How to Interact with a HERMIT
(Draft Extended Abstract)

Andy Gill, Andrew Farmer, Neil Sculthorpe, Adam Howell, Robert F Blair, Ryan Scott, Patrick G Flor, and Michael Tabone

ITTC / EECS
The University of Kansas
Lawrence, KS 66045

Abstract. To the user, compilers are often black-boxes. Even if a compiler can be configured with flags and parameters, the user typically only sees the end-result of those configuration options. When and where those optimizations are applied, and the changes that the intermediate representation of the code goes through, remain mysterious. Sometimes a user just wants to be able to view the intermediate code, or a particular interesting fragment of that code, before and after the optimization pass of interest, to see what changes are occurring. Sometimes it would be helpful to be able to speculatively apply a transformation in isolation, to visually see what effect that has on the intermediate code, and also what effect it has on the performance of the generated code. And if a user is designing a new optimization, it is helpful to be able to experiment with that optimization before putting in the effort of implementing it.
The Haskell Equational Reasoning Model-to-Implementation Tunnel (HERMIT) is a compiler plugin that provides transformations-as-a-service, augmenting the Glasgow Haskell Compiler (GHC) to allow engineers to login virtually into their compilation sessions, and view and manipulate GHC's core language. The HERMIT system looks after the details: what transformations are feasible; binding issues; recording undischarged preconditions; history; and other bookkeeping. An interactive language with about 100 primitive commands is available for directing transformations.
This paper is about the interactive capabilities of HERMIT, and the way it allows a user to view and manipulate GHC's core language. In this paper, we present and contrast in detail two different interfaces for HERMIT: an Android-based interface designed for code viewing and exploration, with the aim of allowing the user to understand what code is being generated, and the effects of optimizations on that code; and a shell-based interactive interface, designed to support the scripting of, and experimentally applying, new transformations.

Keywords: Program Transformation, Shells, Tablets, Optimization

1 Introduction

How do you interact with a compiler? Typically, compilers are stream processors, taking source code, and outputting machine code. You interact with a compiler

by saying something different in the source file (what you want to compile) or changing a flag (how the compilation is done), or even by using source-level pragmas that act as a hybrid between source and configuration option. We still think about and use our compilers in fundamentally the same way as we have been for 60 years, as a batch processors and black boxes.

Yet, inside the black box, there are many, many choices the compiler is making for us, some allowing escape from tedium, some with profound execution profile consequences. An example of the mundane is the layout of a stack frame; an example of the profound is the choice of representation of data structures. As compilers become more powerful and complicated, these choices, in particular the poor ones, have higher impact.

In high level languages, like Haskell, there are *many* open choices for the compiler to resolve. Immutability of structure in particular opens up many additional compiler choices. Yet our compilers have been tuned using Appel's full employment theorem for compiler writers [2], and are still externally structured like the original batch assemblers of old.

HERMIT is a compiler plugin that allows a user to freeze a batch compile session midway, and then interact with it, in the same way as a SSH session interacts with a remote machine. The internals of the program under compilation can be altered, as well as the phase ordering, in a manner analogous to the way a remote shell can alter a remote filesystem. Of course, unrestricted manipulation of the compiler's internal structures may result in next to useless output. This paper is about the choices of interactions possible in HERMIT, and in particular, channeling the choices to allow sensible exploration of options.

2 Designing a Haskell Rewriting System

We want to be able to transform functional programs, as we have been taught when learning functional programming. This is our primary objective with interacting with a compiler, and there are many useful things we can do, including space and time optimizations, API retargeting, and informal proofs. However, the pragmatics of the real world clash with this idealism from early functional programming. We therefore make the following design decisions and compromises:

- Real Haskell – We want to transform real Haskell programs, not simplifications of Haskell or toy examples. In particular, real world Haskell has typeclasses, GADTs and other advanced type-system extensions. As a design decision, we therefore interact directly with *Core*, GHCs internal language, rather than a source language. At this level, all the various GHC extensions have been unified into a single language, based on System-FC [8]. This decision has pervasive consequences, but is based on what Haskell hackers actually do – they look at Core and understand Core semantics, and then decide what to do.
- HERMIT is therefore a plugin into GHC, and interacts with the internals directly.

