 189–202. ACM, 2014b.

C. M. Brown. *Tool Support for Refactoring Haskell Programs*. PhD thesis, University of Kent, 2008.

D. Burkett, J. Blitzer, and D. Klein. Joint Parsing and Alignment with Weakly Synchronized Grammars. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 127–135. Association for Computational Linguistics, 2010.

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

M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 241–253, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7. doi: 10.1145/1086365.1086397. URL `http://doi.acm.org/10.1145/1086365.1086397`.

K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, pages 268–279. ACM, 2000.

K. Claessen, N. Smallbone, and J. Hughes. Quickspec: Guessing formal specifications using testing. In *Proceedings of the 4th International Conference on Tests and Proofs*, TAP'10, pages 6–21, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13976-0, 978-3-642-13976-5. URL `http://dl.acm.org/citation.cfm?id=1894403.1894408`.

K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In *International Conference on Automated Deduction*, volume 7898 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2013.

D. Coutts. *Stream Fusion: Practical Shortcut Fusion for Coinductive Sequence Types*. PhD thesis, University of Oxford, 2010.

D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 315–326, Freiburg, Germany, 2007. ACM.

D. Coutts, I. Potoczny-Jones, and D. Stewart. Haskell: Batteries included. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 125–126, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-064-7. doi: 10.1145/1411286.1411303. URL http://doi.acm.org/10.1145/1411286.1411303.

N. A. Danielsson and P. Jansson. Chasing bottoms: A case study in program verification in the presence of partial and infinite values. In *International Conference on Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 85–109. Springer, 2004.

L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL http://dl.acm.org/citation.cfm?id=1792734.1792766.

D. Delahaye. A tactic language for the system Coq. In *Logic for Programming and Automated Reasoning*, pages 85–95. Springer, 2000.

N. Dershowitz, J. Hsiang, N. A. Josephson, and D. A. Plaisted. Associative-commutative rewriting. In *International Joint Conference on Artificial Intelligence*, volume 2, pages 940–944. Morgan Kaufmann, 1983.

C. Elliott. personal communication, 2015.

A. Farmer, C. Höner zu Siederdissen, and A. Gill. The HERMIT in the stream: Fusing Stream Fusion's concatMap. In *Workshop on Partial Evaluation and Program Manipulation*, pages 97–108. ACM, 2014.

A. Farmer, N. Sculthorpe, and A. Gill. Hermit case studies: Proving Type-Class Laws & Making a Century, 2015. URL `http://ku-fpg.github.io/research/HERMIT/case-studies`.

GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.8.4*, 2014. URL `http://downloads.haskell.org/~ghc/7.8.4/docs/html`.

J. Gibbons and G. Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae*, 66(4):353–366, 2005.

A. Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, The University of Glasgow, January 1996.

A. Gill. Introducing the Haskell equational reasoning assistant. In *Haskell Workshop*, pages 108–109. ACM, 2006.

A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, 2009.

A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. pages 223–232. ACM Press, 1993.

A. Gill, T. Bull, A. Farmer, G. Kimmell, and E. Komp. Types and associated type families for hardware simulation and synthesis: The internals and externals of Kansas Lava. *Journal of*

Higher-Order and Symbolic Computation, pages 1–20, 2013. ISSN 1388-3690. doi: 10.1007/s10990-013-9098-7. URL `http://dx.doi.org/10.1007/s10990-013-9098-7`.

J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989. ISBN 0-521-37181-3.

D. Grune and C. J. Jacobs. *Parsing Techniques: A Practical Guide*. Springer-Verlag New York Inc, 2008.

W. Guttmann, H. Partsch, W. Schulte, and T. Vullinghs. Tool support for the interactive derivation of formally correct functional programs. *Journal of Universal Computer Science*, 9(2):173–188, 2003.

J. J. Hallett and A. J. Kfoury. Programming examples needing polymorphic recursion. *Electron. Notes Theor. Comput. Sci.*, 136:57–102, July 2005. ISSN 1571-0661. doi: 10.1016/j.entcs.2005.06.014. URL `http://dx.doi.org/10.1016/j.entcs.2005.06.014`.

T. Hallgren, J. Hook, M. P. Jones, and R. B. Kieburtz. An overview of the Programatica toolset. In *High Confidence Software and Systems*, 2004.

R. Hinze. Generalizing generalized tries. *J. Funct. Program.*, 10(4):327–351, July 2000. ISSN 0956-7968. doi: 10.1017/S0956796800003713. URL `http://dx.doi.org/10.1017/S0956796800003713`.

R. Hinze, T. Harper, and D. W. James. Theory and Practice of Fusion. *Implementation and Application of Functional Languages*, pages 19–37, 2011.

