Making a Century in HERMIT

Extended Abstract

Neil Sculthorpe

Computer Science Department
Swansea University
{N.A.Sculthorpe}@swansea.ac.uk

Andrew Farmer Andrew Gill

Information and Telecommunication Technology Center
The University of Kansas
{afarmer,andygill}@ittc.ku.edu

Abstract

A benefit of pure functional programming is that it encourages equational reasoning. However, the Haskell language currently lacks 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). 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 extended abstract presents a detailed case study of HERMIT usage in practice: mechanising Bird's classic "Making a Century" pearl. We also use the mechanised pearl to introduce recent HERMIT developments for supporting for 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, Optimisation

1. Introduction

Currently, most equational reasoning on Haskell programs is performed manually, using pen-and-paper or text editors, because of the lack of up-to-date tool support. While some equational-reasoning tools do exist for Haskell [14, 35], they either target Haskell 98 or some subset thereof, and have not attempted to keep pace with the (frequently advancing) GHC-extended version of Haskell that is widely used in practice. They also work at the syntactical level, without access to the results of the type inference performed by the Haskell compiler. 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 [7]. This was recently demonstrated by Jeuring et al. [18], 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 [22] or Coq [33]. The desired transformations and proofs can then be performed in that language, and the resultant program or property transliterated back into Haskell. However, the semantics of these languages differ from Haskell, sometimes in subtle ways, so the transformations and proofs used may not carry over to Haskell. Again, partial and infinite values are a particular concern.

To address this situation, we have implemented a GHC plugin called HERMIT [9, 10, 29]. HERMIT is a toolkit that supports interactive equational reasoning on Haskell programs, and the mechanical verification of proof scripts. HERMIT operates on GHC's internal core language, part-way through the compilation process. User proofs of program properties are checked, and user-specified transformations are applied to the program being compiled. By performing proof checking during compilation, HERMIT ensures not only that the proof is correct, but that it corresponds to the current implementation of the program, in the context of the language extensions currently being used.

The initial HERMIT implementation [9] 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 [29], as well as for encoding prototypes of new optimisation techniques [1, 11]. 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 verified by hand. Furthermore, it was not possible to state and prove auxiliary lemmas, or to use inductive proof techniques. This extended abstract describes the addition of these facilities to HERMIT, and discusses our experiences of using them on a case study. Specifically, the two main contributions of this extended abstract are:

- We describe the new equational reasoning infrastructure provided by HERMIT, discussing the challanges that arose during implementation, and the design choices we made for providing equational-reasoning support to Haskell programmers. (Section 2).
- Using our new infrastructure, we present a case study of equational reasoning in HERMIT, by mechanising a chapter from *Pearls of Functional Algorithm Design* [2], a recent textbook about deriving Haskell programs by calculation (Section 3).

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 [16, 25] extended with let-binding, constructors, and first-class type equalities [32]. Type checking is performed during the translation, and GHC Core retains the typing information as annotations.

HERMIT provides commands for navigating a GHC Core abstract syntax tree, applying transformations, version control, selecting different pretty printers, and invoking GHC analyses and optimisation passes. To direct and combine transformations, HERMIT uses the strategic programming language KURE [30] to provide a family of rewriting 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.
- 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.
- A domain-specific language for transformation [30], 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 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.

This extended abstract describes HERMIT's new equational reasoning infrastructure, but will not otherwise discuss its implementation or existing commands. Interested readers should consult the previous HERMIT publications [9, 29], or try out the HERMIT toolkit [10] for themselves.

2.1 Stating and Proving Lemmas

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. 29]. This was limiting, as it is often necessary to state and prove auxiliary lemmas, which can then be used to validate the transformation steps.

To address this, we have added support for auxiliary lemmas in HERMIT. As encoding a complete logic in HERMIT is a substantial task, we have begun with the simplest form of lemma that allows us to perform some interesting equational reasoning. A HERMIT lemma is (currently) an equality between two GHC Core expressions, which may contain universally quantified variables. For example:

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

HERMIT maintains a set of lemmas, and records which have been proven and which have not. Proven lemmas can be applied as transformations (left-to-right or right-to-left), or used to validate transformation steps that have preconditions. A user can prove a lemma by providing a sequence of transformations on either (or both) sides of the lemma. HERMIT then checks the proof by comparing both sides of the transformed lemma using alpha equality. This proof can either be performed interactively, or loaded from a script.

