The HERMIT in the Machine

A Plugin for the Interactive Transformation of GHC Core Language Programs

Andrew Farmer Andy Gill Ed Komp Neil Sculthorpe

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

Abstract

The importance of reasoning about and refactoring programs is a central tenet of functional programming. Yet our compilers and development toolchains only provide rudimentary support for these tasks. This paper introduces a programmatic and compiler-centric interface that facilitates refactoring and equational reasoning. To develop our ideas, we have implemented HERMIT, a toolkit enabling informal but systematic transformation of Haskell programs from *inside* the Glasgow Haskell Compiler's optimization pipeline. With HERMIT, users can experiment with optimizations and equational reasoning, while the tedious heavy lifting of performing the actual transformations is done for them.

HERMIT provides a transformation API that can be used to build higher-level rewrite tools. One use-case is prototyping new optimizations as clients of this API before being committed to the GHC toolchain. We describe a HERMIT application—a read-eval-print shell for performing transformations using HERMIT. We also demonstrate using this shell to prototype an optimization on a specific example, and report our initial experiences and remaining challenges.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

General Terms Experimentation, Languages, Performance, Verification.

Keywords DSLs, Equational Reasoning, GHC, Optimization, Strategic Programming

1. Introduction

We want to do equational reasoning on real Haskell programs. There are many tools for formalizing symbolic mathematics (Harrison 2009; Paulson 1989; Bertot and Castéran 2004), but currently, paper and pencil, or even text editors and LaTeX, are the state of the art when performing equational reasoning on real (GHC-extended) Haskell in the Haskell community. Towards being able to mechanize such reasoning, we are developing the Haskell Equational Reasoning Model-to-Implementation Tunnel (HERMIT), a toolkit for transforming GHC Core programs.

We aim to go further, and write *tools* that do equational reasoning on real Haskell programs. HERMIT is therefore designed as a framework that provides highly specific transformations as a service, as well as the general scripting capabilities of rewriting strategies. Our motivation is the exploration and possible automation of high-level program transformations, such as the worker/wrapper transformation (Gill and Hutton 2009).

1.1 A Taste of HERMIT

Imagine you are sitting at your terminal wishing your Haskell program would go faster. The optimization flag has been turned on, and you know of an unimplemented transformation that could help. What do you do? You could add a new optimization pass to GHC, taking part in the dark art of tuning heuristics to allow it to play well with others. Or you could experiment, using HERMIT.

As a first example we use the Fibonacci function, not because it is interesting, but because it is so well known.

$$\textbf{module } \mathit{Main} \textbf{ where}$$

$$\mathit{fib} :: \mathit{Int} \to \mathit{Int}$$
$$\mathit{fib}\ n = \textbf{if } n < 2 \textbf{ then } 1 \textbf{ else } \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2)$$

Compiling with `-O2`, and using Criterion (O'Sullivan) to average over 100 tests, we observe that *fib* 35 runs in 124.0ms ± 2.6ms on our development laptop.

To enable further optimization of *fib*, we want to try unrolling the recursive calls. We want to do this without changing the source, which is clear and concise. To do so, we fire up HERMIT, choosing to use the command-line interface. HERMIT uses the GHC Plugins mechanism (GHC Team 2012) to insert itself into the optimization pipeline as a rather non-traditional compiler pass, capturing programs mid-compilation and allowing the user to manipulate them.

```
ghc -fplugin=HERMIT -fplugin-opt=HERMIT:main:Main Main.hs
[1 of 1] Compiling Main ( Main.hs, Main.o )
[...]
module main:Main where
  fib :: Int -> Int
[...]
hermit> _
```

GHC has compiled our program into its intermediate form, called *GHC Core*, and HERMIT is asking for input. At this point we can start exploring our captured program.

```
hermit> consider 'fib
rec fib = λ n →
      case (<) ▲ $fOrdInt n (I# 2) of wild
        False →
          (+) ▲ $fNumInt
            (fib ((-) ▲ $fNumInt n (I# 1)))
            (fib ((-) ▲ $fNumInt n (I# 2)))
        True → I# 1
```

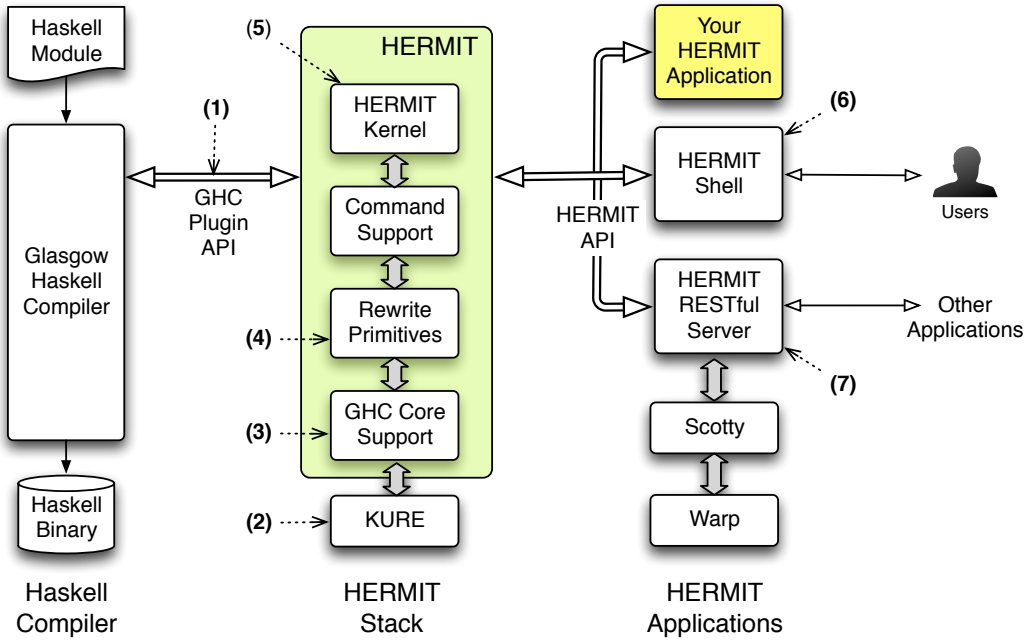

Figure 1. The HERMIT architecture.

The `consider` command moves the focus onto the specified function, and displays the function's abstract syntax. This is HERMIT's "compact" pretty printer for GHC Core. Core programs necessarily contain a substantial amount of information, and in this render engine HERMIT uses ▲ as a placeholder for type arguments. Notice that we refer to a Haskell syntactical name using the Template Haskell convention of prefixing with a tick.

Next we want to unroll `fib`, to see if this improves performance. Specifically, we want to inline all calls to `fib` (but nothing else) in a bottom-up manner. HERMIT uses strategic programming as its basis, so we build our command using two HERMIT strategies.

```
hermit> any-bu (inline 'fib)
rec fib = λ n →
      case (<) ▲ $fOrdInt n (I# 2) of wild
        False →
          (+) ▲ $fNumInt
            ((λ n →
                case (<) ▲ $fOrdInt n (I# 2) of wild
                  False →
                    (+) ▲ $fNumInt
                      (fib ((-) ▲ $fNumInt n (I# 1)))
                      (fib ((-) ▲ $fNumInt n (I# 2)))
                  True → I# 1)
               ((-) ▲ $fNumInt n (I# 1)))
            ((λ n →
                case (<) ▲ $fOrdInt n (I# 2) of wild
                  False →
                    (+) ▲ $fNumInt
                      (fib ((-) ▲ $fNumInt n (I# 1)))
                      (fib ((-) ▲ $fNumInt n (I# 2)))
                  True → I# 1)
               ((-) ▲ $fNumInt n (I# 2)))
        True → I# 1
```

Each inner instance of `fib` has been replaced with its definition, effectively unrolling `fib`. There are many more simplifications that can be done, such as β-reduction, but GHC is good at this. For now, we just want to test this version; so we type `resume` and GHC resumes compiling our program.

Now we time this new version. Again, taking the average over 100 tests using Criterion, we observe that the post-HERMIT *fib* 35 runs in about 76.7ms ± 3.6ms. We have modified GHC's internal representation of our program mid-compilation, improving it such that the resulting program runs measurably faster. Furthermore, we did not need to alter the original program to do so.

