The HERMIT in the Tree

Mechanizing Program Transformations in the GHC Core Language

Neil Sculthorpe, Andrew Farmer, and Andy Gill

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

Abstract. This paper describes our experience using the HERMIT toolkit to apply well-known transformations to the internal core language of the Glasgow Haskell Compiler. HERMIT provides several mechanisms to support writing general-purpose transformations: a domain-specific language for strategic programming specialized to GHC's core language, a library of primitive rewrites, and a shell-style–based scripting language for interactive and batch usage.
There are many program transformation techniques that have been described in the literature but have not been mechanized and made available inside GHC — either because they are too specialized to include in a general-purpose compiler, or because the developers' interest is in theory rather than implementation. The mechanization process can often reveal pragmatic obstacles that are glossed over in pen-and-paper proofs; understanding and removing these obstacles is our concern. Using HERMIT, we implement eleven examples of three program transformations, report on our experience, and describe improvements made in the process.

Keywords: GHC, mechanization, transformation, worker/wrapper

1 Introduction

HERMIT (Haskell Equational Reasoning Model-to-Implementation Tunnel) [4] is a recently implemented plugin for the Glasgow Haskell Compiler (GHC) [5] that provides an interactive interface for applying transformations directly to GHC's internal intermediate language. This plugin is part of a larger HERMIT toolkit, a Haskell framework that is being developed with the aims of supporting equational reasoning and allowing custom optimizations to be applied without modifying either GHC or the Haskell users' source code.

There are a wide variety of transformation techniques for optimizing functional programs. Many such transformations have been implemented, and many are used by modern compilers. However, there are also techniques that have been described on paper but not mechanized, either because the transformation is too specialized to include as an optimization in a general-purpose compiler,

or because the developers' interest is in theory rather than implementation. We want to implement these more specialized transformations using the custom optimization capabilities of HERMIT.

We believe there is a lot to be learned from mechanizing program transformations. The mechanization process can often reveal obstacles that do not appear in pen-and-paper proofs, either because of implementation-specific details, or because the pen-and-paper proofs gloss over details that may seem obvious to a human, but are less obvious to a machine.

This paper reports on our experience using HERMIT to mechanize optimization techniques, using the worker/wrapper [7, 25], concatenate vanishes [29] and tupling [1, 9] transformations as case studies. We first introduce these transformations (§2), then we overview HERMIT and what it offers to the mechanization process (§3). We then give an extended example of using HERMIT to specifically apply tupling (§4), then discuss our general experience using HERMIT on our 11 examples (§5). Finally we discuss related work (§6), and draw conclusions from our mechanization efforts (§7).

Whereas the previous HERMIT publication [4] described HERMIT itself, this paper describes HERMIT in use, on a suite of examples. The main contribution of this work is pragmatic — showing by example that the HERMIT system is sufficiently mature to be able to encode and apply well-understood transformation techniques, in the context of the full power of GHC. We report on our experience, the obstacles that arose during mechanization, and our approaches to overcoming them, including a new combinator for tree traversal: `any-call`. Additionally, we demonstrate that it is straightforward to augment HERMIT with new specialized transformations as needed.

At this stage of our investigations we are explicitly concerned with mechanization rather than formal proof; for example, a number of the transformations we use have pre-conditions that HERMIT does not verify. We return to this shortcoming in §5.1, and for now observe that correctness and mechanization are both important, but independently challenging.

2 Transformations for Mechanization

This section overviews the program-transformation techniques that we chose as case studies. While mechanizing these techniques we observed that the concatenate vanishes transformation, and our main tupling transformation, are instances of the worker/wrapper transformation. A proof of the former, and an informal sketch of the latter, are given in the extended version of this paper, which is available on the first author's webpage.

2.1 Concatenate Vanishes

The concatenate vanishes transformation (CV) [29] is a technique for increasing the efficiency of programs that make repeated use of list concatenation. Consider the following standard definition:

$$
\begin{aligned}
(+\!\!+) &\quad :: [a] \to [a] \to [a] \\
[\,] &\quad +\!\!+ bs = bs \\
(a : as) &+\!\!+ bs = a : (as +\!\!+ bs)
\end{aligned}
$$

The time complexity of this definition is linear in the length of its first argument, but constant in the length of its second argument. Thus, while $+\!\!+$ is associative, $(as +\!\!+ bs) +\!\!+ cs$ will evaluate less efficiently than $as +\!\!+ (bs +\!\!+ cs)$. The essence of CV is to exploit this observation to restructure programs using repeated concatenation into a more efficient form.

CV can be summarized as follows. Given a function that returns a list[1],

$$
\begin{aligned}
f &\quad :: a \to [b] \\
f\ a &= expr
\end{aligned}
$$

where *expr* is an expression that may contain f and a, define a new function that returns a list-to-list function (known as a *difference list* [10]):

$$
\begin{aligned}
f' &\quad :: a \to [b] \to [b] \\
f'\ a\ bs &= expr +\!\!+ bs
\end{aligned}
$$

Then redefine the original function f as:

$$
\begin{aligned}
f &\quad :: a \to [b] \\
f\ a &= f'\ a\ [\,]
\end{aligned}
$$

The efficiency gains (if any are possible) are then achieved through refactoring the definition of f': first by applying the associativity and unit laws of $+\!\!+$, and then by folding [2] the definition of f' to eliminate any recursive calls to f.