Currently, we generate lemmas by exploiting GHC rewrite-rule pragmas [23]. For example, the Map Fusion lemma above can be expressed using the following RULES pragma:

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

HERMIT previously allowed any rewrite rule in scope to be utilised directly as a unidirectional HERMIT transformation [9]. Any such rule can now also be converted into a HERMIT lemma, and thence proved. A side-benefit of this is that a user can experiment with applying GHC rewrite rules backwards, which was not possible previously. Note that there are some restrictions on the form of the left-hand-side of a GHC rewrite rule [23, Section 2.2], so this approach can only generate a subset of all possible lemmas.

Rewrite rules that are not intended to be used by GHC's optimiser can be annotated with the notation `[~]`, as we did for map-fusion above. This causes GHC to consider the rule as always inactive, and never attempt to use it for optimisation [12, Section 7.21.1]. In the long term, we aim to add a specific HERMIT pragma to GHC, allowing HERMIT lemmas to be stated in the source file yet be clearly distinguished from any rewrite rules. We could then be more liberal with the lemmas that can be stated than the limitations of GHC rewrite rules.

2.2 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, consider this standard definition of list concatenation:

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

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.

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. There are several induction principles that are relevant to Haskell programs. Thus far, we have encoded only one such principle in HERMIT: structural induction. This is implemented as a built-in proof technique that can be used to prove a lemma. Structural induction has been sufficient to prove all of the lemmas in our cases studies, but we anticipate that we will need to add other forms of induction when attempting more complex examples. The remainder of this section will formalise the structural-induction inference rule that HERMIT provides.

We first introduce some notation. Given $a_1, a_2 :: A$ for any type A, then let $a_1 \equiv a_2$ denote that a_1 and a_2 are semantically equivalent. We write $\overrightarrow{vs}$ to denote a sequence of variables, and $\forall(C\ \overrightarrow{vs} :: A)$ to quantify over all constructors C of the 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:

- the induction hypothesis holds for undefined values;

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

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

Figure 1: Structural induction.

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

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

Figure 2: Structural induction on lists.

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

As a 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. While this is sufficient for the case study we describe in this extended abstract, it does not allow inductive proofs over recursive types where the recursion is deeper in the data type. The following type of *rose trees* is one such type, having a recursive call that is two constructors deep:

$\mathbf{data}\ \mathit{RoseTree}\ a = \mathit{Node}\ a\ [\mathit{RoseTree}\ a]$

As future work we need to generalise HERMIT's structural induction principle to n constructors deep.

3. 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 entirely dedicated to reasoning about Haskell programs, with each chapter calculating an efficient program from an inefficient specification program. Additionally, many of the transformation steps used have preconditions, and thus there are several proof obligations along the way.

The program in *Making a Century* computes the list of all arithmetic expressions formed from ascending digits, where juxtaposition, addition, and multiplication evaluate to 100. For example, one possible solution is

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

The details of the program are not overly important to the 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 proof techniques, including fold/unfold transformation [4], structural induction (Section 2.2), fold fusion [21], and numerous auxiliary lemmas.

We will not present the entire case study here. Instead, we will give a representative extract, and then discuss the aspects of the mechanisation that proved challenging. The complete case study is available on the authors' web pages.

3.1 HERMIT Scripts

Our approach to mechanisation was to first state any auxiliary lemmas as (inactive) rewrite rules in the Haskell source file (as discussed in Section 2.1). To verify these lemmas, we first worked in HERMIT's interactive mode until the proof was successful, and then saved the final proof to a script that could be invoked thereafter. Finally, we developed the main transformation interactively, invoking these auxiliary proof scripts as necessary. For this case study the proofs were simply transliterations of the proofs in the textbook, but we expect developing new proofs would proceed in a similar manner, but with more experimentation and backtracking during the interactive phases.

As an example, we present the proof of Lemma 6.8, comparing the textbook proof with the HERMIT script. Figure 3a presents the proof extracted verbatim from the textbook [2, Page 36], and Figure 3b 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 performing the proof interactively. We manually added these comments to the HERMIT proof scripts to help readability and maintenance of the scripts.