1.2 HERMIT Architecture and Contributions

HERMIT is a Haskell framework that provides tool support for transforming GHC Core programs. Using HERMIT, we intend to build tools for interactively performing equational reasoning, scripting equational reasoning, and, in general, constructing program-level equivalences via transformation. Figure 1 gives a pictorial representation of the principal HERMIT components. Referring to Figure 1:

(1) HERMIT is connected to GHC via the recently implemented GHC Plugins architecture. (§2)

(2) Our underlying rewrite engine is called KURE. We have considerably reworked this library since Gill (2009), with much of the development being driven by the needs of HERMIT. (§3)

(3) Using KURE, we have defined a complete set of congruence combinators specifically for the GHC Core syntax. (§3.2.4)

(4) There are numerous primitive transformations over Core, using KURE, or tunneling into GHC's existing rewrite capabilities. These transformations form the bulk of the HERMIT API. (§4)

(5) To manage rewrites, we have a HERMIT kernel, which operates as an actor on the HERMIT users' behalf, applying KURE rewrites and transformations, and recording everything in a virtual abstract syntax tree store. (§5.1)

(6) HERMIT has a read-eval-print loop (REPL) shell. (§5.2, 6)

(7) HERMIT also has a web-server interface. (§5)

This paper makes the following contributions:

- An improved design and implementation of the KURE strategic-programming library. New features include the use of lenses for navigation, a more reliable identity-detection mechanism based on a new family of traversal combinators, and better integration with existing Haskell infrastructure by making transformations instances of well-known type classes. (§3)
- The first large case study of the GHC Plugins capability. HERMIT is designed from the ground up to be a server architecture that provides access to the GHC internals. (§4)
- A case study of using the HERMIT shell to implement the worker/wrapper reverse example (Gill and Hutton 2009). (§6)

2. HERMIT: A GHC Plugin

The Glasgow Haskell Compiler (GHC) recently added support for plug-and-play optimization passes (GHC Team 2012). HERMIT uses this mechanism to provide a highly customizable transformation system for GHC's internal intermediate language, *GHC Core*. GHC Core is an implementation of System $\mathrm{F}_C^{\uparrow}$ (Sulzmann et al. 2007; Weirich et al. 2011a; Yorgey et al. 2012; Vytiniotis et al. 2012), which is the well-understood System F (Pierce 2002) extended with let-binding, constructors, and support for type coercions via proofs of type equality. In System $\mathrm{F}_C^{\uparrow}$, types are explicitly passed as arguments, but never returned.

Figure 2 gives the Haskell syntax for GHC Core, presented using some inlined type synonyms for clarity. Notice that the key type, $CoreExpr$, is quite small (only 10 constructors). A module, captured inside $ModGuts$, is (among other things) an ordered list of bindings. Bindings come in two types: single non-recursive bindings, and groups of mutually recursive bindings.

GHC accepts a plugin as a $ModGuts \rightarrow CoreM\ ModGuts$ function, and HERMIT provides such a function. Choosing to have HERMIT transform GHC Core, rather than some other representation, was a significant design decision. Other similar systems have made different choices; for example, HaRe (Brown 2008; Li and Thompson 2008) operates directly on Haskell syntax.

There are two major advantages to HERMIT's approach. First, GHC Core is a small language, having stripped away all of Haskell's syntactic sugar. This makes it much easier to work with, as there are fewer cases to consider. Second, HERMIT inserts itself inside the GHC optimization pipeline, allowing transformations to be intermixed with GHC's optimization passes, as desired. However, the primary disadvantage is that we cannot output Haskell source code.

Using ideas from strategic programming, we have built a set of abstractions for providing functions that operate over GHC Core. This is the subject of the next section.

3. Strategic Rewriting

When modifying a data structure, it can be easy to express the change you want to make, but tedious to route that change through the structure to the point at which you want to make it. Languages such as Haskell support polymorphic *combinators* (higher-order functions) to help with this. For example, map automates the traversal of a list, saving the programmer from having to explicitly encode a recursive traversal for every function she wishes to apply to every element in a list. However, while Haskell provides a rich set of traversal combinators for standard data types, user-defined data types lack this support. A programmer can define a set of traversals for each data type, but this is time consuming. Ideally, we want a single set of generic traversals that can be used on *any* user-defined data type.

$$
\begin{aligned}
&\textbf{data}\ ModGuts = \mathsf{ModGuts}\ \{\_ :: [CoreBind], \ldots\}\\
&\textbf{data}\ CoreBind = \mathsf{NonRec}\ Id\ CoreExpr\\
&\qquad\qquad \mid \mathsf{Rec}\ [(Id, CoreExpr)]\\
&\textbf{data}\ CoreExpr = \mathsf{Var}\ Id\\
&\qquad\qquad \mid \mathsf{Lit}\ Literal\\
&\qquad\qquad \mid \mathsf{App}\ CoreExpr\ CoreExpr\\
&\qquad\qquad \mid \mathsf{Lam}\ Id\ CoreExpr\\
&\qquad\qquad \mid \mathsf{Let}\ CoreBind\ CoreExpr\\
&\qquad\qquad \mid \mathsf{Case}\ CoreExpr\ Id\ Type\ [CoreAlt]\\
&\qquad\qquad \mid \mathsf{Cast}\ CoreExpr\ Coercion\\
&\qquad\qquad \mid \mathsf{Tick}\ (Tickish\ Id)\ CoreExpr\\
&\qquad\qquad \mid \mathsf{Type}\ Type\\
&\qquad\qquad \mid \mathsf{Coercion}\ Coercion\\
&\textbf{type}\ CoreAlt = (AltCon, [Id], CoreExpr)\\
&\textbf{data}\ AltCon = \mathsf{DataAlt}\ DataCon\\
&\qquad\qquad \mid \mathsf{LitAlt}\ Literal\\
&\qquad\qquad \mid \mathsf{DEFAULT}
\end{aligned}
$$

Figure 2. GHC Core.

This is a well-studied problem, and a variety of approaches have been proposed (Hinze 2000; Lämmel and Peyton Jones 2003, 2004; Löh 2004; Visser 2005; Rodriguez et al. 2008; Yakushev et al. 2009; Magalhães et al. 2010). This section describes one such approach to the problem: the Kansas University Rewrite Engine (KURE), a Haskell-embedded domain-specific language for strategic rewriting. The novel feature of KURE is the use of associated type synonyms (Kiselyov et al. 2010) to allow strongly typed traversals over data structures containing nodes of different types.

An earlier version of KURE was presented by Gill (2009). Here we summarize the KURE basics, and describe the new features of our updated version.

3.1 Strategic Programming and Stratego

KURE is based on *strategic programming* (Visser 2005), a paradigm for expressing transformations and traversals of tree-structured data. Two key concepts are *transformation rules*, which are local rewrites on a node in the tree, and *transformation strategies*, which combine rules to make new rules (Visser 2004). Transformation rules can either succeed or fail when applied to a node. Typically, a strategic programming language provides a selection of rules and strategies as primitives, a means of defining new primitive rewrites, and a means of defining new strategies and rules in terms of existing strategies and rules.

As an example, we consider Stratego (Bravenboer et al. 2008), the most widely used strategic rewrite system. Some representative rules and strategies are given by the following grammar:

$$S ::= id \mid fail \mid S\ ;\ S \mid S <+\ S \mid all\ (S)$$

Briefly, the semantics of these primitives are:

- id is the identity transformation that always succeeds;
- $fail$ is the always failing transformation;
- $s_1\ ;\ s_2$ is sequencing of transformations (requiring both to succeed);
- $s_1 <+\ s_2$ is a "catch" (apply s_1, and if it fails then apply s_2 instead);
- $all\ (s)$ applies s to all immediate child nodes (requiring all to succeed).

all is the most interesting, as it exploits the tree-structure of the data being transformed.

Using these combinators, new strategies can be defined. A classic example is *try*, which catches a failed transformation with the identity transformation (and thus always succeeds):

```
try (s) = s <+ id
```

As another example, the following strategies perform pre-order and post-order traversals (respectively), applying s to every node:

```
topdown (s)  = s ; all (topdown (s))
bottomup (s) = all (bottomup (s)) ; s
```

3.2 The KURE Transformation Library

KURE provides the transformations of Stratego as strongly typed Haskell combinators (Stratego is untyped). Much of the design of KURE is also inspired by StrategyLib (Lämmel and Visser 2002), another Haskell library for strategic programming. We overview the differences between the two systems in §3.3.