2.2 Tupling Transformations

Tupling transformations come in several forms. The main one we consider in this paper involves transforming a recursive function that repeatedly solves subproblems into one that uses tabulation, a form of dynamic programming optimization where each subproblem is only solved once, and the solutions to subproblems are only stored as long as needed [16, 1, 15]. We will refer to this particular tupling transformation as TT.

As an example, consider the call tree for the Fibonacci function, in Fig. 1a. Computing *fib* n requires computing $fib(n-1)$ and $fib(n-2)$, but computing $fib(n-1)$ also requires computing $fib(n-2)$. We would like to avoid this duplication by exploiting sharing, such that our transformation results in the call *graph* in Fig. 1b.

In general, to perform TT on a function f, we define a function t whose body is an n-ary tuple of the n calls to f that share a common recursive call. By case-splitting on the arguments to t, we establish base cases for well-founded recursion. The recursive case of t is then calculated by selectively unfolding calls

[1] For clarity of presentation we assume the function is in uncurried form, but CV is valid for functions that take any number of arguments; see [29].

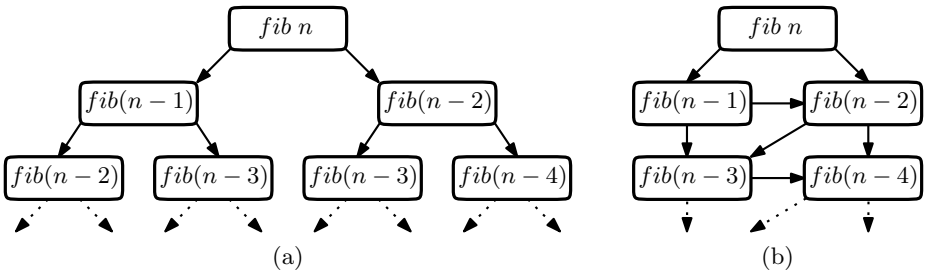

Fig. 1: Call graphs for *fib*, illustrating duplicated computation.

to f to expose the common recursive call. All distinct calls to f are let-bound, introducing sharing, which is the goal of the transformation. These let-bound calls are themselves grouped into an n-ary tuple, which can be folded into a call to t, leaving t recursively defined. Finally, f is redefined non-recursively in terms of t. This is demonstrated in detail for the Fibonacci function in §4.

2.3 Worker/Wrapper Transformation

The worker/wrapper transformation (WW) [7, 25] is a technique for improving the efficiency of a recursive program by changing the data type being operated on. The idea is to factorize a program $prog :: a$ into a more efficient *worker* program $work :: b$, and a *wrapper* function $wrap :: b \to a$ that converts the result into a value of the original type.

The first step is to ensure that the program is expressed as the least fixed point of a non-recursive function f, which may involve rewriting as follows (where *expr* is an expression that may contain *prog*):

$$prog = expr \qquad \Rightarrow \qquad prog = \mathbf{let}\ f = \lambda prog \to expr\ \mathbf{in}\ fix\ f$$

Next comes the key step: choosing a more efficient data type. Once chosen, we define conversion functions between the two types:

$$\begin{aligned} unwrap &:: a \to b \\ wrap &:: b \to a \end{aligned}$$

These conversion functions are required to satisfy the property that

$$fix\ (wrap \circ unwrap \circ f) \equiv fix\ f$$

and often satisfy the stronger property $wrap \circ unwrap \equiv id$. It is then valid to redefine the original program as follows (this is called *WW factorization*):

$$prog = wrap\ work$$

The definition of *work* can be derived in a number of ways [25]. Typically, we start from either $work = fix\ (unwrap \circ f \circ wrap)$ or $work = unwrap\ prog$, and then simplify the definition using any laws specific to the types a and b.

In practice, Haskell programs are typically defined using general recursion, rather than a fixed-point operator. Consequently, using the WW transformation often involves the following sequence of steps: introduce *fix*; perform WW factorization; eliminate *fix*. To factor out this repetition, we define an additional transformation that comprises the three steps, converting a generally recursive function into a non-recursive function that calls a recursive worker (we call this the *WW split*):

$$prog = expr \quad \Rightarrow \quad \begin{array}{l} prog = \textbf{let}\ f = \lambda prog \rightarrow expr \\ \quad\ \textbf{in let}\ work = unwrap\ (f\ (wrap\ work)) \\ \quad\ \ \textbf{in}\ wrap\ work \end{array}$$

3 HERMIT

This section briefly overviews the HERMIT toolkit; for more details consult [4].

3.1 GHC Core

GHC recently added support for custom compiler plugins that can be inserted amid GHC's optimization passes [5]. HERMIT uses this mechanism to provide a transformation system for GHC Core, GHC's internal intermediate language.

GHC Core is an implementation of System $\mathrm{F}^{\uparrow}_{C}$ [27, 30], which is System F [21] extended with let-bindings, constructors, type coercions and algebraic and polymorphic kinds. Fig. 2 shows HERMIT's representation of GHC Core, omitting a few constructors that aren't used in this paper.

3.2 User Interface

HERMIT provides several interfaces at different levels of abstraction. In this paper we will use just one of those interfaces: a read-eval-print loop (REPL).