When translating the textbook proof into HERMIT, we decided to split the middle step into two steps, as we felt that made the proof easier to read, but this is purely stylistic. Otherwise, the main difference between the two calculations is that in HERMIT we must specify where, and in which direction, to apply a lemma, whereas in the textbook the lemma is merely named, relying on the reader to be able to deduce how it was applied. Here, `one-td` (once, traversing top-down) and `any-td` (anywhere, traversing top-down) are *strategy combinators* from KURE [30], the strategic programming language that underlies HERMIT.

In this proof, and most others in the case study, we think that the HERMIT scripts are as clear, and not much more verbose, than the textbook calculations. There is one notable exception though, which involves manipulating terms containing adjacent occurrences of the function composition operator.

3.2 Associative Operators

On paper, associative binary operators such as function composition are typically written without parentheses. However, in HERMIT, a term is represented as 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.

One way to avoid this is to work with eta-expanded terms and unfold all occurrences of function composition, as this always produces an abstract syntax tree consisting of a left-nested sequence of

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

(a) Textbook extract.

```
-- unzip · map (fork (f, g))
   one-td (unfold 'unzip)
-- fork (map fst, map snd) · map (fork (f, g))
   forward (lemma "6.6")
-- fork (map fst · map (fork (f, g)), map snd · map (fork (f, g)))
   any-td (forward (lemma "map-fusion"))
-- fork (map (fst · fork (f, g)), map (snd · fork (f, g)))
   one-td (forward (lemma "6.5a"))
   one-td (forward (lemma "6.5b"))
-- fork (map f, map g)
```

(b) HERMIT script.

Figure 3: Comparison of HERMIT script with textbook calculations for Lemma 6.8 ($\mathit{fork}\ (\mathit{map}\ f, \mathit{map}\ g) \equiv \mathit{unzip} \circ \mathit{map}\ (\mathit{fork}\ (f, g))$).

applications. However, we did not do so for this case study, as the textbook proofs are written in a point-free style, and we wanted to match those proofs as closely as possible.

More generally, rewriting terms containing associative (and commutative) operators is a well-studied problem [e.g. 3, 8, 19], and it remains as future work to provide better support for manipulating such operators in HERMIT.

3.3 Proofs Omitted in the Textbook

During mechanisation we discovered that several auxiliary properties in the textbook (Lemmas 6.2, 6.3, 6.4, 6.5, 6.6, 6.7 and 6.10, and several minor unnamed properties) are stated as assumptions without proof. The lack of proofs is not commented on in the textbook, but we suspect that they are deemed either "obvious" or "uninteresting", as is common practice with pen-and-paper proofs. While performing reasoning beyond that presented in the textbook was not intended to be part of the case study, we decided to investigate how easy it is to prove these auxiliary properties.

In most cases, these properties had fairly straightforward inductive proofs, which were easy to encode in HERMIT. This mostly consisted of inlining definitions and then simplifying the resultant expressions as much as possible. Systematic proofs 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 proof scripts were short and concise.

Assumption 6.2 also had a simple proof, but it relied on arithmetic properties of Haskell's built-in Int type (specifically, that $m \equiv n \Rightarrow m \leqslant n$). HERMIT does not yet provide any support for reasoning about built-in types, so we were not able to encode this proof. This is a clear deficiency of HERMIT, and adding such support is important future work. We found that assumptions 6.3 and 6.4 were non-trivial properties, without (to us) obvious proofs.

Additionally, the simplification of the definition of expand is stated in the textbook without presenting the transformation steps [2, Page 40]. This simplification is non-trivial, and involves changing the type of an auxiliary function, and we did not find an easy way to encode this in HERMIT.

3.4 Proof Techniques Unsupported by HERMIT

Two proof techniques are used in the textbook that HERMIT does not directly support. The first is the *fold fusion* law [21]. Specialised to lists, fold fusion gives the following inference rule:

$$
\frac{f\,\bot \equiv \bot \qquad f\ a \equiv b \qquad \forall x, y\,.\, f\ (g\ x\ y) \equiv h\ x\ (f\ y)}{f \circ \mathit{foldr}\ g\ a \equiv \mathit{foldr}\ h\ b}
$$

This cannot be expressed as a HERMIT lemma, as HERMIT (currently) only supports equality lemmas, not implications. We therefore encoded *foldr-fusion* as a new primitive transformation using HERMIT's transformation DSL. We were able to reuse a substantial amount of existing HERMIT infrastructure to encode this rule, and so the encoding of the rule was only 20 lines of Haskell code. The plugin as a whole took another 30 lines of Haskell code, but that involved reusable auxiliary functions and plugin infrastructure that would be shared with any other user-added transformations. While this approach is only recommended for experienced HERMIT users, we think this is a viable approach for encoding custom transformations in HERMIT.

