Reasoning with the HERMIT

Tool Support for Equational Reasoning on GHC Core Programs

Andrew Farmer
Information and Telecommunication Technology Center
University of Kansas, USA
afarmer@ittc.ku.edu

Neil Sculthorpe
Department of Computer Science
Swansea University, UK
N.A.Sculthorpe@swansea.ac.uk

Andy Gill
Information and Telecommunication Technology Center
University of Kansas, USA
andygill@ittc.ku.edu

Abstract

A benefit of pure functional programming is that it encourages *equational reasoning*. However, the Haskell language has lacked direct tool support for such reasoning. Consequently, reasoning about Haskell programs is either performed manually, or in another language that does provide tool support (e.g. Agda or Coq).

HERMIT is a Haskell-specific toolkit designed to support equational reasoning and user-guided program transformation, and to do so as part of the GHC compilation pipeline. This paper describes HERMIT's recently developed support for equational reasoning, and presents two case studies of HERMIT usage: checking that type-class laws hold for specific instance declarations, and mechanising textbook equational reasoning.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

Keywords HERMIT, Equational Reasoning, Type Class Laws

1. Introduction

Currently, most equational reasoning on Haskell programs is performed manually, using pen-and-paper or text editors, because of the lack of direct tool support. While some equational-reasoning tools do exist for Haskell [21, 40], they only target a subset of Haskell 98, not the full language (and certainly not the GHC-extended version of Haskell that is widely used in practice). This is unfortunate, as pen-and-paper reasoning is slow, error prone, and allows the reasoner to neglect details of the semantics. For example, a common mistake is to neglect to consider partial and infinite values, which are notoriously tricky [8]. This was recently demonstrated by Jeuring et al. [22], who showed that the standard implementations of the state monad do not satisfy the monad laws.

An alternative approach is to transliterate a Haskell program into a language or proof assistant that does provide support for equational reasoning, such as Agda [26] or Coq [38]. The desired reasoning can then be performed in that language, and the resultant program (or program property) transliterated back into Haskell. However, the semantics of these languages differ from Haskell, sometimes in subtle ways, so the reasoning steps used may not carry over to Haskell.

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.

A similar situation arises regarding GHC's rewrite rules [27]. 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 pen-and-paper reasoning, any change to the implementation of the involved functions requires the reasoning steps to be updated accordingly. Such revisions can easily be neglected, and, furthermore, even if a calculation is up-to-date, a user cannot be sure of that without manually examining the individual steps herself. What is needed is a mechanical connection between the source code, the reasoning steps, and the compiled program.

To address this situation, we have implemented a GHC plugin called HERMIT [11, 12, 14, 33]. HERMIT is a toolkit that supports interactive equational reasoning, and provides mechanical assurances of the correctness of that reasoning. HERMIT operates on GHC's internal core language, part-way through the compilation process. User-specified transformations are applied to the program being compiled, and the user's equational reasoning steps are checked. By performing equational reasoning during compilation, HERMIT is able to check that the reasoning steps correspond to the current implementation of the program, in the context of the language extensions currently being used.

The initial HERMIT implementation [12] only supported equational reasoning that was *transformational* in nature; that is, HERMIT allowed the user to apply a sequence of correctness-preserving transformations to the Haskell program, resulting in an equivalent but (hopefully) more efficient program. This was sufficient to allow some specific instances of known program transformations to be mechanised [33], as well as for encoding prototypes of new optimisation techniques [1, 13]. However, some of the transformation steps used were only valid in certain contexts, and HERMIT had no facility for checking the necessary preconditions. Thus these preconditions had to be verified by hand. Furthermore, it was not possible to state and reason about auxiliary properties of the program being transformed, or to use inductive techniques to verify their correctness. This paper describes the addition of these facilities to HERMIT, and discusses our experiences of using them on two case studies. Specifically, the contributions of this paper are:

Haskell'15, September 3-4, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3808-0/15/09...
http://dx.doi.org/10.1145/2804302.2804303

- We describe the new equational-reasoning infrastructure provided by HERMIT, discussing the issues that arose as a consequence of our choice to work within GHC and target its internal language. (Section 2).
- We demonstrate interactive HERMIT usage by mechanically verifying a GHC rewrite rule, giving a detailed walk-through of the reasoning steps (Section 3).
- We present a case study of mechanically verifying that type-class laws hold for specific class instances (Section 4).
- We explain how HERMIT can be integrated with Cabal such that HERMIT scripts are automatically checked and run when a package is built (Section 4.3).
- We present a case study of mechanising a chapter from *Pearls of Functional Algorithm Design* [2], a textbook dedicated to deriving Haskell programs by calculation (Section 5).

2. Equational Reasoning using HERMIT

HERMIT is a GHC plugin that allows a user to apply custom transformations to a Haskell program amid GHC's optimisation passes. HERMIT operates on the program after it has been translated into GHC Core, GHC's internal intermediate language. GHC Core is an implementation of System $F_C^{\uparrow}$, which is System F [20, 29] extended with let-binding, constructors, and first-class type equalities [37]. Type checking is performed during the translation, and GHC Core retains the typing information as annotations.

The availability of typing information is a significant advantage of working with GHC Core rather than with Haskell source code. Another significant advantage is that GHC Core is a syntactically smaller language, and consequently there are far fewer cases to consider. For example, both **if**-**then**-**else** expressions and the special *seq* function are translated into explicit **case** expressions in GHC Core, and thus do not need to be considered. Arguably, some of this second advantage could instead be gained by working with desugared Haskell source code. However, there are limits to what can be desugared within Haskell source code: for example, while **if**-**then**-**else** can be desugared to a *Haskell* **case** expression, the *seq* function cannot. (The semantics of GHC Core **case** expressions differ slightly from Haskell **case** expressions.)

HERMIT provides commands for navigating a GHC Core program, applying transformations, version control, pretty printing, and invoking GHC analyses and optimisation passes. To direct and combine transformations, HERMIT uses the strategic programming language KURE [34] to provide a rich family of traversal and strategy combinators. HERMIT offers three main interfaces:

- An interactive read-eval-print loop (REPL). This allows a user to view and explore the program code, as well as to experiment with transformations. In Section 3 we present an example of using the REPL to verify a GHC rewrite rule; examples of using the REPL to perform program transformations can be found in Farmer et al. [12] and Sculthorpe et al. [33].
- HERMIT scripts. These are sequences of REPL commands, which can either be loaded and run from within the REPL, or automatically applied by GHC during compilation. We present an example HERMIT script in Section 5.
- A domain-specific language (DSL) for transformation, embedded in Haskell. This allows the user to construct a custom GHC plugin using all of HERMIT's capabilities. The user can run transformations in different stages of GHC's optimisation pipeline, and add custom transformations to the REPL. New transformations and program properties can be encoded by defining Haskell functions directly on the Haskell data type representing the GHC Core abstract syntax, rather than using the more limited (but safer) monomorphically typed combinator language available to the REPL and scripts. We present an example of a user-defined program property in the Appendix.

This paper will describe HERMIT's new equational-reasoning infrastructure, but will not otherwise discuss its implementation or existing commands. Interested readers should consult the previous HERMIT publications [11, 12, 33], or try out the HERMIT toolkit [14] for themselves.

2.1 Program Properties

As discussed in Section 1, the HERMIT toolkit initially only supported program transformation, and any equational reasoning had to be structured as a sequence of transformation steps applied to the original source program [e.g. 33]. This was limiting, as equational reasoning often involves stating and verifying properties of the program being transformed, so that they can be used to validate the transformations being applied.