The REPL allows navigation over a GHC Core abstract syntax tree (AST), displaying the current sub-tree via a choice of pretty printers. The REPL provides a statically typed monomorphic functional language with overloading. Most commands construct a *rewrite* from AST to AST, and the result of executing such a command is the newly transformed AST. Historic versions of the AST are maintained, and it is possible to step back and forth through the history of ASTs, or create branches to explore alternative transformation sequences. That is, HERMIT provides a version-control tree, where each node of the tree is an AST. When the user has finished applying transformations, she selects one of the ASTs for GHC to compile, and the rest are discarded. We give an extended example using the REPL in §4.

3.3 Extendability

HERMIT is designed to facilitate the addition of new transformations. There are three methods of doing this: writing a script to combine existing transformations,

```
data ModGuts  = ModGuts { _ :: [CoreBind], ... }
data CoreBind = NonRec Var CoreExpr | Rec [CoreDef]
data CoreDef  = Def Var CoreExpr
data CoreExpr = Var Var | Lit Literal | Type Type
              | App CoreExpr CoreExpr | Lam Var CoreExpr
              | Let CoreBind CoreExpr  | Case CoreExpr Var Type [CoreAlt]
type CoreAlt  = (AltCon, [Var], CoreExpr)
```

Fig. 2: GHC Core.

leveraging the GHC RULES mechanism [20], or adding an internal primitive. We used all three methods extensively while mechanizing our examples.

Scripting is the least powerful method, as it can only construct transformations by sequencing HERMIT-shell commands. However, it does allow transformations to be named and abstracted, as scripts can be called by other scripts.

RULES allow Haskell source files to be annotated with directed rewrite rules. HERMIT exposes any such rules as rewrite commands, allowing the user to selectively apply them as desired. This provides a lightweight mechanism for adding transformations that cannot be expressed in terms of existing commands, albeit limited to those that can be expressed by RULES.

Adding an internal primitive is the most powerful method, and our experience has been that typically new transformations can be constructed fairly easily out of the large suite of low-level congruence combinators and strategic traversals provided by HERMIT and its underlying strategic-programming library, KURE (see [4, 24]). The main drawback of this approach is that it requires additions to the HERMIT source code, and consequently recompilation of the package.

4 Example: Fibonacci Tupling

In this section we demonstrate the mechanization process in detail by performing TT on the Fibonacci function using the HERMIT REPL. Starting with the clear but inefficient (exponential time) definition over Peano naturals,

```
data Nat = Z | S Nat

fib             :: Nat → Nat
fib Z           = Z
fib (S Z)       = S Z
fib (S (S n))   = fib (S n) + fib n
```

we transform it into the following efficient (linear time) definition:

```
fib'    :: Nat → Nat
fib' n  = fst (work n)
          where work :: Nat → (Nat, Nat)
                work Z       = (Z, S Z)
                work (S m)   = let (x, y) = work m in (y, x + y)
```

As TT is an instance of WW, we will make use of our existing WW infrastructure. Following [26], we choose the more efficient data type to be a function that returns a tuple of consecutive Fibonacci numbers, and define *wrap* and *unwrap* as follows:

$$
\begin{aligned}
&\mathit{wrap} \quad :: (\mathit{Nat} \to (\mathit{Nat}, \mathit{Nat})) \to \mathit{Nat} \to \mathit{Nat} \\
&\mathit{wrap}\ h = \mathit{fst} \circ h \\
&\mathit{unwrap} :: (\mathit{Nat} \to \mathit{Nat}) \to \mathit{Nat} \to (\mathit{Nat}, \mathit{Nat}) \\
&\mathit{unwrap}\ h\ n = (h\ n, h\ (\mathsf{S}\ n))
\end{aligned}
$$

Trivially, the $\mathit{wrap} \circ \mathit{unwrap} \equiv \mathit{id}$ precondition holds.

Placing the definitions of *fib*, *wrap* and *unwrap* into a file Fib.hs, we load the file into HERMIT, give some initialization commands (see §5.1), and zoom to the definition of *fib* using the `consider` command:

```
hermit "Fib.hs"
```

```
hermit> set-pp-expr-type Show ; flatten-module ; consider 'fib
```

```
fib = λ ds → case ds of wild
               Z    → Z
               S ds → case ds of wild
                        Z   → S Z
                        S n → (+) (fib (S n)) (fib n)
```

We can now see the GHC Core that has been generated.

The next step is to apply the WW split. We have written a script for this transformation (see §5.1), which we load and apply with the `load` command:

```
hermit> load "WWSplitTactic.hss"
```

```
fib = let f = λ fib ds → case ds of wild
                           Z    → Z
                           S ds → case ds of wild
                                    Z   → S Z
                                    S n → (+) (fib (S n)) (fib n)
          rec work = unwrap (f (wrap work))
      in wrap work
```

As we will need this definition of *work* later, we save it under the name *origwork* using the `remember` command:

```
hermit> consider 'work ; remember origwork
```

```
work = unwrap (f (wrap work))
```

We now need to η-expand the body of *work* so that we can unfold *unwrap*:

```
hermit> 0 ; eta-expand 'n
hermit> any-call (unfold 'unwrap)
```

```
λ n → (,) Nat Nat (f (wrap work) n) (f (wrap work) (S n))
```

There are several things to note here. Numbers designate a child node to descend into, with 0 designating the right-hand-side of the definition in this case (the sole child, as variables and literals are not considered to be children). `any-call` is a

higher-order command that applies its argument everywhere it can succeed in the current sub-tree (we discuss this further in §5.4). Finally, the tuple constructor is polymorphic, and thus takes two type arguments (both *Nat* in this case).