The second (unsupported) proof technique that the textbook uses is to postulate the existence of an auxiliary function (expand), use that function in the foldr-fusion rule, and then calculate a definition for that function starting from the foldr-fusion pre-conditions. This style of reasoning is not supported by HERMIT, nor is there an easy way to encode it. However, we were able to *verify* the calculation by working in reverse: starting from the definition in the textbook, we proceeded to prove the foldr-fusion pre-condition and thus validate the use of fold-fusion.

3.5 Calculation Sizes

As demonstrated by Figure 3, 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. We present some statistics in Table 1, but we don't recommend extrapolating anything from them beyond a rough approximation of the scale of the proofs. We give the size of the two main calculations (transforming $\mathit{solutions}$ and deriving expand), as well as the named auxiliary lemmas. In the textbook we measure lines of natural language reasoning as well 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 HERMIT 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 Lemma 6.8 in Table 1 to the calculation in Figure 3.

Calculation	Textbook Lines	HERMIT Commands		
		Transformation	Navigation	Total
solutions	16	12	7	19
expand	19	18	20	38
Lemma 6.5	not given	4	4	8
Lemma 6.6	not given	2	1	3
Lemma 6.7	not given	2	0	2
Lemma 6.8	7	5	8	13
Lemma 6.9	1	4	4	8
Lemma 6.10	not given	23	13	36
Total	43	70	57	127

Table 1: Comparison of calculation sizes.

3.6 Summary

Our overall experience was that mechanising the textbook calculations was fairly straightforward, and it was pleasing that we could translate most steps of the textbook reasoning into an equivalent HERMIT command. The only annoyance was the need to manually apply associativity occasionally (see Section 3.2), 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 more complicated proof scripts than in the textbook, we don't actually consider that to be more work. Rather, we view a pen-and-paper proof that doesn't specify the location as 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` (apply the lemma anywhere it matches) can be used to avoid specifying precisely which sub-term the lemma should be applied to.

Encoding the foldr-fusion rule (Section 3.4) was a non-trivial amount of work, but once encoded, it was a reusable transformation. Furthermore, in the future we plan to extend HERMIT's representation of lemmas with logical connectives. This would allow rules such as foldr-fusion to be represented as lemmas rather than as primitive transformations, which would greatly simplify encoding them in HERMIT.

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. However, we cannot claim this as detecting an error in a pen-and-paper proof, as this was caught by GHC's type checker, not by HERMIT.

4. Related Work

Equational reasoning is used both to prove 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, 7, 13, 15]. Prior to HERMIT there have been several tools for mechanical equational reasoning on Haskell programs, including the Programming Assistant for Transforming Haskell (PATH) [35], the Ulm Transformation System (Ultra) [17], and the Haskell Equational Reasoning Assistant (HERA) [14]. However, to our knowledge, none of these tools is currently being maintained. Furthermore, these tools all operate on Haskell source code (or some variant thereof), and do not attempt to support GHC-extended Haskell.

Another similar tool is the Haskell Refactorer (HaRe) [20, 34], 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 and proof. 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.

Other than equational reasoning, there have been two main approaches taken to verifying properties of Haskell programs: testing and automated theorem proving. The most prominent testing tool is QuickCheck [5], which automatically generates large quantities of test cases in an attempt to find a counterexample. Other testing tools include SmallCheck [27], which exhaustively generates test values of increasing size so that it can find minimal counter examples, and Lazy SmallCheck [24, 27], which also tests partial values. Jeuring et al. [18] 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 proofs of those laws. Of course testing does not constitute a proof, but it is lightweight and effective at finding counter-examples for false properties.