To address this, we have added support for stating program properties, which are referred to as *lemmas* within HERMIT. A primitive HERMIT lemma takes the form of an equality between two GHC Core expressions, and may involve universally quantified variables. For example:

Map Fusion
$\forall\, f\ g.\ \ map\ f \circ map\ g \equiv map\ (f \circ g)$

Composite lemmas can be formed using the logical connectives implication, conjunction and disjunction, and universal quantifiers can scope over composite lemmas. For example:

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

HERMIT maintains a set of lemmas, and tracks which of them have been verified by a HERMIT calculation. Once verified, lemmas can be mechanically used to validate transformation steps that have preconditions, and primitive lemmas can be applied as program transformations (left-to-right or right-to-left). Composite lemmas can be manipulated by HERMIT commands corresponding to standard introduction and elimination rules for logical connectives, and universally quantified variables can be instantiated. A primitive lemma can be verified by applying a sequence of transformation steps to either (or both) sides of the lemma, until HERMIT is satisfied that both sides are α-equivalent. Such reasoning can either be performed interactively, or by loading a HERMIT script. We will demonstrate interactive reasoning in Section 3.

The most convenient way of introducing primitive lemmas into HERMIT is by exploiting GHC rewrite-rule pragmas [27]. These pragmas are *intended* to allow a user to introduce custom optimisations into the GHC optimisation pipeline. For example, a rewrite rule corresponding to the Map Fusion lemma above can be added to a Haskell source file as follows:

```
{-# RULES "map-fusion" [~]
        forall f g. map f . map g = map (f . g)
 #-}
```

GHC will parse and type check this rule, and translate it into GHC Core. HERMIT exploits this mechanism by generating a lemma from the translated rule. The [∼] annotation renders this rule inactive [16, Section 7.21.1], which allows us to use this mechanism to introduce lemmas that are not intended to be used as standalone optimisations.

There are some restrictions on the form of the left-hand side of a GHC rewrite rule [27, Section 2.2], so this approach can only generate a subset of all possible primitive lemmas. However, in practice, all of the primitive lemmas that we needed for our case studies fell within this subset.

Currently, we introduce composite lemmas using HERMIT's transformation DSL. Primitive lemmas can also be introduced in this way, but as this requires working with the GHC Core data type directly, it tends to be less convenient and more error-prone than rewrite-rule pragmas. We present an example of such a composite-lemma definition in the Appendix. As future work, we aim to allow the user to introduce composite lemmas into HERMIT using (extended) Haskell syntax, which HERMIT would then type check, parse and convert to GHC Core.

2.2 Explicit Types and Dictionaries

In GHC Core, polymorphic functions have explicit type arguments (as GHC Core is based on System F), and type-class methods are implemented as functions that take a *class dictionary* as an argument [45]. A class dictionary is essentially a record containing a definition for each method of the type class, specialised to a concrete class parameter (or parameters). Thus each class instance generates a single dictionary (which may, in turn, be abstracted over other dictionaries).

To allow us to demonstrate the presence of these implicit type and dictionary arguments, first consider the explicit types of the following polymorphic functions:

$$\begin{aligned}&\mathit{id} &&:: \mathit{forall}\ a\ .\ \ a \to a\\ &\mathit{fmap} &&:: \mathit{forall}\ f\ .\ \ \mathit{Functor}\ f \Rightarrow \mathit{forall}\ a\ b\ .\ \ (a \to b) \to f\ a \to f\ b\end{aligned}$$

In GHC Core, each universally quantified type variable is an additional (type) argument to the function, and each class constraint is an additional dictionary argument to the function. For example, consider the functor identity law, expressed as a GHC rewrite rule:

```
{-# RULES "fmap-id" [~]
      fmap id = id
  #-}
```

The rewrite rule is written using Haskell source code, so the type and dictionary arguments are implicit. However, the representation of this law as a HERMIT lemma in GHC Core makes these extra arguments explicit:

fmap-id
$\forall\ f\ t\ \$dFunctor\ .\ \ fmap\ f\ \$dFunctor\ t\ t\ (id\ t) \equiv id\ (f\ t)$

Here, f and t are type variables, and $\$dFunctor$ is a dictionary variable. When generating names for dictionary variables, GHC prefixes the class name with "$d". Also, as we will see shortly, GHC prefixes the names of dictionary *instances* with "$f". A significant advantage of using GHC rewrite rules to generate lemmas is that these type and dictionary arguments are inferred and automatically inserted by GHC.

Returning to the functor identity law, note that this is not a lemma to be verified. Rather, the law is a *specification* that the class author expects any *instance* of the class to satisfy. To check that the law holds for a specific instance, we must first *instantiate* the type variable f and its corresponding dictionary "$\$dFunctor$", thereby generating the lemma that we wish to hold. For example, instantiating this lemma to the $Maybe$ data type would give the following:

fmap-id
$\forall\ t\ .\ \ fmap\ Maybe\ \$fFunctorMaybe\ t\ t\ (id\ t) \equiv id\ (Maybe\ t)$

HERMIT provides commands to specialise lemmas by instantiating variables in this way, whether dictionary variables or otherwise, as we will demonstrate in Section 4.1.

2.3 Missing Unfoldings

Equational reasoning often involves *fold/unfold* transformations [5]. One consequence of our choice to 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, for recursive functions that are not marked with an explicit INLINE pragma, GHC does not normally include their unfoldings in the interface files for their defining modules. This prevents us from unfolding those functions in HERMIT. This includes, for example, the $+\!\!+$ and map functions.

We currently have three work-arounds for this issue. The first option is to recompile the defining packages with GHC's `-fexpose-all-unfoldings` flag. In the case of $+\!\!+$ and map, this means recompiling the `base` package. The second option is to redefine the function with a new name, and use that function in our programs instead of the library function. For example:

$$\begin{aligned}&myAppend :: [a] \to [a] \to [a]\\ &myAppend\ [\,]\qquad ys = ys\\ &myAppend\ (x : xs)\ ys = x : myAppend\ xs\ ys\end{aligned}$$