- We want to make the reading of Core, real Core, as easy as possible. Because of this we have a flexible Core pretty printer, with various visual abstraction options to help understand a program at different levels.
- The basic model of interaction is based on interacting with a filesystem like object, with the current AST being presented using our pretty printer. Commands can be issued that will change this structure; other commands can ask questions; and compound commands can be run to perform complex operations.
- Given the underlying ability to display and change an AST, how are commands communicated? This is the open question we are exploring. Currently, a command-like shell is the primary mechanism; we elaborate this in the next section.
- There needs to be commands for common rewriting operations, such as β-reduction, and inlining.
- We want compound commands, and the ability to search for suitable derivations. For this we build on strategic programming [9], and our domain-specific language for generic programming, KURE [4, 7].
- We are editing a structure, and as such we want the abilities of a editor (navigation, undo, reply, history, etc). The model we use is one of a tree of read-only AST, and we can navigate between these trees (like version controlled source code), as well as place focus inside specific AST nodes (like a cursor inside a file editor).
- Correctness – In what sense is a file system change or a database update correct? In the context of HERMIT, we want a exploration tool that keeps track of what we do, but does not stop us being "economical" with the original semantics. The line of reasoning goes that if we can not manipulate a program to the desired target even without formal restrictions, then adding restrictions will only make things harder. HERMIT is therefore has a design with three modes: reading, writing, and recording. Reading is read only, writing can change anything, as the interactor requests, and recording adds an audit-trail of what has been changed, in context, to be discharged externally.

Given these decisions, Fig. 1 gives the high-level architecture of HERMIT. The main change since [3], at the block diagram level, is that the shell is now a integral part of HERMIT, with three clients: the command line shell, the scripting shell, and the web-based RESTful API, all of which share a common API into the HERMIT shell.

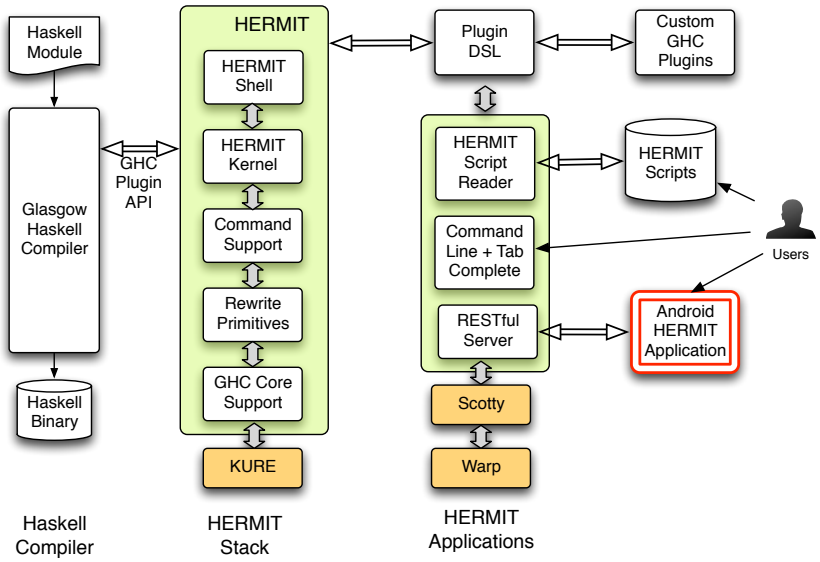

Fig. 1: Architecture of HERMIT

3 Interacting with an Intermediate Form

What commands and options do we want to give the user? What would be useful in practice, and what might be useful though not initially obvious? This paper documents our ideas, experiments, and experiences here.

We want to change how people write functional programs for the better. Specifically, how might you write a functional program if you know that a *usable*, mechanized, post-hoc system for improving programs existed. There is also some play between the architecture decisions, and the commands offered, and we are focused on the experienced users, who know Haskell well, and want to improve their programs. We leave aside much of the detail about *how* many of these commands are implemented, as the user does not concern themselves with these detail, and much of this has already been discussed. A previous paper documents the initial architecture [3]; another documents a successful attempt to mechanize a number of well-known transformations [6]. Neither paper focused on interactive nature of HERMIT.