The design of KURE emerged from the requirements that there should be:

- a common traversal mechanism for both applying rewrites to the tree and extracting information from the tree;
- support for maintaining a context while traversing the tree;
- support for working in an arbitrary monad.

A common traversal mechanism is useful as the same combinators can be used for both tasks, cutting down the number of combinators that KURE needs to provide, and that a user has to learn.

Maintaining a context allows transformations to depend on information from elsewhere in the tree. In HERMIT, the context is the set of bindings in scope (and a few other things that won't be used in this paper):

```
data Context = Context (Map Id HermitBinding) ...
```

Having access to this context allows some transformations (inlining, for example) to be performed locally that would otherwise need to traverse the tree to locate this information.

Having KURE support an arbitrary monad gives transformations access to any required external operations, such as fresh name generation. For example, HERMIT uses the *CoreM* monad provided by GHC Plugins, combined with an error monad:

```
newtype HermitM a = HermitM (CoreM (Either String a))
```

These requirements lead to the following data type, which forms the basis of KURE:

```
data Translate c m a b = Translate { apply :: c -> a -> m b }
```

The four type parameters are: the context, the underlying monad, the node to be transformed, and the result. Allowing the result type to differ from the node type allows a transformation to project arbitrary information from a node. Rewriting a node is just the special case when the result is a node of the same type. Indeed, KURE provides a type synonym specifically for that case:

```
type Rewrite c m a = Translate c m a a
```

Typically, a KURE user will define synonyms for *Translate* and *Rewrite* specialized to their context and monad. For example, HERMIT defines the following synonyms:

```
type TranslateH a b = Translate Context HermitM a b
type RewriteH a = TranslateH a a
```

The KURE library exposes two primitive functions for building primitive transformations, with *rewrite* just being a synonym for *translate* at the more specialized type:

```
translate :: (c -> a -> m b) -> Translate c m a b
translate = Translate

rewrite :: (c -> a -> m a) -> Rewrite c m a
rewrite = translate
```

These functions are KURE's method of allowing the user to define custom transformation rules. For example, an inlining rewrite for GHC Core could be written like so:

```
inline :: RewriteH CoreExpr
inline = rewrite $ \c e -> case e of
  Var v -> case inlinable v c of
             Right e'  -> return e'
             Left msg -> fail msg
  _     -> fail "inline failed: not a variable"
```

We omit the details of the *inlinable* function, which simply looks up a variable in a context to determine if it is defined and is safe to inline at this location (without name clashes).

3.2.1 Transformation Combinators

In KURE, transformation strategies are just combinators operating over the *Translate* data type. KURE's approach to providing strategies is to make *Translate* an instance of well-known structures such as *Monad* and *Arrow*, and then any monadic or arrow combinator gives rise to a strategy. To be able to catch failures at the *Translate* level, the underlying monad is required to support a catch operation, which is expressed by the following type class:

```
class Monad m => MonadCatch m where
  (<+) :: m a -> m a -> m a
```

KURE then derives *Functor*, *Applicative*, *Monad*, *MonadCatch*, *Category*, *Arrow* and *ArrowApply* instances for the *Translate* type parametrized on that monad.

We omit the definitions as they are standard: the *Monad* family of instances equates to applying the Reader monad transformer to the underlying monad, and the Arrow family of instances equate to applying the Reader arrow transformer to the Kleisli arrow induced by the underlying monad. KURE also derives a *Monoid* instance for *Translate* whenever the result type forms a *Monoid*.

Amongst the many combinators these instances provide, the following correspond to the Stratego primitives (where *id* is from the *Category* type class):

```
id    :: Rewrite c m a
fail  :: String -> Translate c m a b
(>>>) :: Translate c m a b -> Translate c m b d ->
         Translate c m a d
(<+ ) :: Translate c m a b -> Translate c m a b ->
         Translate c m a b
```

For clarity, all combinators in this section are presented with their types specialized to *Translate*, but in the KURE library they are generalized to arbitrary monads and arrows. We also omit the *MonadCatch* class constraint as it is common to all of the combinators described.

Defining new combinators is as straightforward as in Stratego, for example:

```
tryR :: Rewrite c m a -> Rewrite c m a
tryR r = r <+ id
```

Experience with HERMIT has shown that when defining new transformations it is sometimes more convenient to work at the level of the *Translate* monad, and sometimes more convenient to work at the level of the underlying monad. By generalizing KURE's library of combinators to work on any monad, they are available at both levels (as opposed to needing two sets of combinators). Additionally, having access to **do** notation at the *Translate* level has proved useful; in particular, the invocation of *fail* on a pattern match failure enables a concise coding style. For example, a rewrite that floats a local let-binding to the top level,

```
v = (let w = ew in ev)          =>   w = ew
...                                  v = ev
                                     ...
```

can be expressed succinctly as follows:

```
letFloatLetTop :: RewriteH CoreProgram
letFloatLetTop =
  do NonRec v (Let (NonRec w ew) ev) : bds ← id
     return (NonRec w ew : NonRec v ev : bds)
```

The use of existing Haskell structures is one of the key improvements over the previous version of KURE, which did not make *Translate* an instance of these classes. One reason for this was that the previous KURE had an identity-detection mechanism whereby identity rewrites could be caught using various combinators (Gill 2009). This is useful when building larger rewrites, as we often want to detect not only whether a rewrite was applicable, but whether it actually changed anything. However, the mechanism violated the arrow laws, and thus prevented an arrow instance. It was also rather fragile, relying on the programmer to correctly use a combinator called *transparently* to mark identity-preserving uses of *rewrite* and *translate*.

The new version of KURE opts for a simpler approach. It supports an (optional) convention whereby rewrites that do not modify the term should fail (although there are exceptions, such as *id* and *tryR*, for when the user does not wish to follow this convention). It is then possible to define recursive rewrites that continue until a fixed point is reached. For example, the *repeatR* strategy repeatedly applies a rewrite until it fails, returning the result before the failure:

```
repeatR :: Rewrite c m a → Rewrite c m a
repeatR r = r ⋙ tryR (repeatR r)
```

In keeping with the convention, it requires at least one rewrite to succeed, else it fails overall.

To support this convention further, KURE provides a variant sequencing operator that succeeds if *either* rewrite succeeds, with each rewrite defaulting to an identity rewrite in the case of failure:

```
(>+>) :: Rewrite c m a → Rewrite c m a → Rewrite c m a
```

This differs from $tryR\ r_1 \ggg tryR\ r_2$, which will always succeed.

As examples of the ease of constructing new strategies in this setting, we define two combinators that sequence a list of rewrites:

```
orR :: [Rewrite c m a] → Rewrite c m a
orR = foldr (>+>) (fail "orR failed")

andR :: [Rewrite c m a] → Rewrite c m a
andR = foldr (⋙) id
```

They differ in that *orR* succeeds if at least one rewrite succeeds, whereas *andR* succeeds only if all the rewrites succeed.

3.2.2 Lenses

Another feature new in this version of KURE is the use of *lenses* (Foster et al. 2007). In our (specialized) setting, a lens is a tool for directing a transformation to a specific descendant node. For example, given a Lens from a node of type a to a descendant node of type b, KURE provides combinators that allow us to apply a *Rewrite* or *Translate* to that descendant:

```
focusR :: Lens c m a b → Rewrite c m b → Rewrite c m a
focusT :: Lens c m a b → Translate c m b r
       → Translate c m a r
```

Note that *focusR* is *not* a synonym for *focusT*, as the two have different semantics: *focusR* rewrites the descendant node, maintaining the structure of the rest of the tree; whereas *focusT* transforms the descendant node into a result value, discarding the rest of the tree.