However, this is not an option if we want to reason about pre-existing code that uses the library version of that 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
  #-}
```

We are not entirely happy with any of these work-arounds, and finding a cleaner solution remains as future work.

GHC offers a systematic means of compiling Haskell libraries in multiple *ways* and installing these builds in parallel. For example, the user may desire to install both normal object files and object files that include extra information for run-time profiling. One possible solution to our problem would be to extend this mechanism by adding a new 'way' that generates interface files that include unfolding information for every exported function.

2.4 Structural Induction

Haskell programs usually contain recursive functions defined over (co)inductive data types. Reasoning about such programs often requires the use of an induction principle. For example, while $[\,] +\!\!+ xs \equiv xs$ can be established simply by unfolding the definition of $+\!\!+$, establishing the similar property $xs +\!\!+ [\,] \equiv xs$ requires reasoning inductively about the structure of xs. Inductive reasoning cannot be expressed as a sequence of transformation steps: both the source and target expression must be known in advance, and the validity of rewriting one to the other is established by verifying the inductive and base cases.

HERMIT provides structural induction over algebraic data types as a built-in transformation. The remainder of this section will formalise HERMIT's induction principle, then in Section 3 we will give an example of using induction in HERMIT.

We first introduce some notation. We write $\overrightarrow{vs}$ to denote a sequence of variables, and $\forall\ (C\ \ \overrightarrow{vs} :: A)$ to quantify over all constructors C of the algebraic data type A, fully applied to a sequence $\overrightarrow{vs}$ of length matching the arity of C. Let $\mathbb{C} : A \rightsquigarrow B$ denote that $\mathbb{C}$ is an expression context containing one or more holes of type A, and having an overall type B. For any expression $a :: A$, then $\mathbb{C}[\![a]\!]$ denotes the context $\mathbb{C}$ with all holes filled with the expression a.

The structural-induction inference rule provided by HERMIT is defined in Figure 1. The conclusion of the rule is called the *induction hypothesis*. Informally, the premises require that:

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

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

Figure 1: Structural induction.

Given contexts $\mathbb{C}, \mathbb{D} : [A] \rightsquigarrow B$, for any types A and B, then:

$$\frac{\mathbb{C}[\![\mathit{undefined}]\!] \equiv \mathbb{D}[\![\mathit{undefined}]\!] \qquad \mathbb{C}[\![\ [\,]\ ]\!] \equiv \mathbb{D}[\![\ [\,]\ ]\!] \qquad \forall\,(a :: A, as :: [A])\,.\ \ (\mathbb{C}[\![as]\!] \equiv \mathbb{D}[\![as]\!]) \Rightarrow (\mathbb{C}[\![a : as]\!] \equiv \mathbb{D}[\![a : as]\!])}{\forall\,(xs :: [A])\,.\ \ \mathbb{C}[\![xs]\!] \equiv \mathbb{D}[\![xs]\!]}$$

Figure 2: Structural induction on lists.

- 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 matching type).

As a more concrete example, specialising structural induction to the list data type gives the inference rule in Figure 2.

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. We are currently in the process of implementing a more general induction principle that will allow the inductive hypothesis to be applied to a variable n constructors deep.

2.5 Correctness and Equivalence

HERMIT provides a large suite of built-in primitive transformations. Some of these are transformations and optimisation passes lifted from the internals of GHC, whereas others are taken from the literature on program transformation (e.g. [5, 9, 19]). Our aim with HERMIT is to provide tool support for the kind of equational reasoning that is commonly done on pen-and-paper, and to that end we seek to support the wide range of transformation and equational-reasoning techniques that are used in practice. This has consequences for correctness, as some of the transformations provided by HERMIT only offer *partial correctness*: the output produced by a program before transformation is compatible with the output after transformation. By *compatible* we mean that the output can be either more or less defined, but that the defined output does not differ in value. For example, fold/unfold transformation can easily introduce non-termination if used carelessly [40], but is widely used in pen-and-paper reasoning.

HERMIT's equivalence relation ($\equiv$) is based on transformation steps: two GHC Core expressions are considered to be equal if one can be transformed into the other, modulo α-equality. Consequently, whether an equivalence is partially or totally correct depends on the correctness of the transformations used. The majority of HERMIT's transformations are totally correct, and many of the remaining partially correct transformations are totally correct, *given certain preconditions*. These preconditions are encoded in HERMIT, but by default a user may choose to ignore them when applying a transformation. This is a pragmatic design decision: it allows a user to experiment without having to keep detouring to satisfy pre-conditions. In the next version of HERMIT, we intend to allow the user the option of disabling the set of partially correct transformations, and of enforcing that any preconditions are satisfied before a transformation can be used. The user can then choose the desired trade-off between correctness, expressiveness and convenience.

Note that there is no built-in semantic model within HERMIT (or GHC). The primitive transformations are essentially axioms, and themselves have no mechanical verification of their correctness beyond a check that they produce a well-typed GHC Core term. A substantial avenue for future work is to create a mechanical connection between HERMIT's primitive transformations and a semantic model, so that they can be formally verified. There has been recent work on translating GHC Core to first-order logic so that properties can be verified by an external automated theorem prover [44], and it seems plausible that this approach could be incorporated into HERMIT.

3. Interactive Proof Example

In this section we will demonstrate HERMIT's interactive mode by performing a calculation to validate a GHC rewrite rule. We will use the Haskell source file in Figure 3 for this example. The rule `map-fusion` is (a slight reformulation of) the motivating example of a rewrite rule given by Peyton Jones et al. [27, Section 2]. Note that we define *map* explicitly, rather than using the definition in the standard Prelude, to avoid the issues with unfolding pre-compiled definitions (as discussed in Section 2.3).

We begin by invoking HERMIT on the source file:

```
> hermit MapFusion.hs +MapFusion
```

GHC begins compiling `MapFusion.hs`, performing parsing, type checking and desugaring, before pausing compilation and passing control to the HERMIT interactive shell. The `+MapFusion` flag specifies that we wish to invoke HERMIT on the *MapFusion* module. (In general, compiling a module with GHC may trigger the compilation of dependent modules, and we may wish to run HERMIT on any or all of them.)

```
module main:MapFusion where
  map :: ∀ a b . (a → b) → [a] → [b]
```

HERMIT presents a summary of the module, which in this case just contains one function, `map`. Here we only see the type signature; to see the definition we tell HERMIT to focus on the binding:

```
hermit> binding-of 'map

map = λ △ △ f ds →
  case ds of wild ▲
    [] → [] ▲
    (:) a as → (:) ▲ (f a) (map ▲ ▲ f as)
```

Notice that the top-level pattern matching has been desugared into explicit lambdas and a case expression, and that the infix cons operator has been moved into a prefix position. Type arguments are displayed as triangles by HERMIT's default pretty printer, but the full details can displayed if desired. For this example we will not need to manipulate any type arguments, so we choose to hide them:

```
hermit> set-pp-type Omit
```

```
map = λ f ds →
  case ds of wild
    [] → []
    (:) a as → (:) (f a) (map f as)
```

To begin the calculation, we need to tell HERMIT that we want to reason about the map-fusion rule. We return to the top of the module, convert the map-fusion rule to a HERMIT lemma, and tell HERMIT that we wish to begin proving that lemma:

```
hermit> top
hermit> rule-to-lemma "map-fusion"
hermit> prove-lemma "map-fusion"

Goal:
  ∀ f g. (.) (map f) (map g) ≡ map ((.) f g)
```

Verifying this lemma in HERMIT requires the use of structural induction on the list data type (as per Figure 2). However, as the rule is written in a point-free style, there is initially no list argument to perform induction on. Thus we first apply *extensionality* to eta-expand the rule:

```
proof> extensionality 'xs

Goal:
  ∀ f g xs.
  (.) (map f) (map g) xs ≡ map ((.) f g) xs
```

We also unfold the composition operator, as this will provide us with a more convenient induction hypothesis:

```
proof> any-call (unfold '.)

Goal:
  ∀ f g xs.
  map f (map g xs) ≡ map (λ x → f (g x)) xs
```

We now begin the inductive part of the calculation. HERMIT provides structural induction as a lemma transformation: treating the current goal as the induction hypothesis, the goal is transformed into the conjunction of the base and inductive cases. That is, the inference rule in Figure 1 is instantiated such that its conclusion matches the current goal, and then the current goal is replaced by the premise of the rule.

```
proof> induction 'xs

Goal:
  ∀ f g.
  (map f (map g undefined)
   ≡
   map (λ x → f (g x)) undefined)
  ∧
  ((map f (map g []) ≡ map (λ x → f (g x)) [])
   ∧
   (∀ a b.
    (map f (map g b) ≡ map (λ x → f (g x)) b)
    ⇒
    (map f (map g ((:) a b))
     ≡
     map (λ x → f (g x)) ((:) a b))))
```

Here, the three clauses in the premise are the two base cases (for `undefined` and `[]`), and one inductive case (for `(:)`), as per Figure 2.

In each of the cases, two of the three occurrences of `map` are now applied to either `undefined` or an explicit list constructor. We need to unfold the definition of `map` and reduce the resultant expression in each of those two occurrences. Rather than doing this step by step, we build a *strategy* to perform these reductions in one step. We use the strategy combinators `any-bu` (anywhere, traversing bottom-up), `>>>` (sequencing) and `<+` (choice). These combinators are lifted from the strategic programming language KURE [34], which underlies HERMIT. Note that the sequencing strategy requires that both its component strategies succeed, which in this case ensures that occurrences of `map` are only unfolded if they can subsequently be reduced by `undefined-case` or `case-reduce`.