Next we case-split on n to establish a base case for *work*:

```
hermit> 0 ; case-split-inline 'n
```

```
case n of n
  Z   → (,) Nat Nat (f (wrap work) Z) (f (wrap work) (S Z))
  S a → (,) Nat Nat (f (wrap work) (S a)) (f (wrap work) (S (S a)))
```

Now we selectively unfold f in three of the four places it is called[2]:

```
hermit> { 1 ; any-call (unfold 'f) }
hermit> { 2 ; 0 ; 1 ; any-call (unfold 'f) }
hermit> simplify
```

```
case n of n
  Z   → (,) Nat Nat Z (S Z)
  S a → (,) Nat Nat (f (wrap work) (S a))
                    ((+) (wrap work (S a)) (wrap work a))
```

We move into the second case alternative for the remainder of the derivation. In the second tuple component, we unfold the saved definition of *work*:

```
hermit> 2 ; 0 ; { 1 ; any-call (unfold origwork) }
```

```
(,) Nat Nat (f (wrap work) (S a))
            ((+) (wrap (unwrap (f (wrap work))) (S a))
                 (wrap (unwrap (f (wrap work))) a))
```

This creates an opportunity for fusing *wrap* and *unwrap* via the worker/wrapper precondition, which we encoded in the source file as a GHC RULES pragma:

$$\{\text{-\# RULES} \quad \texttt{"precondition"} \quad \forall\, x.\quad \mathit{wrap}\ (\mathit{unwrap}\ x) = x\ \text{\#-}\}$$

```
hermit> any-call (unfold-rule precondition)
```

```
(,) Nat Nat (f (wrap work) (S a))
            ((+) (f (wrap work) (S a)) (f (wrap work) a))
```

Now the duplicated computation of $f\ (\mathit{wrap}\ \mathit{work})\ (\mathsf{S}\ a)$ is evident. We name each *distinct* call to f by introducing let bindings, float the lets outside of the tuple, and then fold the duplicated computation of y:

```
hermit> { 1 ; 1 ; let-intro 'x }
hermit> { 0 ; 1 ; let-intro 'y }
hermit> innermost let-float
hermit> any-call (fold 'y)
```

```
let x = f (wrap work) a
    y = f (wrap work) (S a)
in (,) Nat Nat y ((+) y x)
```

[2] Curly braces denote scoping: within a scope it is impossible to navigate above the node at which the scope starts, and when the scope ends the cursor returns to the starting node.

These steps caused us to wish for better navigation capabilities for moving into case alternatives and tuples, as the use of numbers is unclear and brittle. We think that this will be especially problematic as examples grow in size.

We now combine x and y into a case-analyzed tuple,

```
hermit> let-tuple 'xy
case (,) Nat Nat (f (wrap work) a) (f (wrap work) (S a)) of xy
  (,) x y → (,) Nat Nat y ((+) y x)
```

thereby exposing the opportunity to fold *unwrap*:

```
hermit> any-call (fold 'unwrap)
case unwrap (f (wrap work)) a of xy
  (,) x y → (,) Nat Nat y ((+) y x)
```

All that remains is to fold our saved definition of *work*. This results in a definition with no calls to f, and no conversions via *wrap* and *unwrap*:

```
hermit> any-call (fold origwork)
case work a of xy
  (,) x y → (,) Nat Nat y ((+) y x)
```

Zooming out to see all of *fib*, we notice that f is now dead code. This would be removed by GHC's optimizer, but for presentation purposes we do so here. We also unfold the remaining call of *wrap*:

```
hermit> top ; consider 'fib
hermit> innermost dead-let-elimination
hermit> any-call (unfold 'wrap) ; simplify
fib = let rec work = λ n → case n of n
                             Z   → (,) Nat Nat Z (S Z)
                             S a → case work a of xy
                                     (,) x y → (,) Nat Nat y ((+) y x)
      in λ x → fst Nat Nat (work x)
```

We now have the efficient version of *fib*, and so tell GHC to resume compilation:

```
hermit> resume
```

5 User Experience

In this section we discuss our experience using HERMIT to mechanize our suite of transformations. After selecting the three transformation techniques, we chose the following representative examples from the literature as our suite, and mechanized them using HERMIT:

- WW: CPS [7], Last [26], Reverse [7, 4], Memoization [7], Unboxing [7, 18]
- CV: Flatten [29, 11], Quicksort [29], Reverse [29, 11]
- Tupling: Fibonacci [2, 1, 26], Mean [9], Towers of Hanoi [16, 3]

Our resulting scripts are bundled with the HERMIT package, and are summarized in Table 1. The Fibonacci script presented in §4 should provide the reader with a point of comparison.