We implement *Lens* as a subtype of *Translate*:

```
type Lens c m a b = Translate c m a ((c, b), (b → m a))
```

That is, as a translation from a term of type a to a triple: a term of type b, the context for that term, and a monadic function that converts a term of type b back to a term of type a. The *focus* combinators can now be defined as follows:

```
focusR l r = rewrite $ λc a → do ((c', b), k) ← apply l c a
                                 apply r c' b >>= k
focusT l t = translate $ λc a → do ((c', b), _) ← apply l c a
                                   apply t c' b
```

3.2.3 Tree Traversals

Notably absent from the discussion of KURE thus far have been traversal combinators such as Stratego's *all* strategy. This is because defining such combinators in a strongly typed setting is non-trivial. Consider, the argument rewrite to *all* should operate on all children of a node, which may have distinct types from each other and the parent, and the result should be a rewrite that operates on the parent. So what type should it have? A type:

```
allR :: Rewrite c m a → Rewrite c m a
```

would only work on children of the *same type* as the parent. Whereas a type:

```
allR :: Rewrite c m a → Rewrite c m b
```

does not relate a and b at all, and so *allR* cannot use the argument rewrite in any meaningful way. (Except in a language that provides runtime type comparisons, which is the approach taken by Dolstra (2001).) A detailed discussion of this problem and the development of a solution can be found in Gill (2009). Here, we just summarize the solution.

For each node that the KURE user wishes to traverse (henceforth referred to as *traversable nodes*), she is required to define a so-called "generic" data type that is the sum type of all traversable descendant nodes. Often, as is the case of HERMIT, a single data type is the generic type for all traversable nodes. For example, the sum type for GHC Core (Figure 2) is:

```
data Core = ModGutsCore ModGuts
          | BindCore    CoreBind
          | ExprCore    CoreExpr
          | AltCore     CoreAlt
```

KURE provides a type class *Node*, with an associated type function (Kiselyov et al. 2010) *Generic* that specifies the corresponding sum type. This type function is constrained to be idempotent (the sum type must itself be an instance of *Node*, and be its own *Generic*), and there must be an injective function (and a corresponding retraction) between a node and its sum type. The class also has a function to count the number of traversable children.

```
class Generic a∼Generic (Generic a) ⇒ Node a where
  type Generic a :: *
  inject  :: a → Generic a
  retract :: Generic a → Maybe a
  numChildren :: a → Int
```

Using *inject* and *retract*, KURE provides a number of combinators that lift (and lower) translations and rewrites to (and from) the sum type (see Figure 3). Notice that a failing retraction causes the entire translation to fail.

The KURE user is required to define instances of *Node* for each traversable node in her data type, but this is straightforward. For example, the instances for *CoreBind* are:

```
instance Node CoreBind where
  type Generic CoreBind = Core
  inject                = BindCore
  retract (BindCore bnd) = Just bnd
  retract _              = Nothing
  numChildren (NonRec _ _) = 1
  numChildren (Rec defs)   = length defs
```

```
injectT :: Node a ⇒ Translate c m a (Generic a)
injectT = arr inject

retractT :: Node a ⇒ Translate c m (Generic a) a
retractT = translate (λ_ → maybe (fail "...") return ∘ retract)

extractT :: Node a ⇒
            Translate c m (Generic a) b → Translate c m a b
extractT t = injectT ⋙ t

promoteT :: Node a ⇒
            Translate c m a b → Translate c m (Generic a) b
promoteT t = retractT ⋙ t

extractR :: Node a ⇒ Rewrite c m (Generic a) → Rewrite c m a
extractR r = injectT ⋙ r ⋙ retractT

promoteR :: Node a ⇒ Rewrite c m a → Rewrite c m (Generic a)
promoteR r = retractT ⋙ r ⋙ injectT
```

Figure 3. Lifting and lowering combinators.

We can now give a sensible type signature to *allR*:

```
allR :: Node a ⇒ Rewrite c m (Generic a) → Rewrite c m a
```

That is, given a rewrite that can be applied to any traversable descendant of a, then we have a rewrite over a.

We now have a type, but what about the definition? First we introduce another type class *Walker*, which contains a function that constructs a *Lens* to each traversable child (indexed by an *Int*):

```
class Node a ⇒ Walker c m a where
  childL :: Int → Lens c m a (Generic a)
```

The KURE user is required to provide this instance for each traversable node. Definitions of *childL* are often quite verbose, but fit a standard pattern. Unfortunately they cannot be generated automatically, as the user must specify how to update the context. However, KURE does provide a family of combinators to facilitate writing these instances.

Using *childL*, it is possible to define *allR*. Therefore, KURE provides *allR* as a *Walker* class method with a default definition (which can be overwritten in situations where that definition would be inefficient, such as when there is a list of children). The full definition of the *Walker* class, and some auxiliary functions, is shown in Figure 4. Notice that *Walker* provides a combinator *anyR*, which is similar to *allR* except that it succeeds if the rewrite succeeds on *any* child. Also notice the *allT* combinator, which applies a *Translate* to all children and combines the results in a monoid. A monoid is appropriate because we have an arbitrary number of results to combine together into a single value.

We can now define traversals such as *topdown* and *bottomup* from Stratego. KURE provides two variants of each strategy, depending on whether we want the rewrite to succeed *everywhere*, which is the behavior in Stratego, or merely *anywhere*, which is often more useful in the context of HERMIT.

```
alltdR r  = r ⋙ allR (alltdR r)
anytdR r  = r >+> anyR (anytdR r)

allbuR r  = allR (allbuR r) ⋙ r
anybuR r  = anyR (anybuR r) >+> r
```

A related strategy is a variant of *anytdR* that prunes at each success (i.e. it does not descend below any node at which the rewrite succeeds):

```
prunetdR r = r <+ anyR (prunetdR r)
```

As a final traversal example, the following strategy repeatedly applies a rewrite, starting with the innermost term and working outwards, until a fixed point is reached:

```
innermostR r = anybuR (r ⋙ tryR (innermostR r))
```

```
class Node a ⇒ Walker c m a where
  childL :: Int → Lens c m a (Generic a)

  allT :: Monoid b ⇒
    Translate c m (Generic a) b → Translate c m a b
  allT t = do n ← arr numChildren
              mconcat [childT i t | i ← [0..(n − 1)]]

  allR :: Rewrite c m (Generic a) → Rewrite c m a
  allR r = do n ← arr numChildren
              andR [childR i r | i ← [0..(n − 1)]]

  anyR :: Rewrite c m (Generic a) → Rewrite c m a
  anyR r = do n ← arr numChildren
              orR [childR i r | i ← [0..(n − 1)]]

childT :: Walker c m a ⇒
  Int → Translate c m (Generic a) b → Translate c m a b
childT n = focusT (childL n)

childR :: Walker c m a ⇒
  Int → Rewrite c m (Generic a) → Rewrite c m a
childR n = focusR (childL n)
```

Figure 4. The *Walker* class and interrelated functions.

3.2.4 Congruence Combinators

When traversing a tree, the context has to be updated accordingly. What that context is, and how exactly it should be updated, depends on the data type being traversed and the transformations that the user wants to be able to define. However, updating the context explicitly in the definition of each traversal combinator and primitive transformation is repetitive and error prone. It is better to define for each node a general-purpose traversal combinator that handles the context update, and then define all other combinators that manipulate that node in terms of that combinator.

These are called *congruence combinators* (Visser 2004), and we recommend the KURE user defines one for each constructor of each traversable node. This may seem like a lot of work, but our experience has been that there is a significant pay-off in the simplicity with which subsequent transformations can be defined.

More concretely, a congruence combinator builds a translation over a node from translations to apply to that node's traversable children and a function to combine the results of those translations with the leaves of the node. For example, the congruence combinator for the `Lam` constructor of *CoreExpr* is as follows:

```
lamT :: TranslateH CoreExpr a → (Id → a → b) →
        TranslateH CoreExpr b
lamT t f = translate $ λc e → case e of
  Lam b e1 → f b <$> apply t (addLamBind b c @@ 0) e1
  _        → fail "no match for Lam"
```

We won't discuss the details of the context update done by *addLamBind*; the important point is that the translation is applied to the child expression *in an updated context*.