```
proof> any-bu (unfold 'map
                   >>> (undefined-case <+ case-reduce))

Goal:
  ∀ f g.
  (undefined ≡ undefined)
  ∧
  (([] ≡ [])
   ∧
   (∀ a b.
    (map f (map g b) ≡ map (λ x → f (g x)) b)
    ⇒
    ((:) (f (g a)) (map f (map g b))
     ≡
     (:) ((λ x → f (g x)) a) (map (λ x → f (g x)) b))))
```

Observe that both base cases have been reduced to evident equivalences. They can be eliminated altogether using HERMIT's `simplify-lemma` strategy. This strategy checks all equalities for α-equivalence, and reduces any it finds to a primitive truth clause. The strategy then attempts to apply a set of standard logical simplifications to eliminate connectives wherever possible (in this case, the unit law of conjunction is applied twice).

```
proof> simplify-lemma

Goal:
  ∀ f g a b.
  (map f (map g b) ≡ map (λ x → f (g x)) b)
  ⇒
  ((:) (f (g a)) (map f (map g b))
   ≡
   (:) ((λ x → f (g x)) a) (map (λ x → f (g x)) b))
```

Now all that remains is the inductive case. In HERMIT, when we navigate to the consequent of an implication, the antecedent becomes available as an assumed lemma. The HERMIT REPL displays all such in-scope antecedents to facilitate their use.

```
proof> forall-body ; consequent

Assumed lemmas:
ind-hyp-0 (Built In)
  map f (map g b) ≡ map (λ x → f (g x)) b
Goal:
  (:) (f (g a)) (map f (map g b))
  ≡
  (:) ((λ x → f (g x)) a) (map (λ x → f (g x)) b)
```

We now need to apply the induction hypothesis, which HERMIT has named `ind-hyp-0`. We could apply it in either direction, so we arbitrarily choose to apply it in a backwards direction:

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

module *MapFusion* **where**

import *Prelude* **hiding** (*map*)

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

$map :: (a \to b) \to [a] \to [b]$
$map\ f\ [\,] \quad\ = [\,]$
$map\ f\ (a : as) = f\ a : map\ f\ as$

Figure 3: Haskell source file `MapFusion.hs`.

```
Assumed lemmas:
ind-hyp-0 (Built In)
  map f (map g b) ≡ map (λ x → f (g x)) b
Goal:
  (:) (f (g a)) (map f (map g b))
  ≡
  (:) ((λ x → f (g x)) a) (map f (map g b))
```

All that remains is to perform a β-reduction. HERMIT's primitive `beta-reduce` transformation transforms a β-redex to a non-recursive let-binding, which can then be eliminated by inlining the binding. Rather than navigating to the redex and invoking these two transformations, we instead make use of HERMIT's general-purpose `simplify` strategy. This strategy repeatedly traverses a term, applying a set of basic simplification transformations until no more are applicable. Amongst others, this set includes `beta-reduce`, the elimination and inlining of let-bindings where the binding is used at most once, and the inlining of the definitions of several basic function combinators such as *id*, *const* and $(\circ)$.

```
proof> simplify

Assumed lemmas:
ind-hyp-0 (Built In)
  map f (map g b) ≡ map (λ x → f (g x)) b
Goal:
  (:) (f (g a)) (map f (map g b))
  ≡
  (:) (f (g a)) (map f (map g b))
```

The two sides are now equivalent, so the calculation is complete.

```
proof> end-proof

Successfully proven: map-fusion
```

The lemma is now available for use in further calculations. The sequence of reasoning steps that we performed can also be saved as a script, and thence re-run in future HERMIT sessions. This was a toy example, but we will now present two more realistic case studies, each of which contains a multitude of lemmas.

4. Case Study: Type-Class Laws

In this case study we use equational reasoning to verify that a number of type-class instances for common Haskell data types satisfy the expected type-class laws. We consider the laws in Figure 4. The data types we consider are lists, *Maybe*, and *Map* from the `containers` package, as well as *Identity* and *Reader* from the `transformers` package. Our approach was to state each law as a GHC rewrite rule, and load it into HERMIT as a lemma (as we did for the example in Section 3). We instantiated the laws for each data type, and then transformed the instantiated laws until HERMIT was satisfied that they held. Note that we used the actual data types and class instances defined in the `base`, `containers`, and `transformers` packages.

We present the case study as follows. Section 4.1 demonstrates the full details of verifying a single law. Section 4.2 then discusses some practical issues that arose as a consequence of using GHC Core as the object language. Section 4.3 describes how to modify the `containers` Cabal file to cause pre-written reasoning to be automatically loaded, checked and applied during compilation. Finally, Section 4.4 reflects on the overall success of the case study.

4.1 Example: return-left Monad Law for Lists

To give a flavour of the work involved in checking that a type-class law holds for a specific instance, we present the calculation for the return-left monad law for lists. The reasoning steps in this calculation involve more complex transformations than our Map Fusion example, which allows us to demonstrate the advantages of using KURE's strategy combinators for directing transformations.

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	$\forall\, u\ y\,.$	$u \circledast pure\ y \equiv pure\ (\lambda f \to f\ y) \circledast u$
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	$\forall\, j\ k\ l\,.$	$(j \gg\!= k) \gg\!= l \equiv j \gg\!= (\lambda x \to k\ x \gg\!= l)$
fmap-liftm	$\forall\, f\ x\,.$	$liftM\ f\ x \equiv fmap\ f\ x$

Figure 4: Laws used in the 'Type-Class Laws' case study.

In order to observe the effect of instantiation on the types of the lemma quantifiers, we begin by instructing HERMIT's pretty printer to display detailed type information. We then copy the general law, which has already been loaded from a rewrite-rule pragma, 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, we instantiate the type variable m 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.) The type of the dictionary binder has now been fully determined, so we instantiate it as well:

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

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

Next we note that the application of `return` can be simplified to a singleton list. We achieve this by unfolding `return`, which will expose a case expression that scrutinises the `$fMonad[]` dictionary. This can be simplified away by using HERMIT's `smash` strategy, which is a more aggressive version of the `simplify` strategy. This will leave the actual instance method defining *return* for lists, which can also be unfolded. Rather than doing this step by step, we direct HERMIT to focus on the application of `return` and repeatedly `unfold` and `smash` the expression. (The { } brackets limit the scope of the focus change.)

```
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 we need to simplify away the >>= application. Unfolding >>= directly results in a locally defined recursive worker named go, in terms of which the list instance of >>= is defined. Reasoning in the context of this recursive worker is tedious and brittle. We find it cleaner to state and verify the following pair of lemmas separately, then apply them as necessary during this proof:

bind-left-nil	$\forall\, k\,.$	$[\,] \gg\!= k \equiv [\,]$
bind-left-cons	$\forall\, x\ xs\ k\,.$	$(x : xs) \gg\!= k \equiv k\ x +\!\!+ (xs \gg\!= k)$

```
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
```

To eliminate the list append we appeal to another auxiliary lemma, which can itself be verified by straightforward induction.