Script Name	Number of HERMIT Commands[a]				Scripts Called
	Rewrites	Strategy Combinators	Navigation	Total	
WWSplit	12	0	8	20	–
CPS	13	4	10	27	WWSplit
Last	10	1	8	19	WWSplit
Reverse	21	16	7	44	WWSplit
Memoisation	6	2	6	14	WWSplit
Unboxing	15	7	10	32	WWSplit
ConcatVanishes	23	8	5	36	–
Flatten	1	0	2	3	ConcatVanishes
Quicksort	3	1	2	6	ConcatVanishes
Reverse	1	0	2	3	ConcatVanishes
Fibonacci	21	12	21	54	WWSplit
Hanoi	34	21	36	91	WWSplit
Mean	19	5	27	51	–

[a] Rewriting commands are those that modify the syntax tree, navigation commands focus the cursor onto specific nodes, and strategy combinators modify rewrites to apply them in some systematic manner.

Table 1: HERMIT script sizes.

5.1 Worker/Wrapper

WW was the first transformation that we mechanized. Introducing *fix*, the first step of WW, was not a transformation originally provided by HERMIT, nor was it definable in terms of other HERMIT commands. However, using the existing HERMIT infrastructure, it was straightforward to add a new rewrite for this task. Adding a rewrite to eliminate *fix* was unnecessary, as that can be achieved by using HERMIT's existing `unfold` command.

We chose to encode WW factorization using GHC RULES. Thus no modification to HERMIT was required, we just included the following pragma in the source code of each example, along with appropriate *wrap* and *unwrap* functions:

$$\{\text{-\# RULES} \quad \texttt{"ww"} \quad \forall f. \quad \mathit{fix}\ f = \mathit{wrap}\ (\mathit{fix}\ (\mathit{unwrap} \circ f \circ \mathit{wrap}))\ \text{\#-}\}$$

This use of GHC RULES works, but is clunky to use, being specific to each *wrap* and *unwrap*. We are currently working on creating a HERMIT command that takes *wrap* and *unwrap* functions as parameters, thereby avoiding the need to repeat this rule for every specific *wrap* and *unwrap*.

We encoded the WW split as a HERMIT script that calls WW factorization. That is, it assumes the existence of an appropriate `ww` rule in the source file. This is even more clunky, and we likewise intend to replace this script with a paramaterized HERMIT command.

HERMIT does not yet have a mechanism for checking preconditions, so it is up to the user to ensure that factorization is used only when the WW precondition holds. This is not ideal, and providing some mechanism within HERMIT for verifying pre-conditions, or at least for recording which pre-conditions have

been assumed during the transformation, is an obvious next step in its development. Furthermore, as well as the danger of the HERMIT user incorrectly using a rule, it is also possible that the GHC optimizer may apply a rule (which is the intended purpose of GHC RULES after all). We addressed this using GHC's phase annotations, which allow the user to specify which optimization phases the rule is eligible to be applied in. Inconveniently, these annotations required at least one phase to be specified, but patching the GHC parser to accept zero phases was trivial. This patch will be included in the GHC 7.8 release.

Another issue is that, unlike in a Haskell source file, the top-level bindings are not treated as a mutually recursive group. During type checking (before generating GHC Core), a dependency analysis separates the bindings into minimal recursive groups and orders these groups by their dependencies [17, §6.2.8]. This can be problematic when applying GHC RULES, as some of the variables in the rule may not be in scope. To address this, we added a `flatten-module` rewrite that combines the top-level binding groups into a single recursive group, thereby ensuring that all variables that can appear in a rule will be in scope.

Other than these issues, we found mechanizing the WW examples to be straightforward uses of HERMIT's basic transformations and GHC RULES. A detailed walk-through of the Reverse example, in the spirit of §4, can be found in our earlier description of HERMIT [4].

We also encountered some unexpected behavior involving type-level universal quantification. GHC Core passes around type arguments explicitly; thus when a call is made to a polymorphic function, the type argument has to be provided. For example, the Core generated from *last* has the following type and structure:

$$
\begin{aligned}
&last :: \forall\, \tau\,.\,[\tau] \to \tau \\
&last = \Lambda\, \tau \to \lambda\, as \to ...last\ \tau\ ...
\end{aligned}
$$

However, we discovered that if the type signature of a top-level polymorphic function is omitted in the source code, GHC generates different Core. Specifically, it performs the static-argument transformation [22], producing an outer non-recursive polymorphic function, and an inner recursive monomorphic function. That is, the type is fixed outside the recursion, avoiding the need to provide the type as an argument to each recursive call.

$$
\begin{aligned}
&last :: \forall\, \tau\,.\,[\tau] \to \tau \\
&last = \Lambda\, \tau \to \mathbf{let}\ last :: [\tau] \to \tau \\
&\qquad\qquad\qquad\quad last = \lambda\, as \to ...last\ ... \\
&\qquad\qquad\ \mathbf{in}\ last
\end{aligned}
$$

This difference, which is not noticeable at the level of Haskell source code, is significant enough to allow a GHC rule to fire in one case and not another. For example, WW factorization only fires for monomorphic functions, not polymorphic ones. In our opinion, HERMIT's ability to interactively display information on selected fragments of GHC Core was most helpful in understanding why the rule was not firing. Indeed, we believe that experimenting with and debugging GHC RULES is a potential application of the HERMIT system.