C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972. ISSN 0001-5903. URL `http://dx.doi.org/10.1007/BF00289507`. 10.1007/BF00289507.

K. Hoder, L. Kovács, and A. Voronkov. Interpolation and symbol elimination in vampire. In *Proceedings of the 5th International Conference on Automated Reasoning*, IJCAR'10, pages 188–195, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14202-8, 978-3-642-14202-4. doi: 10.1007/978-3-642-14203-1_16. URL http://dx.doi.org/10.1007/978-3-642-14203-1_16.

C. Höner zu Siederdissen. Sneaking around concatMap: Efficient combinators for dynamic programming. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, pages 215–226, Copenhagen, Denmark, 2012. ACM.

C. Höner zu Siederdissen, I. L. Hofacker, and P. F. Stadler. How to Multiply Dynamic Programming Algorithms. In *Brazilian Symposium on Bioinformatics (BSB 2013), Lecture Notes in Bioinformatics*, volume 8213. Springer, Heidelberg, 2013.

Z. Hu, T. Yokoyama, and M. Takeichi. Program optimizations and transformations in calculation form. In *Proceedings of the 2005 International Conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, pages 144–168, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-45778-X, 978-3-540-45778-7. doi: 10.1007/11877028_5. URL http://dx.doi.org/10.1007/11877028_5.

F. W. Huang, J. Qin, C. M. Reidys, and P. F. Stadler. Partition function and base pairing probabilities for RNA–RNA interaction prediction. *Bioinformatics*, 25(20):2646–2654, 2009.

P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. doi: 10.1145/1238844.1238856. URL http://doi.acm.org/10.1145/1238844.1238856.

J. Hughes. Why functional programming matters. In *The Computer Journal*, volume 32, pages 98–107, 1989.

G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

J. Jeuring, S. Leather, J. Magalhães, and A. Rodriguez Yakushev. Libraries for generic programming in Haskell. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 165–229. Springer, 2009. ISBN 978-3-642-04651-3.

J. Jeuring, P. Jansson, and C. Amaral. Testing type class laws. In *Haskell Symposium*, pages 49–60. ACM, 2012.

T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 3-387-15975-4. URL `http://dl.acm.org/citation.cfm?id=5280.5292`.

I. Jones. The Haskell Cabal: a common architecture for building applications and libraries. pages 340–354, 2005. URL `http://www.cs.ioc.ee/tfp-icfp-gpce05/tfp-proc/24num.pdf`.

M. P. Jones. Dictionary-free overloading by partial evaluation. *Lisp Symb. Comput.*, 8(3):229–248, Sept. 1995. ISSN 0892-4635. doi: 10.1007/BF01019005. URL `http://dx.doi.org/10.1007/BF01019005`.

N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-020249-5.

S. P. Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In Hammond and O'Donnell, editors, *Functional Programming*, Springer Verlag Workshops in Computing, pages 201–220, 1993.

H. Kirchner and P.-E. Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Types in Languages Design and Implementation*, pages 26–37. ACM, 2003.

R. Lämmel and J. Visser. Typed combinators for generic traversal. In *Practical Aspects of Declarative Programming*, pages 137–154. Springer, 2002.

J. Launchbury and T. Sheard. Warm fusion: deriving build-catas from recursive definitions. In *FPCA '95: Proceedings of the 7th international conference on Functional programming languages and computer architecture*, pages 314–323. ACM Press, 1995.

H. Li and S. Thompson. Tool support for refactoring functional programs. In *Partial evaluation and semantics-based program manipulation*, pages 199–203. ACM, 2008a.

H. Li and S. Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2Nd Workshop on Refactoring Tools*, WRT '08, pages 2:1–2:4, New York, NY, USA, 2008b. ACM. ISBN 978-1-60558-339-6. doi: 10.1145/1636642.1636644. URL `http://doi.acm.org/10.1145/1636642.1636644`.

H. Li and S. Thompson. A domain-specific language for scripting refactorings in erlang. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 501–515, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28871-5. doi: 10.1007/978-3-642-28872-2_34. URL `http://dx.doi.org/10.1007/978-3-642-28872-2_34`.

H. Li, S. Thompson, and C. Reinke. The Haskell refactorer, HaRe, and its API. In *Workshop on Language Descriptions, Tools, and Applications*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 29–34. Elsevier, 2005.

S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199528. URL `http://doi.acm.org/10.1145/199448.199528`.

G. Mainland and G. Morrisett. Nikola: Embedding compiled gpu functions in haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 67–78, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: 10.1145/1863523.1863533. URL `http://doi.acm.org/10.1145/1863523.1863533`.

S. Marlow. Haskell 2010 Language Report, 2009.

S. Marlow and S. Peyton Jones. The Glasgow Haskell Compiler. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications*, volume II. 2012.