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

We apply append-right to complete the calculation:

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

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

proof> end-proof
```

4.2 Reasoning in GHC

Equational reasoning in HERMIT is performed in the context of GHC Core. While this is a small, relatively stable, typed intermediate language, it was designed for compilation and optimisation, not equational reasoning. Consequently, there are a few practical concerns and limitations regarding reasoning with this language.

4.2.1 Implications

The *Monoid* instance for *Maybe a* requires a *Monoid* instance to exist for the type a:

instance *Monoid* $a \Rightarrow$ *Monoid* (*Maybe* a) **where** ...

Correspondingly, the calculation to verify that the monoid associativity lemma holds for *Maybe a* relies on the monoid associativity lemma for a. Thus, we used the following lemma for *Maybe*:

mappend-assoc-impl

$$\forall\, m\,.\,(\forall\,(x :: m)\ y\ z\,.\,(x \diamond y) \diamond z \equiv x \diamond (y \diamond z))$$
$$\Rightarrow$$
$$(\forall\,(i :: Maybe\ m)\ j\ k\,.\,(i \diamond j) \diamond k \equiv i \diamond (j \diamond k))$$

HERMIT cannot generate such an implication lemma from the original mappend-assoc lemma automatically. Although it can spot the superclass constraint, the associated laws are not part of the Haskell language, and thus are not available within GHC. Instead, we constructed the implication lemma ourselves using HERMIT's transformation DSL.

```
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,
                   hermit == 1.0.*
    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
```

Figure 5: Additions to the Cabal configure file for `containers` in order to automatically re-run the HERMIT scripts.

4.2.2 Newtypes

GHC's **newtype** declaration offers the benefits of type abstraction with no runtime overhead [4, 43]. This is accomplished by implementing **newtype** constructors in GHC Core as type casts around the enclosed expression, rather than as algebraic data constructors. These casts are erased before code generation.

Reasoning in the presence of **newtype**s must deal with these casts explicitly. HERMIT's `smash` strategy attempts to float-out and eliminate type casts where possible, and was effective at doing so in the majority of our examples. In the few cases where the `smash` strategy did not eliminate all the casts, the resultant expressions were still α-equivalent and thus this did not pose a problem.

4.3 Configuring Cabal

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

We co-opt this ability to direct HERMIT to run and check our scripts whenever the package is rebuilt. As an example, we added a `laws/` directory to the `containers` package, containing three files. The first is `Laws.hs`, which provides the RULES pragmas representing the type-class laws. The other two files are the HERMIT scripts: one for the functor-law calculations, the other for the monoid-law calculations.

We then added a `Test-suite` section, seen in Figure 5, 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 does not change the dependencies of the `containers` library itself.

Cabal runs the HERMIT scripts by providing GHC with the required series of flags. The `-fexpose-all-unfoldings` flag was described in Section 2.3. The `-fplugin=HERMIT` flag invokes 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 HERMIT scripts should also be added to the configuration file's `extra-source-files` section, so that they are included in source distributions. In order to run the scripts, we use the normal Cabal testing work-flow:

```
> cabal configure --enable-tests
> cabal build
```

Note that we do not actually have to run the generated test, as the HERMIT scripts are run at compile time.

Table 1: Script lengths in the ‘Type-Class Laws’ case study.

Law (see Fig. 4)	Data Type				
	List	Maybe	Map	Identity	Reader
mempty-left	7	5	7		
mempty-right	6	5	7	N/A	
mappend-assoc	15	15	-		
fmap-id	9	7	12	5	10
fmap-distrib	10	8	16	5	10
identity	7	8		5	15
homomorphism	8	5		5	15
interchange	18	5		5	15
composition	23	5	N/A	5	15
fmap-pure	20	5		5	15
return-left	9	5		5	12
return-right	18	5		5	12
bind-assoc	14	5		5	12

4.4 Reflections

Results for the case study are listed in Table 1, and the complete set of HERMIT scripts are available online [15]. The numbers in the table represent number of lines in the HERMIT script, including instantiation steps. Overall, verifying type-class laws in GHC Core appears to be viable with the simple reasoning techniques offered by HERMIT.

In general, we found that reasoning about type classes and dictionaries proceeded in much the same way as the example in Section 4.1. Handling class methods requires many unfolding and simplification steps. Once this is done, any required inductive reasoning tends to be short and straightforward.

The one law we did not verify was mappend-assoc for the Map data type. This was not because of any technical limitation of HERMIT, but rather because the required reasoning steps were not obvious to us. The $\mathit{mappend}$ operation for Map is an efficient left-biased hedged union whose implementation relies on several functions that maintain invariants of the underlying data structure. We expect that this law could be verified by a user with a better understanding of these functions and the invariants they maintain.

Unsurprisingly, stating smaller auxiliary lemmas for (re-)use in the larger calculations helped to manage complexity. In contrast, our initial attempts to perform the larger calculations directly required working at a lower level, and led to a substantial amount of duplicated effort. This was especially true of the $\mathit{Applicative}$ laws, as the $\mathit{Applicative}$ instances were often defined in terms of their Monad counterparts. In the case of lists, naively unfolding $\gg\!=$ results in a local recursive worker function. Reasoning in the presence of such workers requires many tedious unfolding and let-floating transformations. Using auxiliary lemmas about $\gg\!=$ allowed us to avoid this tedium.

We did not attempt to quantify the robustness of the HERMIT scripts with respect to changes in the underlying source code. The types and instances that we considered are standard and relatively stable over time. However, as most of the calculations were fairly heavy on unfolding and simplification, we expect they would be sensitive to changes. To lower the burden of amending existing scripts, HERMIT’s interactive mode allows a user to pause a script midway, and to step backwards and forwards.

Configuring a Cabal package to re-run scripts on recompilation is straightforward, requiring a single additional section to a package’s Cabal configuration file. End users of the package can still build and install the package as before, but the HERMIT scripts can be checked by enabling the package tests.

5. Case Study: Making a Century

To assess how well HERMIT supports general-purpose equational reasoning, we decided to mechanise some existing textbook reasoning as a case study. We selected the chapter *Making a Century* from the textbook *Pearls of Functional Algorithm Design* [2, Chapter 6]. The book is dedicated to reasoning about Haskell programs, with each chapter calculating an efficient program from an inefficient specification program. The program in *Making a Century* computes the list of all the ways that 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 solution is:

$$12 + 34 + 5 \times 6 + 7 + 8 + 9 = 100$$

The details of the program are not overly important to the presentation of our case study, and we refer the interested reader to the textbook for details [2, 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 reasoning techniques, including fold/unfold transformation [5], structural induction (Section 2.4), and fold fusion [25].

We will not present the entire case study in this paper. Instead, we will give a representative extract, and then discuss the aspects of the mechanisation that proved challenging. The HERMIT scripts for the complete case study are available online [15].

5.1 HERMIT Scripts

After creating a Haskell file containing the definitions from the textbook, our next task was to introduce the lemmas used in the equational reasoning. The main lemmas (specifically, those that are named in the textbook) are displayed in Figure 6, which should give an impression of their complexity. The majority of these lemmas are equivalences between expressions, so we were able to introduce them via rewrite rules in the Haskell source file (see Section 2.1). The one exception was Fold Fusion, which we introduced using HERMIT’s transformation DSL. Lemma 6.5 is also a composite lemma, but we found it more convenient to introduce a pair of lemmas rather than constructing an explicit conjunction.

Throughout this case study, we took a consistent approach to mechanising the equational reasoning in the textbook. For each lemma, we first worked step-by-step in HERMIT’s interactive mode, and then, when the calculation was complete, saved it as a script that could be invoked thereafter. We took the same approach to the main program transformation ($\mathit{solutions}$), invoking the lemmas as necessary. Roughly half of the HERMIT equational reasoning in this case study was transliterated from the textbook equational reasoning, and the remaining half was calculations that we developed for ourselves (see Section 5.3). Both halves proceeded in a similar manner, but with more experimentation and backtracking during the interactive phases for the latter.

As an example, compare the calculations to verify Lemma 6.8. Figure 7a presents the calculation extracted verbatim from the textbook [2, Page 36], and Figure 7b presents the corresponding HERMIT script. Note that lines beginning “--” in a HERMIT script are *comments*, and for readability we have typeset them 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 working interactively. When generating a script after an interactive session, HERMIT can insert these comments if desired. The content of the comments can be configured by various pretty-printer modes — in this case we have opted to have HERMIT omit the type arguments (as in Section 3) to improve the correspondence with the textbook extract.

The main difference between the two calculations is that in HERMIT we must specify where in the term we are working, and in which direction lemmas are applied. In contrast, in the textbook the

Fold Fusion	$\forall\, f\ g\ h\ a\ b\,.$	$(f\ \mathit{undefined} \equiv \mathit{undefined} \quad \wedge \quad f\ a \equiv b \quad \wedge\ \forall\, x\ y\,.\ f\ (g\ x\ y) \equiv h\ x\ (f\ y)) \quad \Rightarrow \quad f \circ \mathit{foldr}\ g\ a \equiv \mathit{foldr}\ h\ b$
Lemma 6.2		$\mathit{filter}\ (\mathit{good} \circ \mathit{value}) \equiv \mathit{filter}\ (\mathit{good} \circ \mathit{value}) \circ \mathit{filter}\ (\mathit{ok} \circ \mathit{value})$
Lemma 6.3	$\forall\, x\,.$	$\mathit{filter}\ (\mathit{ok} \circ \mathit{value}) \circ \mathit{extend}\ x \equiv \mathit{filter}\ (\mathit{ok} \circ \mathit{value}) \circ \mathit{extend}\ x \circ \mathit{filter}\ (\mathit{ok} \circ \mathit{value})$
Lemma 6.4	$\forall\, x\,.$	$\mathit{map}\ \mathit{value} \circ \mathit{extend}\ x \equiv \mathit{modify} \circ \mathit{map}\ \mathit{value}$
Lemma 6.5	$\forall\, f\ g\,.$	$\mathit{fst} \circ \mathit{fork}\ (f, g) \equiv f \quad \wedge \quad \mathit{snd} \circ \mathit{fork}\ (f, g) \equiv g$
Lemma 6.6	$\forall\, f\ g\ h\,.$	$\mathit{fork}\ (f, g) \circ h \equiv \mathit{fork}\ (f \circ h, g \circ h)$
Lemma 6.7	$\forall\, f\ g\ h\ k\,.$	$\mathit{fork}\ (f \circ h, g \circ k) \equiv \mathit{cross}\ (f, g) \circ \mathit{fork}\ (h, k)$
Lemma 6.8	$\forall\, f\ g\,.$	$\mathit{fork}\ (\mathit{map}\ f, \mathit{map}\ g) \equiv \mathit{unzip} \circ \mathit{map}\ (\mathit{fork}\ (f, g))$
Lemma 6.9	$\forall\, f\ g\,.$	$\mathit{map}\ (\mathit{fork}\ (f, g)) \equiv \mathit{zip} \circ \mathit{fork}\ (\mathit{map}\ f, \mathit{map}\ g)$
Lemma 6.10	$\forall\, f\ g\ p\,.$	$\mathit{map}\ (\mathit{fork}\ (f, g)) \circ \mathit{filter}\ (p \circ g) \equiv \mathit{filter}\ (p \circ \mathit{snd}) \circ \mathit{map}\ (\mathit{fork}\ (f, g))$

Figure 6: Main lemmas in the ‘Making a Century’ case study.

$$
\begin{array}{ll}
 & \mathit{unzip} \cdot \mathit{map}\ (\mathit{fork}\ (f, g)) \\
= & \{\ \text{definition of } \mathit{unzip}\ \} \\
 & \mathit{fork}\ (\mathit{map}\ \mathit{fst}, \mathit{map}\ \mathit{snd}) \cdot \mathit{map}\ (\mathit{fork}\ (f, g)) \\
= & \{\ (6.6) \text{ and } \mathit{map}\ (f \cdot g) = \mathit{map}\ f \cdot \mathit{map}\ g\ \} \\
 & \mathit{fork}\ (\mathit{map}\ (\mathit{fst} \cdot \mathit{fork}\ (f, g)), \mathit{map}\ (\mathit{snd} \cdot \mathit{fork}\ (f, g))) \\
= & \{\ (6.5)\ \} \\
 & \mathit{fork}\ (\mathit{map}\ f, \mathit{map}\ g)
\end{array}
$$

(a) Textbook extract.

```
-- forall 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 7: Comparison of the textbook calculation with the HERMIT script for Lemma 6.8.

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. Here, `forall-body` and `eq-rhs` are navigation commands that direct HERMIT to descend into the body of the universal quantifier, and then into the right-hand side of the equivalence. `one-td` (once, traversing top-down) and `any-td` (anywhere, traversing top-down) are *strategy combinators* lifted from KURE [34].

In this calculation, and most others in the case study, we think that the HERMIT scripts are about as clear, and not much more verbose, than the textbook calculations. There was one exception though: manipulating the function-composition operator.

5.2 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 term before a transformation can be applied.

For function composition, one way to avoid this problem is to unfold all occurrences of the composition operator and work with the η-expanded terms, as this always produces an abstract syntax tree consisting of a left-nested sequence of applications. However, we did not do so for this case study because the textbook calculations are written in a point-free style, and we wanted to match them as closely as possible.

More generally, rewriting terms containing associative (and commutative) operators is a well-studied problem [e.g. 3, 10, 23], and it remains as future work to provide better support for manipulating such operators in HERMIT.

5.3 Assumed Lemmas in the Textbook

As is common with pen-and-paper reasoning, several properties that are used in the textbook are assumed without an accompanying proof being given. This included some of the named lemmas from Figure 6, as well as several auxiliary properties, some explicit and some implicit (Figure 8). While performing reasoning beyond that presented in the textbook was not intended to be part of the case study, we decided to attempt to verify these properties in HERMIT.

Of the assumed named lemmas, Fold Fusion 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 verifying them in HERMIT mostly consisted of unfolding definitions and simplifying the resultant expressions, with the occasional basic use of induction. The same was true for the auxiliary lemmas, which we list in Figure 8. Systematic calculations such as these are ripe for mechanisation, and HERMIT provides several strategies that perform a suite of basic simplifications to help with this. Consequently, the HERMIT scripts were short and concise.

Lemmas 6.2, 6.3 and 6.4 were more challenging. For Lemma 6.2 we found it helpful to introduce the `filter-split` auxiliary lemma (Figure 8), which we consider to capture the essence of the key optimisation in the case study. After this, the calculation was fairly straightforward. However, we found Lemmas 6.3 and 6.4 to be non-trivial properties, without (to us) obvious proofs, and so we did not verify them in HERMIT. This did not inhibit the rest of the case study, as HERMIT allows an unverified lemma to be taken as an assumption. If such assumed lemmas are used in a calculation, by default 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 verified lemmas are used.

Finally, the simplification of the definition of *expand* is stated in the textbook without presenting any intermediate transformation steps [2, Page 40]. It is not obvious to us what those intermediate transformation steps would be, and thus we did not encode this simplification in HERMIT.