We make heavy use of a shell-style command line. This was partly the vintage of the authors, who prefer command-line tools, and that shells naturally lead into a scripting languages. The intention always was to replace our shell with a recursive invocation of the GHCi prompt, but the shell has taken on a life of its own. The big breakthrough was the inclusion of tab-complete, with necessary contextual lookups, which turned a clunky interface into something remarkably useable overnight.

In order to help explain our possible commands, we categorize our commands into the following groups. Note that they are not mutually exclusive.

Eval ..... The arrow of evaluation (reduces a term)
KURE ..... Commands that directly reflect the KURE DSL
Loop ..... Command may operate multiple times
Deep ..... Command may make a deep change, can be O(n)
Shallow ..... Command operates on local nodes only, O(1)
Navigation ..... Navigate via focus, or directional command
Query ..... Questions we ask
Predicate ..... Something that passes or fails
Introduce ..... Introduce something, like a new name
Commute ..... Commute is when you swap nested terms
PreCondition ..... Operation has a precondition
Debug ..... Commands specifically to help debugging
VersionControl ..... Version Control for Core Syntax
Bash ..... Commands that run as part of the bash command
Context ..... Commands that use their context, like inlining

We focus on three major classifications: Navigation, Shallow/Deep, and Version Control.

3.1 Navigation

Navigation lets you get to where you want to go. These commands are the arrow keys of a traditional editor, but instead move around a read-only syntax tree. One important command is `consider`, which takes a binder-name argument, transporting the cursor to this binder. There are also commands to move up and down the tree, stash return points, reset to the root, and other navigation aids. In each case, after the commands is executed, the overall tree remains the same, just the location changes. After each navigation event, the pretty printer display the content of everything beneath the cursor, letting the user see the focus point.

3.2 Shallow and Deep

Most commands that change the tree are either Shallow or Deep rewrites. The Shallow commands make local changes at or near the cursor node, and the deep commands make changes that can scope into the entire sub-tree (though not above the cursor, only below). An example of a shallow command is η-reduction. Shallow commands are simple to explain using rewrite rules.

$$(\lambda v \to e\ v) \Longrightarrow e$$

An example of a Deep command is α-renaming. Deep commands need some form of special syntax; in this case substitution.

$$e \Longrightarrow e[v_1/v_2]$$

Rewrites can fail, and in which case no effect is recorded, and an error message is given to the user.

3.3 Version Control

A rewrite creates a entirely new AST for the Haskell program in question, raising the issue of version control. In order to facilitate experimentation, HERMIT provides quite a rich set of version control commands. Specifically, every time a transformation or navigation is done, a new labelled AST is created, with a link from its parent AST. This AST contains the whole program, and the location of the cursor.

We use a simple ASCII representation to show the nodes, and the transitions between them.

```
0 o
  | consider 'fib
1 o
  | down
2 *
```

From here the user can move to any node (think `git checkout`), replay steps, and backup through old nodes. If a new command is used on any non-leaf node, a branch is created. The shell also provides a way to pre-load commands from a file, for execution or stepping through. This branch-centric version control has turned out to be really useful in practice!

4 Android HERMIT

Given much of value-added of HERMIT is presentation and background bookkeeping, we wanted to consider an Android interface into HERMIT. We are specifically interested in tablets, like the Nexus 7 or 10, not phones, which would be too small for reading code fragements. Navigation could be done using gestures, rewrites could be buttons and menus, and version control could be a second-level navigation, again using tablet-native operations. Towards this we are implementing Armatus, an Android port of the HERMIT shell API. We are starting with a port of the shell, communicating with a remote HERMIT instance via HTTP and RESTful commands, and from there we are introducing experiments with gestures. Currently, the system is prototyped, but the connection with the server is not yet operational. We give a short overview of the prototype here.