5.2 Concatenate Vanishes

Mechanizing CV proved straightforward. The main step can be expressed as WW factorization, so most of the transformation proceeded in the same manner as in §5.1. Mechanizing Flatten and Quicksort proved very similar to Reverse, with only a few differences in the basic rewrites required to simplify the resultant worker function. It was not necessary to add any new functionality to HERMIT.

Encouraged by the similarity of the three HERMIT scripts, we wrote a single generic script that works for all three examples, using HERMIT's higher-level commands. For this we did need to add a new command to HERMIT. The issue was that case-floating (taking a function applied to a case expression and applying it to each case alternative instead) is only valid if the function is strict:

$$
\begin{array}{lcl}
f\ (\textbf{case}\ x\ \textbf{of} & & \textbf{case}\ x\ \textbf{of} \\
\quad a_1 \to e_1 & & \quad a_1 \to f\ e_1 \\
\quad a_2 \to e_2 & \Rightarrow & \quad a_2 \to f\ e_2 \\
\quad \ldots & & \quad \ldots \\
\quad a_n \to e_n) & & \quad a_n \to f\ e_n
\end{array}
$$

As HERMIT lacks a mechanism for verifying preconditions (§5.1), it is the user's responsibility to ensure that case-floating is only applied to strict functions. This was fine when considering each example in isolation, as we explicitly stated when and where to float a case. But as this differed between examples, the usage in the generic script was potentially unsafe. To address this, we added a command that floats case (and let) expressions, but only past a specific function that it takes as a parameter. Again, adding this was straightforward.

Our generic CV script makes heavy use of GHC RULES, which encode the monoid laws for $((+\!\!+), [\,])$ and $((\circ), id)$, and a monoid homomorphism between them. We also used a rule to encode the fusion law relating the conversion functions between lists and difference lists [7]. This rule also has a precondition, and currently its usage in the generic script is unsafe in general (although in each specific example it is used safely). We are working on adding a rewrite to HERMIT that will allow us to restrict this rule to situations where the precondition is met, in a similar manner to the case-floating previously discussed.

Note that we do not claim that our generic script would work for any CV example; indeed we are quite confident it would not. Its purpose was just to test how well HERMIT copes with abstracting from multiple similar examples. HERMIT is designed as an interactive system where transformations are user-guided; we do not aim nor expect to be able to fully automate transformations in general. What we do aim for is to make HERMIT commands as robust as possible, in an effort to minimize the changes required if the source code changes, and more abstract commands help in this regard.

5.3 Tupling Transformations

The tupling examples motivated several new capabilities in HERMIT. Recall that in the Fibonacci example (§4) we established a base case for *work* by case-splitting on a variable. This functionality required us to create a new rewrite,

`case-split-inline`[3], that performs the following transformation (where $C_1..C_n$ are the constructor patterns of type T):

$$
\begin{array}{lcl}
 & & \textbf{case } x \textbf{ of} \\
 & & \quad C_1 \rightarrow expr\ [\, C_1 \,/\, x \,] \\
expr\ [\, x :: T \,] & \Rightarrow & \quad C_2 \rightarrow expr\ [\, C_2 \,/\, x \,] \\
 & & \quad ... \\
 & & \quad C_n \rightarrow expr\ [\, C_n \,/\, x \,]
\end{array}
$$

This rewrite was straightforward to implement using capabilities provided by the HERMIT API. It exposes an issue, however, when dealing with primitive types. For example, the only constructor for the *Int* type is $I\#$, which wraps a primitive unboxed integer, rendering case-splitting rather unproductive. One could imagine an alternative rewrite that accepts a literal value as the case to introduce, rather than enumerating the constructors. However, implementing this rewrite would require modifying the HERMIT REPL parser to parse Haskell values (or at the very least, Haskell literals), and so remains future work.

The tupling examples use the fold/unfold equational-reasoning technique [2]. When using fold/unfold, it is common to need access to *past* definitions of functions; a non-issue when working on paper (one simply looks up the page), but one that we needed to address. While the HERMIT kernel maintains a record of every version of the AST, we found it preferable to provide a command `remember` that explicitly saves a definition, rather than dig through the kernel's history. This also allows fold/unfold to be a lower-level notion that does not assume the existence of a version-control history, and means a definition can be saved and then applied within a single composite rewrite.

Our implementation of `fold` performs a straightforward structural comparison of two expressions, attempting to instantiate one in terms of the other, and thus is currently limited to folding syntactically α-equivalent expressions. This was the most challenging new rewrite to add because it traverses two ASTs in lockstep, and therefore cannot use much of the automation provided by KURE.

Exposing fold opportunities required a new rewrite `let-tuple` that combines the right-hand sides of multiple non-recursive let-bindings into a tuple, which is then scrutinized by a case statement to project out the original bindings:

$$
\begin{array}{lcl}
\textbf{let } v_1 = e_1 & & \\
\quad\ \ v_2 = e_2 & & \\
\quad\ \ ... & \Rightarrow & \textbf{case } (e_1, e_2, ..., e_n) \textbf{ of} \\
\quad\ \ v_n = e_n & & \quad (v_1, v_2, ..., v_n) \rightarrow expr \\
\textbf{in } expr & &
\end{array}
$$

The only complication in encoding this rewrite was locating GHC's tuple constructor, as the name $(,)$ is used at both the type and value level, and in GHC Core they share the same name space. There is also a more general need to improve name lookup, as currently the source code has to explicitly import constructors for them to be visible to HERMIT.