5.4 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

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	$\forall\, p\ q\,.$	$(\forall\, x\,.\ q\ x \equiv False\ \Rightarrow\ p\ x \equiv False) \Rightarrow filter\ p \equiv filter\ p \circ filter\ q$

Figure 8: Auxiliary lemmas in the 'Making a Century' case study.

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 existing definitions, and does not support this style of reasoning; so we were unable to replicate this calculation. However, we were able to *verify* the calculation by working in reverse: starting from the definition of *expand*, we proceeded to validate the use of the fold-fusion law by checking the corresponding pre-conditions.

5.5 Calculation Sizes

As exemplified by Figure 7, the HERMIT 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. We present some statistics in Table 2, but we recommend not extrapolating anything from them beyond a rough approximation of the scale of the calculations. We give the size of the two main calculations (transforming *solutions* and deriving *expand*), as well as those for the named lemmas. In the textbook we measure 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, we measure 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. We do not measure commands for stating lemmas, loading files, switching between transformation and proof mode, or similar, as we consider these comparable to the surrounding discussion in the textbook. To get a feel for the scale of the numbers given, we recommend that the user compares the numbers for Lemma 6.8 in Table 2 to the calculation in Figure 7.

5.6 Reflections

Our overall experience was that mechanising the textbook reasoning was fairly straightforward, and it was pleasing that we could translate most steps of the textbook calculations into an equivalent HERMIT command. The only annoyance was the occasional need to manually apply lemmas for manipulating operator associativity (see Section 5.2) so that the structure of the term would match the transformation we were applying.

While having to specify where in a term each lemma must be applied does result in scripts that are a little more verbose than the textbook calculations, we do not necessarily consider that to be detrimental. Rather, we view a pen-and-paper calculation that does not specify the location as passing on that work to the reader, who must determine for herself where, and in which direction, a lemma is intended to be applied. Furthermore, when desired, strategy combinators can be used to avoid specifying precisely which sub-term the lemma should be applied to.

Table 2: Comparison of calculation sizes in 'Making a Century'.

Calculation	Textbook Lines	HERMIT Commands	
		Transformation	Navigation
Fold Fusion	assumed	19	20
Lemma 6.2	assumed	7	2
Lemma 6.3	assumed	assumed	
Lemma 6.4	assumed	assumed	
Lemma 6.5	assumed	6	4
Lemma 6.6	assumed	2	1
Lemma 6.7	assumed	3	1
Lemma 6.8	7	5	6
Lemma 6.9	1	4	4
Lemma 6.10	assumed	12	16
solutions	16	13	12
expand	19	22	21

During the case study we also discovered one error in the textbook. Specifically, the inferred type of the *modify* function [2, Page 39] does not match its usage in the program. We believe that its definition should include a *concatMap*, which would correct the type mismatch and give the program its intended semantics, so we have modified the function accordingly in our source code.

6. Related Work

There have been three main approaches taken to verifying properties of Haskell programs: testing, automated theorem proving, and equational reasoning. The most prominent testing tool is QuickCheck [6], which automatically generates large quantities of test cases in an attempt to find a counterexample. Other testing tools include SmallCheck [31], which exhaustively generates test values of increasing size so that it can find minimal counterexamples, and Lazy SmallCheck [28, 31], which also tests partial values. Jeuring et al. [22] have recently developed infrastructure to support using QuickCheck to test type-class laws, as well as to test the individual steps of user-provided equational reasoning.

There are several tools that attempt to automatically prove properties of Haskell programs by interfacing with an automated theorem prover. These include Liquid Haskell [41, 42], Zeno [36], HALO [44], and the Haskell Inductive Prover (Hip) [30]. The general approach taken by these tools is to translate the Haskell program, via GHC Core, into a first-order logic. User-stated program properties are then checked by passing them to an external theorem prover for verification. For inductive proofs, these tools provide their own automated induction principle(s), which then invoke the external theorem prover as required. Another similar tool is HipSpec [7], which is built on top of Hip. 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. Thus user-stated properties are optional: HipSpec can check user-stated properties against those it has inferred, but alternatively it can just generate program properties as documentation.

Equational reasoning is used both to verify properties of Haskell programs and to validate the correctness of program transformations. Most equational reasoning about Haskell programs is performed manually with pen-and-paper or text editors, of which there are numerous examples in the literature [e.g. 2, 8, 17, 19, 25]. Prior to HERMIT there have been several tools for *mechanical* equational reasoning on Haskell programs, including the Haskell Equational Reasoning Assistant (HERA) [18], the Ulm Transformation System (Ultra) [21], and the Programming Assistant for Transforming Haskell (PATH) [40].

HERA was our own preliminary tool, and was a direct predecessor of HERMIT. HERA operated on Haskell source code, via Template Haskell [35]. However, the lack of typing information proved an obstacle to many non-syntactic transformations, such as worker/wrapper [19]. This was the primary reason for our switch to GHC Core when designing the HERMIT system, although the large size of the Template Haskell grammar was another consideration.

Ultra has much in common with HERMIT in terms of functionality and available transformations. The main distinction is that Ultra operates on the source code of its own Haskell-like language. This language is a sub-language of Haskell 98 (notably excluding type classes), extended with non-executable descriptive operators (e.g. "there exists a value such that..."). The idea of the descriptive operators is to allow a user to express concise high-level specifications, which can then be transformed into executable programs. This differs from HERMIT, where our starting point is a valid GHC Core program. Ultra also comes with built-in support for a variety of algebraic structures (e.g. monoids), which makes reasoning about binary operators smoother than in HERMIT.

PATH is also based on transforming a small Haskell-like language, called PATH-L. The PATH tool automatically translates from a sub-language of Haskell 98 (excluding type classes, among other things) to PATH-L. The user then performs equational reasoning on the PATH-L program, and finally PATH automatically converts the resultant program back to Haskell. PATH was designed with an emphasis on *total correctness*, and all PATH transformations are guaranteed not to introduce non-termination, even in the presence of infinite or partial values.

Another tool similar to HERMIT is the Haskell Refactorer (HaRe) [24, 39], which supports user-guided refactoring of Haskell programs. However, the objective of HaRe is slightly different, as refactoring is concerned with program transformation, whereas HERMIT supports both transformation of programs and verification of program properties. The original version of HaRe targets Haskell 98 source code, but recently work has begun on a re-implementation of HaRe that targets GHC-extended Haskell.

7. Conclusions

We have presented two case studies of using HERMIT to perform equational reasoning on GHC Core programs. The first case study demonstrated that it is viable to verify type-class laws using HERMIT. The HERMIT scripts were uniformly brief, and predominantly consisted of unfolding definitions and simplification, with relatively simple reasoning steps. Additionally, we note that while we focused on type-class laws in that case study, the same approach can be used to verify GHC rewrite-rule pragmas.

HERMIT now provides structural induction as a built-in transformation, and supports transformations that have preconditions, such as the fold-fusion law (used in our second case study) and the worker/wrapper transformation [19, 32]. In a prior publication [33] we described encoding the worker/wrapper transformation in HERMIT, and used it to optimise a variety of example programs. However, at the time HERMIT had no means of verifying the preconditions, so they were not mechanically checked. Using HERMIT's new equational-reasoning infrastructure, we have updated the worker/wrapper encoding such that user scripts verifying the preconditions are checked before the transformation can be applied. All of the preconditions for the examples in that previous publication have now been verified in HERMIT, and the corresponding scripts are bundled with the HERMIT package [14].

The case studies did highlight HERMIT's need for a good parser for GHC Core expressions and types. GHC once specified an External Core format, including a parser and pretty-printer, but External Core has recently been removed because it was not being maintained. HERMIT already features a strong pretty-printing capability, but better parsing facilities would make working interactively with GHC Core much simpler for the user.