Designing a HERMIT shell that works on a typical Android tablet computer interface requires consideration of screen size constraints, touch-based interaction, and lack of a robust keyboard. Unlike the desktop HERMIT shell, where a keyboard is sufficient for all tasks, Android devices use soft keyboards that limit typing speeds and typically lack advanced features such as arrow keys (to access previously used commands). To remedy these problems, we introduced several visual shortcuts that reduce the amount of typing needed. One such approach is Armatus's command menu. The command menu can be opened by swiping to the right, contains some basic HERMIT commands grouped into different

categories that can be expanded or closed as needed. Each command icon can be dragged after being held down with one finger.

In addition, Armatus has several other visual shortcuts. Armatus features a command history menu (that can be accessed through swiping) that stores a command icon corresponding to every previously used command (either typed out or used via drag and drop) during the session. Armatus also allows for selecting individual entries in the shell and can apply specific transformations. One transformation that lends itself well to Android is rearranging words in an entry, which can be accomplished through methods similar to dragging and dropping command icons. Extra functionality is also possible through finger gesture.s Finger gestures are linear patterns that the user traces in some path (e.g., a circle or a question mark) that could trigger specific Armatus functions (e.g., tracing a question mark could pull up help documentation).

Some parts of Armatus were designed to account for hardware restrictions of Android tablet computers (in particular, the Asus Nexus 7, which was used for testing purposes). Screen real estate is severely limited when using a keyboard in landscape mode. This provided motivation for grouping command icons into collapsible menus in the command menu. We also attempted to add some Bluetooth communication functionality in the event that an Internet connection was not available. However, Android devices vary greatly in their support of Bluetooth standards, and in particular, the Nexus 7 experiences difficulty with transferring data to a desktop computer over Bluetooth, so such functionality may not be feasible.

5 Conclusion

We have outlined the interactive nature of HERMIT, and how choices are presented to the user. The act of experimenting with the internals of functional programs is even more interactive that we envisaged, and the adoption of the version-control style approach to rewriting allows recording and replay. However, the interactions are quite low-level. We want to take these same ideas, and apply them to higher-level transformations. Already, strategic programming can be used for implementing searches, and has been used by others in this way [1]. We want to see if the worker/wrapper transformation [5] can be used as a framework for a meta-rewrite that will allow program refinement on a larger scale, but keep the interactive nature.

The final paper will include a detailed literature survey, as well as more complete overview of the Android system, and a section with observations on the usage of interactions in HERMIT.

6 Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1117569.

References

1. Adams, M., Farmer, A., Magalhes, J.P.: Optimizing syb is easy! (2013), http://www.ittc.ku.edu/csdl/fpg/files/Adams-13-OSIE.pdf, submitted to the International Conference on Functional Programming
2. Appel, A.W.: Compiling with Continuations. Cambridge University Press (1992)
3. 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: Proceedings of the ACM SIGPLAN Haskell Symposium. pp. 1–12. Haskell '12, ACM (2012), http://doi.acm.org/10.1145/2364506.2364508
4. Gill, A.: A Haskell hosted DSL for writing transformation systems. In: Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages. pp. 285–309. DSL '09, Springer-Verlag (July 2009)
5. Gill, A., Hutton, G.: The worker/wrapper transformation. Journal of Functional Programming 19(2), 227–251 (March 2009)
6. Sculthorpe, N., Farmer, A., Gill, A.: The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In: Proceedings of the 24th Symposium on Implementation and Application of Functional Languages (2013), http://www.ittc.ku.edu/csdl/fpg/files/Sculthorpe-13-HERMITinTree.pdf
7. Sculthorpe, N., Frisby, N., Gill, A.: KURE: A Haskell-embedded strategic programming language with custom closed universes (2013), http://www.ittc.ku.edu/csdl/fpg/files/Sculthorpe-13-KURE.pdf, submitted to the Journal of Functional Programming
8. Sulzmann, M., Chakravarty, M.M.T., Peyton Jones, S., n Donnelly, K.: System F with type equality coercions. In: Types in Language Design and Implementaion. pp. 53–66. ACM (2007)
9. Visser, E.: A survey of strategies in rule-based program transformation systems. Journal of Symbolic Computation 40(1), 831–873 (2005)