# Introducing the Haskell Equational Reasoning Assistant

Andy Gill

Galois Connections

12725 SW Millikan Way, Ste. 290

Beaverton, Oregon 97005

andy@galois.com

## Abstract

We introduce the new, improved version of the Haskell Equational Reasoning Assistant, which consists of an Ajax application for rewriting Haskell fragments in their context, and an API for scripting non-trivial rewrites.

### Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments – *Interactive environments, Programmer workbench.*

### General Terms

Design, Experimentation, Languages, Performance, Reliability.

### Keywords

Equational Reasoning, Transformation Tools, Ajax Applications.

## 1. Introduction

Functional programmers often appeal to equational reasoning to justify various decisions made in both design and implementation. In this abstract we introduce the Haskell Equational Reasoning Assistant (HERA), an architecture that provides both a GUI level and a batch level Haskell rewrite engine inside a single tool. The interactive interface is used to create and edit non-trivial translations that can be used to extend the batch level API; the batch level API can be used to implement powerful, context sensitive rewrites that can be provided to the interactive interface.

## 2. The Interactive Interface

Ajax applications use a web browser interface, and provide a dynamic interactive experience beyond traditional CGI scripts, with user interactions causing localized updates to the displayed web pages, rather than complete page reloads[2]. An Ajax engine implemented in Javascript turns user interactions into asynchronous HTTP requests, sending them to the Ajax-compliant web server. The engine then comprehends the responses to the asynchronous requests, and uses Javascript to redraw only the parts of the interface that need updating.

HERA is a classical Ajax application consisting of two components: a client-side Ajax engine running inside the browser customized for HERA; and a server-side Haskell application which provides the rewrite engine, and interacts with the Javascript Ajax engine via a simple web server, also written in Haskell. Figure 1 shows a screenshot of HERA in action, including the three main panels in the interface:
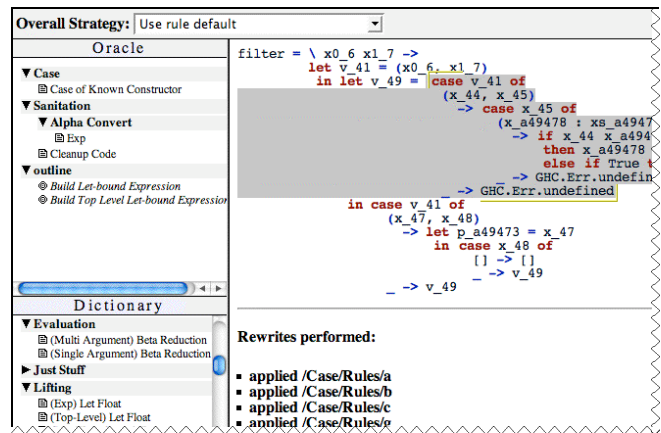
**Figure 1: HERA's Ajax Based Interface**

The **Code panel** contains the pretty-printed *selectable* code fragment, and a list of individual rewrites that have recently been applied. The Ajax engine on the client side is configured to understand the nesting of Haskell expressions, allowing the user to select only valid candidate sub-expressions, using the algorithm described in a previous paper[4]. In the screenshot in Figure 1 above, the right hand side of a let expression has been selected; a case expression.

The **Dictionary panel** contains a comprehensive list of possible rewrites. When the user clicks on a rewrite name, the rewrite is applied to the *currently selected sub-expression* in the code window, or the whole code fragment, if no sub-expression is selected. The Ajax engine asks for a specific *rewrite* to be applied to a specific *sub-expression*, using a specific *strategy*. The server application honors the request, and the engine accepts the new rendition of the code panel after the rewrite has been applied.

The **Oracle panel** gives context sensitive suggestions of possible rewrites to apply. Any time the user selects a sub-expression, and before a possible rewrite has been chosen, the Ajax engine fires an *asynchronous* HTTP request to HERA server, explaining what selection is being considered. A Haskell thread inside the server is then tasked with attempting every reasonable candidate rewrite in the internal rewrite dictionary on the selected sub-expression, sending a list of matching rewrites back to Ajax engine to display in the Oracle panel.

In practice this background search takes a fraction of a second, and includes the rewrite the user actually wants to use almost every time. In our experience, only during the "eureka" steps of rewriting does the user actually need to hunt through the dictionary. In the screenshot in Figure 1, the "Case of known constructor" has been offered to the user as a possible candidate, and examination of the code reveals that the case selected is a case of known constructor candidate.

The interface also has a *strategy* pull down menu in the top bar, which allows the user to choose between a basic range of Stratego[1] like rewrite strategies, from highly focused rewriting through to applying a generic rewrite everywhere applicable using a depth-first strategy.

## 3. Haskell Rewrites in Haskell

HERA uses the Haskell AST used by Template Haskell[6], and provides three mechanisms for specifying rewrites.