Another, simpler, example is the congruence combinator for *Var*, which has zero traversable children:

```
varT :: (Id → b) → TranslateH CoreExpr b
varT f = translate $ λ_ e → case e of
  Var n → return (f n)
  _     → fail "no match for Var"
```

Congruence combinators are used to define the *Walker* instance for *CoreExpr*, as demonstrated in the following fragment:

```
instance Walker Context HermitM CoreExpr where
  allT t =    varT (λ_ → mempty)
           <+ lamT (extractT t) (λ_ b → b)
           <+ appT (extractT t) (extractT t) mappend
           <+ ...
```

```
allR r =   varT Var
         <+ lamT (extractR r) Lam
         <+ appT (extractR r) (extractR r) App
         <+ ...
```

The same approach is taken for defining $anyR$ and $childL$, though the definitions are more involved and so omitted here.

3.3 Comparison with StrategyLib

The key difference between StrategyLib and KURE is that KURE uses associated types to assign a type to transformations that are applied to the children of a node, whereas StrategyLib effectively fixes all transformations to operate on the generic type. Thus combinators in StrategyLib have no type parameter for the type to be operated on, and hence giving a type signature to combinators such as all is trivial.

StrategyLib also differs in that rewrites (named *TP*) are not subtypes of translations (named *TU*). This means that StrategyLib has two versions of most unary combinators, and potentially many versions for combinators that combine multiple transformations. Additionally, the restriction of TUs to operate over the generic type means that it is not possible to sequence two TUs; only two TPs, or a TP and a TU.

StrategyLib offers two implementations for the TP and TU types. The first is based on a universal representation of algebraic data types, which serves as the generic data type. The user is required to define conversion functions to and from that data type (via a type class). Applying a TP then involves translating the tree to that universal data type, performing the TP, and then translating back again. This is inefficient, particularly if only a small portion of the tree will be modified.

The second implementation encodes transformations as *rank-2 polymorphic* functions. This implementation does not suffer from the inefficient conversions of the universal representation approach, but has the significant drawback that it is necessary to know the data types to be transformed before the TP and TU types can be defined. That is, they only work for a set of pre-determined data types, they cannot be used on arbitrary structures. See Lämmel and Visser (2002) for details.

4. Primitive Transformations

HERMIT provides a number of primitive operations on GHC Core as KURE transformations, including basic lambda calculus manipulations such as β-reduction and η-expansion, as well as the local transformations described in Santos (1995). The rule which floats a let binding from an application is one such transformation:

$$f\ (\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2) \quad \Rightarrow \quad \mathbf{let}\ v = e_1\ \mathbf{in}\ f\ e_2$$

Many low-level manipulations such as these are done by GHC before the HERMIT plugin is run. Indeed, this can make testing them on real (simple) Haskell code difficult. However, providing them in HERMIT serves two purposes. They are building blocks for larger transformation strategies, and they *enable* subsequent transformations. For example, depending on the context of the application node, this transformation may enable further let-floating.

Congruence combinators (§3.2.4) can be used to concisely implement rewrites such as the one above:

```
freeVarsT :: TranslateH CoreExpr [Var]
letVarsT  :: TranslateH CoreExpr [Var]
alphaLet  :: RewriteH CoreExpr

letFloatArg :: RewriteH CoreExpr
letFloatArg = do vs <- appT freeVarsT letVarsT intersect
                 appT
                   id
                   (if null vs then id else alphaLet)
                   (λf (Let bnds e) -> Let bnds (App f e))
```

We omit the definitions of $freeVarsT$, $letVarsT$ and $alphaLet$, which collect the free variables in an expression, collect the variables bound by a let expression, and α-convert any variables bound by a let expression to globally fresh names, respectively. The first line of $letFloatArg$ uses the congruence combinator $appT$ to transform the expression App f (Let $bnds$ e) into the intersection of the free variables in f and the variables bound in $bnds$. If these lists intersect, the let expression will be α-converted to avoid variable capture before building the result.

The *primitive* transformations in HERMIT adhere to the following conventions:

- Primitive rewrites either modify a term or fail, they never succeed with an identity transformation. This makes it viable to use traversal strategies such as $innermostR$.
- For primitive rewrites, no traversal is performed. As an example, the rewrite $inline$ (§3.2) only operates on Var nodes (which are replaced by looking them up in the context). By applying a traversal/focus strategy, such as $anybuR$, to $inline$, we get a more traditional notion of inlining.
- The type of the rewrite is as specific as possible. Above, $letFloatArg$ rewrites $CoreExpr$, rather than $Core$. Lifting is done by KURE combinators when necessary (see Figure 3).

Whenever possible, primitive rewrites are lifted versions of the underlying GHC functions for manipulating Core. Care is taken to maintain the above behaviors, when they make sense. (Clearly, a substitution rewrite that does no traversal is incorrect.) In this manner, we implement substitution, free variable collection, and a de-shadowing pass, among others.

We have come to a deep appreciation of GHC's Core Lint pass (Peyton Jones and Santos 1998), which has caught unintended variable capture and other type errors numerous times in the process of writing these primitive transformations. We are especially wary of modifying types, and thankfully can rely extensively on GHC-provided functions on the occasions where we cannot avoid doing so.

4.1 Tags

Primitive transformations are categorized by tags, which are used internally to build higher-level rewrite strategies, and externally to organize them for presentation to the user. These tags denote properties of the transformation. Using a rewrite generator called $metaCmd$ and a small tag predicate language, one can quickly build a higher-level rewrite from the HERMIT dictionary. This is used to implement the $bash$[1] rewrite, which iteratively applies a group of rewrites until no more changes are made.

```
metaCmd :: TagPredicate a =>
           a ->
           ([RewriteH Core] -> RewriteH Core) ->
           RewriteH Core

bash :: RewriteH Core
bash = metaCmd (Local .& Eval) (innermostR ∘ orR)
```

The definition of $bash$ finds all the rewrites which are tagged as both Local and Eval. Local means that no traversal is performed, and so the rewrite should be fairly efficient to apply. Eval means that the rewrite performs a step of computation that reduces the term, such that repeated application of Eval rewrites will eventually terminate. The matching rewrites are then combined with orR (§3.2.1), producing a single rewrite that tries each one and succeeds if any of them succeed. This rewrite is then run using the $innermostR$ traversal strategy (§3.2.3).

[1] The name $bash$ is borrowed from the PVS System (Owre et al. 1992).

4.2 RULES

A key HERMIT command is `unfold-rule`, which constructs a rewrite from a named GHC RULES pragma (Peyton Jones et al. 2001). As an example, consider this simple Haskell program. Using `unfold-rule`, we rewrite *foo* to remove the unnecessary append:

$$\textbf{module}\ Main\ \textbf{where}$$
$$\{\text{-\# RULES "app\_nil"}\ \forall xs\,.\ xs +\!\!+ [\,] = xs\ \text{\#-}\}$$
$$foo = [1] +\!\!+ [\,]$$

```
hermit> consider 'foo
foo = (++) ▲ ((:) ▲ (__integer 1) ([] ▲)) ([] ▲)
hermit> any-bu (unfold-rule app_nil)
foo = (:) ▲ (__integer 1) ([] ▲)
```

This facility allows HERMIT users to specify domain-specific rewrites *inside* the modules they are rewriting, using Haskell syntax. RULES pragmas are type checked, but otherwise provide no guarantees as to the correctness of the rewrite—correctness is the user's responsibility. Haskell library authors already make use of RULES for domain-specific optimizations, but determining whether they are applied as expected is a significant challenge. The `unfold-rule` command provides a way for these library authors to explore rule definitions. Additionally, all of these rewrites are now available to HERMIT users for free.

5. Interfaces to HERMIT

To allow the user to interactively apply transformations to their programs during GHC compilation, HERMIT provides several interfaces at different levels of abstraction. The lowest-level interface is the HERMIT kernel, an intentionally small API of idempotent operations that is designed to support higher-level interfaces or fully automated clients. The kernel interface is directly reflected as a RESTful web service over the ubiquitous HTTP protocol, in order to support the construction of rich cross-platform user interfaces. Also building on the kernel API is the HERMIT shell, which implements additional capabilities in order to offer a familiar interactive REPL. We now overview the kernel interface and the capabilities of the shell.

5.1 Kernel

The HERMIT kernel is an agent that accepts requests to apply transformations to the syntax of individual modules. It ties together KURE and the GHC Plugins sub-system.