[3] There is also a `case-split` command, which does not inline x in the alternatives.

Terminology	Description	Example
To *inline*	To replace a value with its definition.	$(\lambda x \rightarrow (+)\ x\ 4)\ (g\ x)$
To *apply*	To inline in the context of (zero or more) arguments, and perform beta-reduction (to let-binding) on *all* the arguments.	**let** $x_n = g\ x$ **in** $(+)\ x_n\ 4$ (where n is unique)
To *unfold*	To apply, then *attempt* safe/cheap substitution on all the new let-bindings introduced by the application.	$(+)\ (g\ x)\ 4$

Table 2: Inlining terminology and usage examples.

We found the Towers of Hanoi example to be substantially similar to the Fibonacci example, and it did not require any new capabilities beyond those we had already added. The Mean example on the other hand did require a handful of new transformations. However, these were simple local rewrites (such as let-floating) that had been omitted from HERMIT's suite of local transformations, and were straightforward to encode using KURE.

5.4 Observations on Inlining

Initially, it was unclear how best to provide function inlining. We found that the `inline` rewrite was in practice often followed by the general-purpose clean-up command `bash` [4]. Among other things, `bash` performs beta-reduction and inlining repeatedly, and thus was serving as a crude way of unfolding a definition. However, in some cases this was undesirably reducing the content of the inlined function or its arguments. Consider the following bindings:

$f = \lambda x \rightarrow (+)\ x\ 4$
$e = f\ (g\ x)$

What should the result of inlining f in the right-hand-side of e be? After consideration, we settled on three distinct rewrites, summarized in Table 2.

Building on this decision, we found KURE's traversal combinators [4, 24] insufficient for our needs: specifically, it was difficult to include as many arguments as possible when unfolding curried functions, while at the same time ensuring termination of unfolding. This is not an issue with `inline`, only `apply` and `unfold`. To address this, we invented a traversal strategy to support `apply` and `unfold` called `any-call`, which visits nodes in an order that maximizes the number of arguments provided to an inlined function, as well as traversing any arguments before performing the apply/unfold. We have used `any-call` in our examples, as it is now our standard traversal combinator for working with apply and unfold.

When inlining case wildcard binders, there is a choice between using either the case scrutinee, or the pattern matched by the current case alternative. For example, consider the following situation:

case *expr* **of** *wild*
  pat $\rightarrow$ ... *wild* ...

If we inline *wild* on the right-hand-side of the case alternative, should we replace it with *expr* or *pat*? HERMIT initially did the former, but in practice we found that we usually wanted the latter. We thus modified the `inline` rewrite accordingly, and added a rewrite `inline-scrutinee` to provide the old behavior.

6 Related Work

There are several refactoring tools for Haskell programs, including the Haskell Refactorer (HaRe) [14], the Programming Assistant for Transforming Haskell (PATH) [28], the Ulm Transformation System (Ultra) [8], and the Haskell Equational Reasoning Assistant (HERA) [6]. The key distinction of HERMIT from these systems is that they operate on Haskell source code, or some variant thereof, whereas HERMIT operates on GHC Core, midway through the compilation process. The principal advantage of this approach is that GHC Core is a small language, having stripped away all of Haskell's syntactic sugar. This makes HERMIT simpler to use, implement and maintain, as there are far fewer cases to consider. Other advantages are that this automatically supports GHC language extensions, as GHC compiles them to GHC Core, and that inserting HERMIT inside the GHC optimization pipeline allows transformations to be intermixed with GHC's optimization passes. However, a disadvantage is that HERMIT cannot output Haskell source code.

More generally, there are a wide variety of refactoring tools for other languages. However, unlike HERMIT, most do not support higher-order commands and the scripting of composite refactorings [12]. One exception is Wrangler [13], a refactoring tool for Erlang, which has recently added such support [12].

One can also use proof assistants such as Coq or Agda to mechanize program transformations interactively. However, this requires modeling the syntax and semantics of the object language, encoding the program in that model, and then, after transformation, transliterating the result back into the object language before it can be compiled and executed. Even were we to ignore GHC language extensions, or consider only a limited subset of Haskell 98, the presence of partial values and lazy semantics mean we cannot simply define our programs directly in the total languages provided by such proof assistants, but instead have to model Haskell's domain-theoretic setting of continuous functions over pointed ω-complete partial orders [23]. We emphasize that one of the aims of the HERMIT project is to make transforming Haskell programs easy for the user: we do not want familiarity with domain theory and proof assistants to be prerequisites.

7 Conclusions and Future Work

Our experience thus far has been that it is viable to mechanize basic program transformations, and that performing the transformations in HERMIT is no more complicated than on paper. However, while encoding our examples we repeatedly found it necessary to add additional transformations, and higher-level

transformation strategies. This is unsurprising, as the HERMIT system is still in an early stage of development. What remains to be seen is whether, as we try more complex examples, we continue to need to add new transformations, or whether those we have now will scale. In general, we found adding new transformations to HERMIT to be a fairly simple procedure, whether by building them from HERMIT's existing low-level transformations, or by using GHC RULES. More challenging has been verifying the correctness of these transformations, and debugging our HERMIT programs when they fail to do as we expect.