E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

D. Michie. Memo Functions and Machine Learning. *Nature*, 218:19–22, 1968.

U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

B. O'Sullivan. `http://hackage.haskell.org/package/criterion`.

W. Partain. The nofib Benchmark Suite of Haskell Programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, 1993.

L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5 (3):363–397, 1989.

J. Peterson. Dynamic Typing in Haskell. Technical Report YALEU/DCS/RR-1022, Department of Computer Science, Yale University, New Haven, Connecticut.

S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

S. Peyton Jones. Call-pattern Specialisation for Haskell Programs. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 327–337. ACM, 2007.

S. Peyton Jones. personal communication, 2013.

S. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, pages 636–666. Springer, 1991.

S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4&5):393–433, 2002.

S. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, pages 203–233. ACM, 2001.

S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158524. URL `http://doi.acm.org/10.1145/158511.158524`.

A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 151–160, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145836. URL `http://doi.acm.org/10.1145/2145816.2145836`.

J. S. Reich, M. Naylor, and C. Runciman. Advances in lazy smallcheck. In *International Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2013.

D. Rosén. Proving equational Haskell properties using automated theorem provers. Master's thesis, University of Gothenburg, 2012.

C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and Lazy Smallcheck: Automatic exhaustive testing for small values. In *Haskell Symposium*, pages 37–48. ACM, 2008.

D. Sands. Higher-order expression procedures. In *Partial evaluation and semantics-based program manipulation*, pages 178–189. ACM, 1995.

A. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995.

W. L. Scherlis. *Expression procedures and program derivation*. PhD thesis, Stanford University, 1980.

D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for gadts. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 341–352, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: 10.1145/1596550.1596599. URL `http://doi.acm.org/10.1145/1596550.1596599`.

N. Sculthorpe and G. Hutton. Work It, Wrap It, Fix It, Fold It. *Journal of Functional Programming*, 24(1):113–127, Jan. 2014.

N. Sculthorpe, N. Frisby, and A. Gill. The Kansas University Rewrite Engine: A Haskell-embedded strategic programming language with custom closed universes. *Journal of Functional Programming*, 24(4):434–473, 2014.

T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In *Haskell Workshop*, pages 1–16. ACM, 2002.

N. J. J. Smith. *Logic: The Laws of Truth*. Princeton University Press, Princeton, NJ, USA, 2012. ISBN 0691151636, 9780691151632.

W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *Lecture Notes in Computer Science*, pages 407–421. Springer, 2012.

C. Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, Aug. 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1–49, 2000.

M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Types in Language Design and Implementaion*, pages 53–66. ACM, 2007.

G. J. Sussman and G. L. Steele, Jr. An interpreter for extended lambda calculus. Technical report, Cambridge, MA, USA, 1975.

J. Svenningsson. *Shortcut Fusion for Accumulating Parameters Zip-like Functions*. PhD thesis, 2002.

D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language (4th Ed.)*. Kluwer Academic Publishers, Norwell, MA, USA, 1998. ISBN 0-7923-8166-1.

S. Thompson and H. Li. Refactoring tools for functional languages. *Journal of Functional Programming*, 23(3):293–350, 2013.

A. Tolmach, T. Chevalier, and the GHC Team. An External Representation for the GHC Core Language (For GHC 6.10). Unpublished, 2009.

B. Torrence. The Life Changing HERMIT: A Case Study of the Worker/Wrapper Transformation. Master's thesis, University of Kansas, 2015.

M. Tullsen. *PATH, A Program Transformation System for Haskell*. PhD thesis, Yale University, 2002.

E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming*, pages 13–26. ACM, 1998.

D. Vytiniotis and S. Peyton Jones. Evidence normalization in System FC. In *International Conference on Rewriting Techniques and Applications*, pages 20–38. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013.

D. Vytiniotis, S. Peyton Jones, K. Claessen, and D. Rosén. HALO: Haskell to logic through denotational semantics. In *Symposium on Principles of Programming Languages*, pages 431–442. ACM, 2013.

P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the 2nd European Symposium on Programming*, pages 344–358, London, UK, UK, 1988. Springer-Verlag.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75283. URL `http://doi.acm.org/10.1145/75277.75283`.

L. Wasserman, 2013. URL `http://hackage.haskell.org/package/TrieMap`.

M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, Apr. 1991. ISSN 0164-0925. doi: 10.1145/103135.103136. URL `http://doi.acm.org/10.1145/103135.103136`.

S. Weirich, B. A. Yorgey, and T. Sheard. Binders unbound. In *International Conference on Functional Programming*, pages 333–345. ACM, 2011.

M. Wenzel and S. Berghofer. *The Isabelle System Manual*, 2012. URL `http://isabelle.in.tum.de`.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation*, pages 53–66. ACM, 2012.