Specifically, the kernel provides the following operations, wrapped up as named fields in a record called $Kernel$:

$$\textbf{newtype}\ AST = \mathsf{AST}\ Int$$

$$\begin{array}{lll}
\textbf{data}\ Kernel = \mathsf{Kernel} & & \\
\quad \{\, applyK & :: & AST \to RewriteH\ Core \to IO\ AST \\
\quad ,\, queryK & :: \forall \alpha\,.\ & AST \to TranslateH\ Core\ a \to IO\ \alpha \\
\quad ,\, deleteK & :: & AST \to IO\ () \\
\quad ,\, listK & :: & IO\ [AST] \\
\quad ,\, resumeK & :: & AST \to IO\ () \\
\quad ,\, abortK & :: & \to IO\ ()\}
\end{array}$$

- AST is a handle to a specific version of a module's $ModGuts$.
- $applyK$ applies a rewrite to the specified AST and returns the handle to the resulting AST.
- $queryK$ applies a translation to the specified AST returning the resulting value.
- $deleteK$ deletes the internal record of a specific AST.
- $listK$ lists all ASTs tracked by the kernel.
- $resumeK$ halts the kernel and returns control to GHC, which compiles the specified AST.
- $abortK$ halts the kernel and exits GHC without compiling.

This API intentionally leaves the responsibility of tracking the relationship between ASTs to the client, reflecting a filesystem style of thinking. Some user interfaces, such as the REPL, are linear in nature, generally using the most recent AST. An agent which performs a search of the transformation space by speculatively applying rewrites in parallel might maintain a tree of ASTs.

Kernel clients are defined using $hermitKernel$, which accepts client functionality as an IO function over the kernel and the initial AST handle, and generates a GHC Plugins pass.

$$\begin{array}{l}
hermitKernel :: (Kernel \to AST \to IO\ ()) \\
\qquad\qquad \to ModGuts \to CoreM\ ModGuts
\end{array}$$

The client function runs in its own thread. When the kernel receives a $resumeK$ command, it kills this thread and returns the $ModGuts$ of the chosen AST to GHC for compilation.

5.2 Shell

The HERMIT shell builds on the kernel to offer a REPL interface with a rich feature set. Currently, this is the primary user interface.

- The shell interprets a monomorphic but dynamically-typed dictionary of expression language commands and makes the appropriate kernel API calls.
- KURE combinators are reflected as shell commands, with the same typing discipline.
- The shell additionally maintains a focused expression and works over the most recent AST by default, providing the ability to navigate intuitively.
- There are a number of pretty printers for abstract syntax, and renderers for different output formats, including HTML, Unicode, and LaTeX.
- Previously applied transformations can be undone or replayed. This replay tree of transformations can be branched, as well as saved to and loaded from a file.
- There is a rudimentary, automatically generated, help system, comparable to UNIX `man`.

In the next section, we use the shell interface to perform a derivation using HERMIT.

6. Case Study: Optimizing *reverse*

As our case study, we perform the derivation of the efficient version of *reverse*, as given in Gill and Hutton (2009), using equational reasoning. The objective is to get a feel for how feasible these transformations are inside the HERMIT system, as well as to illustrate the HERMIT shell on a larger example.

The formula behind this worker/wrapper derivation is as follows:

1. Transform a recursive function to explicitly use the fixed-point combinator.
2. Use the worker/wrapper transformation, after checking the preconditions.
3. Simplify to expose the underlying representation-changing functions.
4. Move these representation changers to allow fusion. This fusion produces the improvement; everything else is meant to enable this.
5. Simplify and remove the use of the explicit fixed point.

`beta-reduce`	$(\lambda v \to e_1)\ e_2$	$\Rightarrow$ **let** $v = e_2$ **in** e_1
`case-float-app`	(**case** e **of** $a_1 \to e_1$ $a_2 \to e_2$ ... $a_n \to e_n)\ e'$	$\Rightarrow$ **case** e **of** $a_1 \to e_1\ e'$ $a_2 \to e_2\ e'$ ... $a_n \to e_n\ e'$
`eta-expand ys`	e	$\Rightarrow \lambda ys \to e\ ys$
`fix-intro`	$rec\ f = body$	$\Rightarrow f = fix\ (\lambda f \to body)$
`let-float-arg`	f (**let** $v = e_1$ **in** e_2)	$\Rightarrow$ **let** $v = e_1$ **in** $f\ e_2$
`let-float-let`	**let** $v_1 =$ **let** $v_2 = e_2$ **in** e_1 **in** e	$\Rightarrow$ **let** $v_2 = e_2$ **in let** $v_1 = e_1$ **in** e
`let-intro work`	e	$\Rightarrow$ **let** $work = e$ **in** $work$
`let-subst`	**let** $v = e_1$ **in** e_2	$\Rightarrow e_2\ [e_1\ /\ v]$

Table 1. Rewrites used, directly or indirectly, in case study.

6.1 Transforming *reverse* in the HERMIT Shell

We start with the quadratic version of *reverse* (quadratic because it calls append, which is $O(n)$, for each element of the list).

$$
\begin{aligned}
&rev &&:: [a] \to [a]\\
&rev\ [\,] &&= [\,]\\
&rev\ (x : xs) &&= rev\ xs +\!\!+ [x]
\end{aligned}
$$

Loading this in HERMIT, we get the more verbose System $\mathrm{F}_C^{\uparrow}$ output:

```
hermit> consider 'rev
rev = λ ▷ →
  let rec rev = λ ds →
            case ds of wild
              [] → [] ▲
              (:) x xs →
                (++) ▲ (rev xs) ((:) ▲ x ([] ▲))
  in rev
```

We can see that there is an outer *rev*, whose right-hand side is a type lambda. This is capturing as a type argument the type of the list elements, with ▷ denoting the type argument (in this pretty printer). The inner recursive definition of *rev* is what we wish to transform. We decide that, for space and presentation reasons, we are going to turn off the rendering of type expressions. Note that this is purely an observational projection issue; the types are maintained internally. As navigation commands refer to the underlying syntax tree, typically one would operate with the symbols in place.

```
hermit> set-pp-expr-type Omit
rev = λ →
  let rec rev = λ ds →
            case ds of wild
              [] → []
              (:) x xs → (++) (rev xs) ((:) x [])
  in rev
```

We focus on the inner definition, deciding to convert it to use *fix*, by first moving into the right-hand side of the outer *rev* with the shell navigation command `down`. We then use `consider` to jump to the inner *rev*, where we introduce the *fix*. Tables 1 and 2 list all the rewrites used in this example.

```
hermit> down ; consider 'rev ; fix-intro
rev =
  fix (λ rev ds →
    case ds of wild
      [] → []
      (:) x xs → (++) (rev xs) ((:) x []))
```

We are now focused on what was the inner *rev*, and we want to apply the worker/wrapper rule. This step requires introducing the specific *wrap* and *unwrap* functions, as well as verifying a precondition (that $wrap \circ unwrap \equiv id$). We omit the verification, which is a separate, straightforward derivation. Indeed, HERMIT, as a low-level rewrite API, provides no safety checks (other than typing), and we fully expect higher-level systems on top of HERMIT to provide stronger safety properties. The HERMIT shell is a "superuser" shell.

`append`	$\forall x\ xs\ ys.$	$(x : xs) +\!\!+ ys$	$\Rightarrow$	$x : (xs +\!\!+ ys)$
`app_nil`	$\forall xs.$	$[\,] +\!\!+ xs$	$\Rightarrow$	xs
`fusion`†	$\forall xs.$	$repH \circ absH \circ xs$	$\Rightarrow$	xs
`rep_app`	$\forall xs\ ys.$	$repH\ (xs +\!\!+ ys)$	$\Rightarrow$	$repH\ xs \circ repH\ ys$
`ww`†	$\forall work.$	$fix\ work$	$\Rightarrow$	$wrap\ (fix\ (\ unwrap \circ work \circ wrap))$

Table 2. GHC RULES used in case study; † has precondition.