Working within GHC has proved convenient. GHC Core has already been type checked before HERMIT acts on it, making all type information available. Much implementation effort was saved by using existing GHC functions such as substitution and variable de-shadowing, and safety checks such as the Core Lint pass [19], which ensures that the resultant code is type-correct and well-scoped.

More work is now needed. We have mechanized a collection of small examples as a proof of concept, but we need to try transforming larger real-world programs.

Acknowledgements

We thank Ed Komp for his work on implementing the HERMIT system, Jason Reich for suggesting the Mean example, and the anonymous reviewers for their constructive comments and feedback. This material is based upon work supported by the National Science Foundation under Grant No. 1117569.

References

1. Bird, R.S.: Tabulation techniques for recursive programs. ACM Computing Surveys 12(4), 403–417 (1980)
2. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM 24(1), 44–67 (1977)
3. Chin, W.N., Khoo, S.C., Jones, N.: Redundant call elimination via tupling. Fundamenta Informaticae 69(1–2), 1–37 (2006)
4. Farmer, A., Gill, A., Komp, E., Sculthorpe, N.: The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In: 2012 ACM SIGPLAN Haskell Symposium. pp. 1–12. ACM, New York (2012)
5. GHC Team: The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.6.2 (2013), http://www.haskell.org/ghc
6. Gill, A.: Introducing the Haskell equational reasoning assistant. In: 2006 ACM SIGPLAN Haskell Workshop. pp. 108–109. ACM, New York (2006)
7. Gill, A., Hutton, G.: The worker/wrapper transformation. Journal of Functional Programming 19(2), 227–251 (2009)
8. Guttmann, W., Partsch, H., Schulte, W., Vullinghs, T.: Tool support for the interactive derivation of formally correct functional programs. Journal of Universal Computer Science 9(2), 173–188 (2003)
9. Hu, Z., Iwasaki, H., Takeichi, M., Takano, A.: Tupling calculation eliminates multiple data traversals. In: 2nd ACM SIGPLAN International Conference on Functional Programming. pp. 164–175. ACM, New York (1997)

10. Hughes, R.J.M.: A novel representation of lists and its application to the function "reverse". Information Processing Letters 22(3), 141–144 (1986)
11. Hutton, G.: Programming in Haskell. Cambridge University Press (2007)
12. Li, H., Thompson, S.: A domain-specific language for scripting refactoring in Erlang. In: 15th International Conference on Fundamental Approaches to Software Engineering. pp. 501–515. Springer, Berlin (2012)
13. Li, H., Thompson, S., Orosz, G., Tóth, M.: Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In: 7th ACM SIGPLAN Erlang Workshop. pp. 61–72. ACM, New York (2008)
14. Li, H., Thompson, S., Reinke, C.: The Haskell refactorer, HaRe, and its API. Electronic Notes in Theoretical Computer Science 141(4), 29–34 (2005)
15. Liu, Y.A., Stoller, S.D.: Dynamic programming via static incrementalization. Higher-Order and Symbolic Computation 16(1–2), 37–62 (2003)
16. Pettorossi, A.: A powerful strategy for deriving efficient programs by transformation. In: 1984 ACM Symposium on LISP and Functional Programming. pp. 273–281. ACM, New York (1984)
17. Peyton Jones, S.: The Implementation of Functional Programming Languages. Prentice Hall (1987)
18. Peyton Jones, S., Launchbury, J.: Unboxed values as first class citizens in a non-strict functional language. In: 5th ACM Conference on Functional Programming Languages and Computer Architecture. pp. 636–666. Springer, London (1991)
19. Peyton Jones, S., Santos, A.L.M.: A transformation-based optimiser for Haskell. Science of Computer Programming 32(1–3), 3–47 (1998)
20. Peyton Jones, S., Tolmach, A., Hoare, T.: Playing by the rules: rewriting as a practical optimisation technique in GHC. In: 2001 ACM SIGPLAN Haskell Workshop. pp. 203–233. ACM, New York (2001)
21. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
22. Santos, A.: Compilation by Transformation in Non-Strict Functional Languages. Ph.D. thesis, University of Glasgow (1995)
23. Schmidt, D.A.: Denotational Semantics: A Methodology for Language Development. Allyn and Bacon (1986)
24. Sculthorpe, N., Frisby, N., Gill, A.: The Kansas University Rewrite Engine: A Haskell-embedded strategic programming language with custom closed universes, (in preparation)
25. Sculthorpe, N., Hutton, G.: Work it, wrap it, fix it, fold it. Journal of Functional Programming 24(1), 113–127 (2014)
26. Sculthorpe, N., Hutton, G.: Work it, wrap it, fix it, fold it (extended version) (2014), http://dx.doi.org/10.1017/S0956796814000045, extended version of [25]
27. Sulzmann, M., Chakravarty, M.M.T., Peyton Jones, S., Donnelly, K.: System F with type equality coercions. In: 3rd ACM SIGPLAN Workshop on Types in Language Design and Implementation. pp. 53–66. ACM, New York (2007)
28. Tullsen, M.: PATH, A Program Transformation System for Haskell. Ph.D. thesis, Yale University (2002)
29. Wadler, P.: The concatenate vanishes. Tech. rep., University of Glasgow (1989)
30. Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Giving Haskell a promotion. In: 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation. pp. 53–66. ACM, New York (2012)