We have primitive rewrites; such as the beta-reduction and case rewrite rules provided in the Haskell Report. These are implemented as AST to AST transformers, and hidden behind an abstraction inside a trusted module. As an example, the rule (a) of the case rules listed in the Haskell report is implemented using:

```
case_rule_a :: HaskellRewrite Exp
case_rule_a = haskellRewrite' "/Case/Rules/a" $ \ e ->
  case e of
    (CaseE (VarE _) ms) -> fail "…"
    (CaseE e ms) -> do
      v <- liftQ (newName "v")
      return $ AppE (LamE [VarP v] $ CaseE (VarE v) ms) e
    _ -> fail "…"
```

We explicitly use Template Haskell for specifying equations. The equation that provides one of the monad laws is specified using:

```
monadLaw1 :: HaskellRewrite Exp
monadLaw1 = equation "/Monads/do { v <- e; return v }= e"
            $(quote [| do { v <- e; return v} |])
            $(quote [| e |])
```

We also have an embedded DSL for combining rewrites, as well as searching the rewrite space. An example of sequencing is:

```
inlineTrivialVarDCE :: HaskellRewrite Exp
inlineTrivialVarDCE
  = nameRewrite "/Inline/Inline Trivial Vars with DCE" $
      inlineTrivialVar >+> deadCodeElim
```

## 4. Rewriting Haskell Programs

HERA's interactive interface operates on code fragments — both groups of function definitions and expressions. The fragment being rewritten operates inside a rich context; conceptually any reachable part of the fragments' program might be inlined via one of the inlining rewrites in the dictionary.

HERA shares the basic purpose of the HaRe refactoring tool[5]: rewriting Haskell programs. There is an important difference: HaRe works at the source level, and is intended to help with refactoring efforts by applying well-understood software engineering patterns, while HERA handles large-scale rewrites in a different way, using only small steps applied in an explicitly bottom up approach. The HERA user focuses on one binding group at a time, eventually splitting the rewritten functions into a set of workers and wrappers[5], where the wrappers respect the original interfaces, and the workers might have a completely different implementation after rewriting. Inlining any newly created wrapper inside any call site gives correctness preserving rewrite access the worker interface.

HERA records every rewrite as an *equation*, with the left hand side as the starting fragment, and the right hand side as the fragment after all the rewrites are applied. These are exactly the same equations as are used to implement the monadic law above, and the rewrites as stored in this format for easy importing back into HERA. Any equation written using HERA stores a list of primitive rewrites or equations applied, allowing for easy playback, basic proof management and presentation opportunities. HERA will become even more useful when the record of rewrites can be stored, presented and replayed on other contexts, for example a neatly formatted proof or a proof script in a theorem prover like HOL or Isabelle, becoming a Haskell level front end to other members of the formal methods community.

## 5. History and Status

The original version of HERA was developed at Glasgow using a Tcl/Tk based interface and a lightweight functional language parser[4]. This new implementation is an attempt to scale to real Haskell, leveraging Template Haskell to do the Hard Work of parsing and typechecking Haskell. Some of the inspiration also came from Sparkle[2], an interactive rewrite tool for Clean.

HERA is an open architecture of Haskell specific rewrites, with built in audit logging. Outstanding issues include how we will capture type-based rewrites in Template Haskell's untyped AST, and how we store multiple versions of a function inside one Haskell program. We also need to complete the basic dictionary of rewrites. The next step is to apply HERA to some interesting application problems!

## 6. References

[1] Bravenboer, M., Kalleberg, K. T., Vermaas, R. and Visser. E. Stratego/XT Tutorial, Examples, and Reference Manual (latest), *Department of Information and Computing Sciences, Universiteit Utrecht,* Utrecht, The Netherlands, 2006.

[2] de Mol, M., van Eekelen, M., Plasmeijer, R., Theorem Proving for Functional Programmers - SPARKLE: A Functional Theorem Prover, *Proceedings of the 13th International Workshop on the Implementation of Functional Languages*, Älvsjö, Sweden, September 24-26, 2001, Springer-Verlag, LNCS 2312.

[3] Garrett, J.J., Ajax: A New Approach to Web Applications, http://adaptivepath.com/publications/essays/archives/, February 18, 2005.

[4] Gill, A., The Technology Behind a Graphical User Interface for an Equational Reasoning Assistant, *Proceedings of the Eighth Annual Glasgow Workshop on Functional Programming,* July 1995.

[5] Li, H. and Thompson, S. and Reinke, C., The Haskell Refactorer, HaRe, and its API, *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications*, Edinburgh, Scotland, April 2005.

[6] Peyton Jones, S.L. and Launchbury, J., Unboxed values as first class citizens, *Functional Programming Languages and Computer Architecture* (FPCA'91), Boston, LNCS 523, Springer Verlag, Sept 1991.

[7] Sheard, T. and Peyton Jones, S., Template metaprogramming for Haskell*, Proc. Haskell Workshop,* Pittsburg, 2002.