The *wrap* and *unwrap* functions, in this case imported from a pre-compiled module, transform between a version of *rev* that returns a list and one that returns an H-list (Hughes 1986). H-lists have a more efficient, constant time, append.

$$
\begin{aligned}
&\textbf{type}\ H\ a = [a] \to [a]\\
&repH &&:: [a] \to H\ a\\
&repH\ xs &&= (xs +\!\!+)\\
&absH &&:: H\ a \to [a]\\
&absH\ h &&= h\ [\,]\\
&unwrap &&:: ([a] \to [a]) \to ([a] \to H\ a)\\
&unwrap\ f &&= repH \circ f\\
&wrap &&:: ([a] \to H\ a) \to ([a] \to [a])\\
&wrap\ g &&= absH \circ g
\end{aligned}
$$

To invoke the worker/wrapper rule (given in Table 2) we use a command called `unfold-rule`. To direct the rule to the *fix* call site, we use the higher-order command `prune-td`:

```
hermit> prune-td (unfold-rule "ww")
rev =
  wrap
    (fix
       ((.) unwrap
          ((.)
             (λ rev ds →
                case ds of wild
                  [] → []
                  (:) x xs → (++) (rev xs) ((:) x []))
             wrap)))
```

Next, we want to inline and simplify the compose function. To give an idea of the options available to accomplish this, recall the primitive `inline` rewrite that simply replaces a value with its definition. A higher-level rewrite, `unfold`, calls `inline`, then performs β-reductions and (safe) let-inlining to clean up the result. This is typical of the suite of commands provided by HERMIT: focused commands that perform well-understood transformations are combined by higher-level combinators.

```
hermit> prune-td (unfold '.) ; prune-td (unfold '.)
rev =
  wrap
    (fix (λ x →
       unwrap
         ((λ rev ds →
             case ds of wild
               [] → []
               (:) x xs → (++) (rev xs) ((:) x []))
           (wrap x))))
```

Now we want to inline *wrap* and *unwrap* and simplify the result. This exposes our representation changing functions, *repH* and *absH*.

```
hermit> prune-td (unfold 'wrap)
hermit> prune-td (unfold 'wrap)
hermit> prune-td (unfold 'unwrap)
hermit> prune-td (unfold '.)
rev =
  (λ g x → absH (g x))
    (fix (λ x →
       (λ f x → repH (f x))
         ((λ rev ds →
             case ds of wild
               [] → []
               (:) x xs → (++) (rev xs) ((:) x []))
           ((λ g x → absH (g x)) x))))
```

We need to do some more work to get the *repH* to where we want it—at the outermost location of our expression inside *fix*. To perform this cleanup, we use `bash`.

```
hermit> bash
rev =
  let g =
        fix (λ x x →
          repH
            (case x of wild
               [] → []
               (:) x xs →
                 (++) (absH (x xs)) ((:) x [])))
  in λ x → absH (g x)
```

This appears to have gone badly wrong. `x` is bound twice by a lambda, and this code does not seem as if it would even type check! The problem is an artifact of this specific pretty printer, which only shows the *human-readable* part of variable names; there is a hidden unique number that is used to distinguish binders, so the two `xs` are actually distinct to GHC. There is a fundamental tradeoff here between clarity and correctness of representation. We could change the pretty printer to display distinct names, but this is problematic for commands like `consider`, which would then have to be aware of this automatic aliasing.

Places where (pretty-printed) bindings shadow can be automatically detected, and we are working on a longer-term solution that will work something like the post-commit hooks provided by version control tools. For now, we provide an `unshadow` command, which can be called explicitly to rename the human-readable part of binders that clash.

```
hermit> unshadow
rev =
  let g =
        fix (λ x x0 →
          repH
            (case x0 of wild
               [] → []
               (:) x1 xs →
                 (++) (absH (x xs)) ((:) x1 [])))
  in λ x → absH (g x)
```

In order to create the opportunity to fuse *repH* and *absH*, we need to float the case statement out of the argument position, effectively pushing *repH* into each alternative.

```
hermit> any-bu case-float-arg
rev =
  let g =
        fix (λ x x0 →
          case x0 of wild
            [] → repH []
            (:) x1 xs →
              repH ((++) (absH (x xs)) ((:) x1 [])))
  in λ x → absH (g x)
```

Now we apply the `rep_app` rule, which sets us up for fusion of the representation-changing functions.

```
hermit> prune-td (unfold-rule rep_app) ; bash
rev =
  let g =
        fix (λ x x0 →
          case x0 of wild
            [] → repH []
            (:) x1 xs →
              (.)
                (repH (absH (x xs))) (repH ((:) x1 [])))
  in λ x → absH (g x)
```

Now we fuse the representation-changing functions, which performs the key optimization. Again, we have skipped over the verification of a precondition given in the worker/wrapper paper, namely that this fusion law only operates in the context of the recursive call (which it does here).

```
hermit> prune-td (unfold-rule fusion)
rev =
  let g =
        fix (λ x x0 →
          case x0 of wild
            [] → repH []
            (:) x1 xs → (.) (x xs) (repH ((:) x1 [])))
  in λ x → absH (g x)
```

All that remains is cleanup. GHC could do this for us, but we give the commands used here, so that we can see the final result:

```
hermit> prune-td (unfold 'repH)
hermit> prune-td (unfold '.) ; bash
hermit> focus (consider case) (eta-expand 'ys)
hermit> any-bu case-float-app
hermit> prune-td (unfold-rule "append")
hermit> prune-td (unfold-rule "app_nil")
hermit> prune-td (unfold 'fix) ; bash ; unshadow
rev =
  let rec x = λ x0 ys →
            case x0 of wild
              [] → ys
              (:) x1 xs → x xs ((:) x1 ys)
  in λ x0 → absH (x x0)
```

We have reached our linear version of *reverse*. We also performed measurements (not given), to verify that we have indeed removed the quadratic cost of the original *rev*.

6.2 Evaluation

At several stages we needed to exploit the GHC RULES system. For example, both the `ww` rule and `fusion` rule actually have a precondition (see Gill and Hutton (2009) for more details; the specifics of the preconditions are not as important as noting that there are preconditions, and that we do not yet automatically handle them). We need to have some way of distinguishing between "regular" rules that are used by the GHC optimizer, and rules with preconditions, that are only to be used in HERMIT. Furthermore, the user may want to experiment with a non-terminating set of rules. Because of these two cases, we expect that we will somehow need to designate HERMIT-only rules inside GHC in the near future. We are experimenting with two possible solutions for this: using a witness type that only HERMIT generates, or adding a syntactical marker to stop GHC's optimizer from using a rule.

We were originally thwarted by the *foldr/build* representation used for constant lists in GHC (Peyton Jones et al. 2001), which complicated our derivations somewhat. We eventually chose to turn usage of this representation off via a compile-time flag. Again, we need a more encompassing solution going forward.

Is it easier to perform equational reasoning by hand than to use HERMIT? Yes and no. Is is extremely tedious to perform rewrites at this level, even when using a tool. HERMIT takes care of the scoping issues and rule applications, but manipulating the syntax such that the rules successfully match is an acquired skill, and in part a navigational issue. On the other hand, HERMIT may prove useful when investigating why specific GHC rules are not firing, by stepping through what the expected GHC behavior is and observing optimization opportunities (though we have not yet tried this). The next step is to build meta-transformations that can automate as much of the tedium in derivations such as this as possible, while accurately managing pre-conditions.

7. Related Work

There are a wide variety of approaches to formalizing program transformation, such as fold/unfold (Burstall and Darlington 1977), expression procedures (Scherlis 1980; Sands 1995), the CIP system (Bauer et al. 1988), and the Bird-Meertens Formalism (Meijer et al. 1991; Bird and de Moor 1997). These systems vary in their expressive power, often trading correctness for expressiveness. For example, fold/unfold is more expressive than expression procedures, but expression procedures ensures total correctness whereas fold/unfold allows transformations that introduce non-termination (Tullsen 2002).

The most mature strategy rewrite system is Stratego (Bravenboer et al. 2008), which grew out of work on a strategy language to translate RML (Visser et al. 1998), and drew inspiration from ELAN (Borovanský et al. 2001). StrategyLib (Lämmel and Visser 2002) is the system most similar to KURE, and many aspects of the KURE design were drawn from it. We overviewed Stratego in §3.1 and compared KURE and StrategyLib in §3.3. Visser (2005) surveys the strategic programming discipline.