In the past [1, 13] HERMIT has been used to successfully prototype GHC optimisations by encoding them as sequences of transformation steps. Now HERMIT can also be used to reason about any preconditions those transformation steps have, as well as to provide mechanical assurances about equational reasoning that is intended to prove properties of Haskell programs, including type-class laws for instance declarations, and user optimisations stated as GHC rewrite rules. By applying and checking the user's reasoning during compilation, HERMIT enforces a connection between the source code, the reasoning steps, and the compiled program. GHC plugins developed using HERMIT can then be deployed with Haskell's Cabal packaging system, meaning that they integrate with a developer's normal work-flow. HERMIT development is ongoing, and we seek to target ever-larger examples.

Acknowledgments

We thank Patrik Jansson and the anonymous reviewers for their useful comments and suggestions. This material is based upon work supported by the National Science Foundation under Grant No. 1117569.

Appendix

To give an idea of the complexity of defining composite lemmas using HERMIT's transformation DSL, we present here the encoding of the filter-split lemma from the 'Making a Century' case study.

filter-split
$$\begin{array}{ll} \forall\, p\ q\,. & (\forall\, x\,.\ \ q\ x \equiv \mathit{False}\ \Rightarrow\ p\ x \equiv \mathit{False}) \\ \quad\Rightarrow & \\ & \mathit{filter}\ p \equiv \mathit{filter}\ p \circ \mathit{filter}\ q \end{array}$$

We construct this lemma by building the corresponding GHC Core terms. This involves looking up the *filter* and (∘) functions, generating the universally quantified variables (of the correct types), and then constructing the GHC Core expressions and HERMIT lemma. To assist with this, HERMIT provides an assortment of smart constructors, including $$ for expression application and ⟹ and ═══ for logical implication and equivalence. Note that implication lemmas are annotated with a name for the antecedent, so that it can be referred to when reasoning about the consequent.

```
filterSplitLemma :: LemmaLibrary
filterSplitLemma = do
  filterId ← findIdT "filter"
  compId ← findIdT "."
  constT $ do
    a           ← newTyVar "a"
    let aTy     = mkTyVarTy a
    p           ← newVar "p" (aTy ⟶ boolTy)
    q           ← newVar "q" (aTy ⟶ boolTy)
    x           ← newVar "x" aTy
    qx          ← q $$ x
    px          ← p $$ x
    filterp     ← filterId $$ p
    filterq     ← filterId $$ q
    filterpcomp ← compId $$ filterp
    filterpq    ← filterpcomp $$ filterq
    return $ newLemma "filter-split" $
      mkForall [a, p, q] $
          ("filter-split-antecedent",
           mkForall [x] (("qx-False", qx ═══ falseDataConId)
                              ⟹ (px ═══ falseDataConId)))
        ⟹
          filterp ═══ filterpq
```

While this is not an ideal way of constructing composite lemmas, HERMIT can check that only well-typed GHC Core terms are produced, which is effective at catching mistakes. Furthermore, note that a lemma introduced in this way is not treated as an axiom any more than a lemma introduced via other means: the user must still verify the lemma using equational reasoning inside HERMIT.

As future work, we aim to provide a parser for an extension of GHC rewrite-rule syntax, so that composite lemmas can be written in a familiar Haskell-like syntax, with type and dictionary arguments being automatically inferred and inserted.

References

[1] M. D. Adams, A. Farmer, and J. P. Magalhães. Optimizing SYB is easy! In *Workshop on Partial Evaluation and Program Manipulation*, pages 71–82. ACM, 2014.

[2] R. Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.

[3] T. Braibant and D. Pous. Tactics for reasoning modulo AC in Coq. In *International Conference on Certified Programs and Proofs*, volume 7086 of *LNCS*, pages 167–182. Springer, 2011.

[4] 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, 2014.

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

[6] 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.

[7] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In *International Conference on Automated Deduction*, volume 7898 of *LNCS*, pages 392–406. Springer, 2013.

[8] 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 *LNCS*, pages 85–109. Springer, 2004.

[9] A. L. de M. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995.

[10] 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.

[11] A. Farmer. *HERMIT: Mechanized Reasoning during Compilation in the Glasgow Haskell Compiler*. PhD thesis, University of Kansas, 2015.

[12] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In *Haskell Symposium*, pages 1–12. ACM, 2012.

[13] 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.

[14] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. `http://hackage.haskell.org/package/hermit`, 2015.

[15] A. Farmer, N. Sculthorpe, and A. Gill. Hermit case studies: Proving Type-Class Laws & Making a Century, 2015. URL `http://www.ittc.ku.edu/csdl/fpg/HERMIT/case-studies-2015/`.

[16] GHC Team. *GHC User's Guide, Version 7.8.4*, 2014. URL `http://downloads.haskell.org/~ghc/7.8.4/docs/html`.

[17] J. Gibbons and G. Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae*, 66(4):353–366, 2005.

[18] A. Gill. Introducing the Haskell equational reasoning assistant. In *Haskell Workshop*, pages 108–109. ACM, 2006.

[19] A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, 2009.

[20] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris Diderot, 1972.

[21] 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.

[22] J. Jeuring, P. Jansson, and C. Amaral. Testing type class laws. In *Haskell Symposium*, pages 49–60. ACM, 2012.

[23] 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.

[24] 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 *ENTCS*, pages 29–34. Elsevier, 2005.

[25] 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 *LNCS*, pages 124–144. Springer, 1991.

[26] S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming in Agda: Dependent types for relational program derivation. *Journal of Functional Programming*, 19(5):545–579, 2009.

[27] 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.

[28] 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 *LNCS*, pages 53–70. Springer, 2013.

[29] J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *LNCS*, pages 408–423. Springer, 1974.

[30] D. Rosén. Proving equational Haskell properties using automated theorem provers. Master's thesis, University of Gothenburg, 2012.

[31] 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.

[32] N. Sculthorpe and G. Hutton. Work it, wrap it, fix it, fold it. *Journal of Functional Programming*, 24(1):113–127, 2014.

[33] N. Sculthorpe, A. Farmer, and A. Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *International Symposium on Implementation and Application of Functional Languages*, volume 8241 of *LNCS*, pages 86–103. Springer, 2013.

[34] 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.

[35] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In *Haskell Workshop*, pages 1–16. ACM, 2002.

[36] 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 *LNCS*, pages 407–421. Springer, 2012.

[37] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2007.

[38] J. Tesson, H. Hashimoto, Z. Hu, F. Loulergue, and M. Takeichi. Program calculation in Coq. In *Algebraic Methodology and Software Technology*, volume 6486 of *LNCS*, pages 163–179. Springer, 2011.

[39] S. Thompson and H. Li. Refactoring tools for functional languages. *Journal of Functional Programming*, 23(3):293–350, 2013.

[40] M. Tullsen. *PATH, A Program Transformation System for Haskell*. PhD thesis, Yale University, 2002.

[41] N. Vazou, E. L. Seidel, and R. Jhala. LiquidHaskell: Experience with refinement types in the real world. In *Haskell Symposium*, pages 39–51. ACM, 2014.

[42] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton Jones. Refinement types for Haskell. In *International Conference on Functional Programming*, pages 269–282. ACM, 2014.

[43] D. Vytiniotis and S. Peyton Jones. Evidence normalization in System FC. In *International Conference on Rewriting Techniques and Applications*, pages 20–38. Schloss Dagstuhl, 2013.

[44] 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.

[45] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Symposium on Principles of Programming Languages*, pages 60–76. ACM, 1989.