There are several tools that attempt to automatically prove properties of Haskell programs, by interfacing with an automated theorem prover and passing it (a translation of) the Haskell program and the desired properties. These include Liquid Haskell [36], Zeno [31] and the Haskell Inductive Prover (Hip) [26]. Properties in Liquid Haskell are *refinement types*, which the user may add as type annotations in the source file. Like HERMIT, Liquid Haskell and Zeno operate on GHC Core, whereas Hip translates Haskell source code directly into first-order logic. These tools can all support inductive proofs, but 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 [6] is a tool built on Hip that addresses this limitation by exhaustively generating conjectures (up to a fixed term size) about the involved functions. These conjectures are first passed to Hip to prove, and any successes are then made available when attempting the main proof. Thus the user need not state the exact properties to which induction needs to be applied.

5. Future Work and Conclusions

While HERMIT has been used to successfully prototype GHC optimisations [1, 11], it is still very much an experimental tool, and development is ongoing. The next step is to add different modes to HERMIT that will limit the commands that are available. This will include a "read-only" mode, a "safe" mode, and a "super-user" mode. The read-only mode will allow navigation and alpha-renaming, but prohibit any other modifications to the code. The safe mode will only allow transformations that are known to be semantics preserving, and thus any rewrite rules or unproven lemmas will be prohibited. The super-user mode will correspond to the current capabilities of HERMIT, with no imposed limitations. We also anticipate the need for additional modes, perhaps with more fine-grained notions of safety. For example, a transformation that could potentially transform a terminating program into a diverging program should not be available in the safe mode, but a converse transformation that may introduce termination might be acceptable.

Thus far, structural induction (Section 2.2) is HERMIT's only proof technique for reasoning directly about recursive definitions, and it is limited to induction hypotheses that are one constructor deep. Future work includes generalising this to n constructors deep, and adding support for *corecursive* proof techniques [13].

We have successfully encoded some high-level transformation techniques as primitive HERMIT transformations with preconditions. The foldr-fusion rule in Section 3.4 is one example of this, and the worker/wrapper transformation [15, 28] is another. In a prior publication [29] we described encoding worker/wrapper in HERMIT, but at the time HERMIT had no means of verifying the preconditions, so they were not mechanically enforced. Using HERMIT's new equational reasoning infrastructure described in this extended abstract, we have updated the worker/wrapper encoding such that it checks a proof that the preconditions hold before performing the transformation. All of the preconditions for the examples in that previous publication have now been verified by HERMIT, and the proofs are bundled with the HERMIT package [10]. However, encoding primitive transformations in HERMIT is a non-trivial task for the user, so our long-term goal is to build up a library of such high-level transformations, to complement HERMIT's existing library of low-level transformations.

HERMIT continues to prove useful for developing compile-time program transformation and reasoning capabilities that can be used on real Haskell programs. By mechanising proofs during compilation, HERMIT enforces the connection between the source, proof, and compiled program. GHC plugins developed using HERMIT can be deployed with Haskell's Cabal packaging system, meaning they integrate with a developer's normal work-flow. HERMIT development is on-going, and we seek to target ever-larger examples.

Acknowledgments

We would like the thank the Haskell Symposium reviewers for their useful comments on an earlier version of this work. This material is based upon work supported by the National Science Foundation under Grant No. 1117569.

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 *Lecture Notes in Computer Science*, pages 167–182. Springer, 2011.

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

[5] 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.

[6] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. HipSpec: Automating inductive proofs of program properties. In *Workshop on Automated Theory eXploration 2012*, volume 17 of *Proceedings in Computing*, pages 16–25. EasyChair, 2013.

[7] 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.

[8] 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.

[9] 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.

[10] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. `http://hackage.haskell.org/package/hermit`, 2014.

[11] 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.

[12] GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.8.2*, 2014. URL `http://www.haskell.org/ghc/docs/7.8.2/`.

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

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

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] J. S. Reich, M. Naylor, and C. Runciman. Advances in lazy smallcheck. In *24th International Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2013.

[25] J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

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

[27] 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.

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

[29] N. Sculthorpe, A. Farmer, and A. Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *24th International Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2013.

[30] 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.

[31] 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.

[32] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *3rd Workshop on*

Types in Language Design and Implementation, pages 53–66. ACM, 2007.

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

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

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

[36] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2013.