The combinators of Ltac (Delahaye 2000), the tactics language used by the proof assistant Coq (Bertot and Castéran 2004), are very reminiscent of KURE's strategic programming combinators. The key differences are that Ltac tactics operate on proof obligations rather than tree-structured data, and that they return a *set* of sub-goals. We need to investigate what ideas can be incorporated from such tactics languages as we improve HERMIT's support for equational reasoning.

It is well known that handling name bindings when working with abstract syntax is tedious and error prone. There has been a good deal of work in this area, with *Unbound* (Weirich et al. 2011b), a Haskell-hosted DSL for specifying binding structure, being a recent solution. HERMIT uses congruence combinators for this task, which are a general mechanism for encapsulating the maintenance of *any* sort of contextual information, of which bindings are just one example.

The Haskell Refactorer (HaRe) (Brown 2008; Li and Thompson 2008) is a source-level refactoring tool for a superset of Haskell 98. HaRe is a GUI-based interface into Haskell syntax, with support for many built-in transformations. The principal difference, apart from the GUI, is that HaRe works directly on Haskell syntax, while HERMIT works on the lower-level Core. This decision allows HERMIT to support GHC extensions with ease, at the cost of not being able to output Haskell source code (although we could output Core).

Closely related to HERMIT is the Programming Assistant for Transforming Haskell (PATH) (Tullsen 2002). Both are designed to be user directed, rather than fully automated, and are targeted at regular Haskell programmers, without advanced knowledge of language semantics and formal theorem proving tools. Again, the significant difference is the choice of target language for transformations: PATH operates on its own Haskell-like language with explicit recursion.

The Ulm Transformation System (Ultra) (Guttmann et al. 2003) is very similar to PATH, although its underlying semantics are based on CIP whereas PATH develops its own formalism. A distinguishing feature of Ultra is that it operates on a subset of Haskell extended with some non-deterministic operators, thereby allowing concise non-executable specifications to be expressed and then transformed into executable programs.

HERMIT is a direct descendant of HERA (Gill 2006), and the KURE design was inspired by the HERA implementation. HERA operated on Haskell syntax using Template Haskell (Sheard and Peyton Jones 2002). One (unpublished) conclusion from HERA was that meta-transformations such as the worker/wrapper transformation need typing information, such as that provided by GHC Core. This was the original motivation for choosing GHC Core as our subject language in HERMIT. As such, HERA can be considered as an early prototype of HERMIT, now completely subsumed.

8. Conclusion and Future Work

HERMIT provides an API that allows transformations to be performed on Haskell programs as a novel optimization pass inside GHC. By using the shell or RESTful API, we can perform optimization surgery, and observe measurable improvements in Haskell programs. Now the real work starts. Can we use HERMIT to replay derivations between clear code and efficient implementations? What powerful meta-commands can we provide? What form should a GUI interface take to make navigation straightforward?

An important decision was to target Core and work inside GHC. Consequently, we can speed up programs in GHC-extended Haskell (not just Haskell 98 or Haskell 2010) by leveraging the GHC desugaring capabilities. However, this means we operate at the System $\mathrm{F}_C^{\uparrow}$ level, which has two main issues: scale and explicit types. Scale we deal with by providing high-level navigation commands such as `consider`; explicit types we deal with by using abstraction symbols, such as ▲. A lesser issue is name-clash mirages for the pretty-printers that use truncated names, but there are several possible solutions, including a smarter pretty printer, or a set of KURE combinators that detect and rename such clashes. Still, more research into all aspects of Core presentation needs to be done. Overall, our experience is that operating on System $\mathrm{F}_C^{\uparrow}$ directly is possible, and even enjoyable.

Our Eval tag is a very informal method of ensuring that *bash* terminates. We intend to give this a formal treatment in the future, perhaps in line with the recent work by Lämmel et al. (2013). We envision a family of powerful meta-transformations that are restricted to applying rewrites that are guaranteed to terminate. Doing so will require some form of meta-transformation strategy, such as Rippling (Bundy et al. 2005), to prune at each step rewrites that could lead to non-termination. These meta-commands are straying into the territory of the tactics provided by interactive proof assistants such as Coq (Bertot and Castéran 2004), and we intend to look to proof assistants for guidance in this regard.

We found the typed congruence combinators useful when structuring our code, partly because they automatically pass the correct context in the presence of scoping, which is traditionally a significant source of bugs in rewrite systems.

The `unfold-rule` command allows for the specification of transformations in Haskell syntax, in the user's Haskell program, via GHC RULES. We imagine library writers making extensive and ongoing use of this HERMIT command, because it allows them to try out their own rules inside HERMIT.

We have been working on HERMIT prototypes for many years, and are delighted that it is finally ready to be used as a basis for further research into the transformation of Haskell programs.

Acknowledgments

We would like to thank Nicolas Frisby, Janis Voigtländer, and the anonymous reviewers for their constructive feedback. This work was partially supported by the National Science Foundation, under grants CCF-1117569 and DGE-0742523.

References

F. L. Bauer, H. Ehler, A. Horsch, B. Moeller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP*. Springer-Verlag, 1988.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.

P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–98, 2001.

M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52–70, 2008.

C. M. Brown. *Tool Support for Refactoring Haskell Programs*. PhD thesis, University of Kent, 2008.

A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

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

D. Delahaye. A tactic language for the system Coq. In *Logic for Programming and Automated Reasoning*, pages 85–95. Springer, 2000.

E. Dolstra. First class rules and generic traversals for program transformation languages. Technical report, Utrecht University, 2001.

N. J. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *Transactions on Programming Languages and Systems*, 29(3), 2007.

GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.4.1*, 2012. URL `http://www.haskell.org/ghc`.

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

A. Gill. A Haskell hosted DSL for writing transformation systems. In *Domain-Specific Languages*, pages 285–309. Springer, 2009.

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

W. Guttmann, H. Partsch, W. Schulte, and T. Vullinghs. Tool support for the interactive derivation of formally correct functional programs. *Journal of Universal Computer Science*, 9(2):173–188, 2003.

J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

R. Hinze. A new approach to generic functional programming. In *Principles of Programming Languages*, pages 119–132. ACM, 2000.

R. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

O. Kiselyov, S. Peyton Jones, and C. Shan. Fun with type functions. In *Reflections on the Work of C.A.R. Hoare*, chapter 14, pages 301–331. Springer, 2010.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Types in Languages Design and Implementation*, pages 26–37. ACM, 2003.

R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *International Conference on Functional Programming*, pages 244–255. ACM, 2004.

R. Lämmel and J. Visser. Typed combinators for generic traversal. In *Practical Aspects of Declarative Programming*, pages 137–154. Springer, 2002.

R. Lämmel, S. Thompson, and M. Kaiser. Programming errors in traversal programs over structured data. *Science of Computer Programming*, 78 (10):1770–1808, 2013.

H. Li and S. Thompson. Tool support for refactoring functional programs. In *Partial evaluation and semantics-based program manipulation*, pages 199–203. ACM, 2008.

A. Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for Haskell. In *Haskell Symposium*, pages 37–48. ACM, 2010.

E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

B. O'Sullivan. `http://hackage.haskell.org/package/criterion`.

S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer-Verlag, 1992.

L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

S. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, pages 203–233. ACM, 2001.

B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell Symposium*, pages 111–122. ACM, 2008.

D. Sands. Higher-order expression procedures. In *Partial evaluation and semantics-based program manipulation*, pages 178–189. ACM, 1995.

A. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995.

W. L. Scherlis. *Expression procedures and program derivation*. PhD thesis, Stanford University, 1980.

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

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

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

E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, pages 216–238. Springer, 2004.

E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming*, pages 13–26. ACM, 1998.

D. Vytiniotis, S. Peyton Jones, and J. P. Magalhães. Equality proofs and deferred type errors. In *International Conference on Functional Programming*, pages 341–352. ACM, 2012.

S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation. In *Principles of Programming Languages*, pages 227–240. ACM, 2011a.

S. Weirich, B. A. Yorgey, and T. Sheard. Binders unbound. In *International Conference on Functional Programming*, pages 333–345. ACM, 2011b.

A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *International Conference on Functional Programming*, pages 233–244. ACM, 2009.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation*, pages 53–66. ACM, 2012.