Technical Report - Porting HaRe to the GHC API

Chris Ryder and Simon Thompson

October 24, 2005

1 Background to the project

This technical report details a project occurring between February and July of 2005. The aim of the project was to assess the effort required to port the Haskell[6] refactoring tool, HaRe[4], from its current compiler front-end system, Programmatica, to the newly developed GHC[5] API.

1.1 HaRe: An automated refactoring tool

Refactoring[1] is a software development process in which modifications are made to source code to improve the structure of the code without affecting the functionality of the code. Refactorings often have conditions which specify when they can be safely applied. For example, renaming an identifier should not result in any changes to the binding structure of the code. With some programming languages it is possible to automate these refactorings, including checking the conditions associated with a refactoring. HaRe is such an automated refactoring tool for use with programs written in Haskell[6], a lazy functional programming language.

HaRe is implemented in Haskell and to perform its task it must parse and analyse Haskell source code. Rather than implement its own Haskell 'front- end' (lexer, parser, scope analysis, etc), HaRe uses the Programmatica[2] system. Implementing the refactorings often requires a significant amount of boilerplate code for traversing the Haskell abstract syntax tree representations. To further aid the implementation of the refactorings, HaRe uses the Strafunski[3] generic programming library. This is described in more detail in Section 3.2. The HaRe development team are very grateful to the contributions and support provided to the HaRe project by the Programmatica and Strafunski development teams.

Programmatica provides all the components of a Haskell front-end that HaRe needs, but has some limitations:

- Programmatica supports only Haskell 98. Most non-trivial Haskell programs now use various extensions to the Haskell 98 standard which are not supported by Programmatica. This has reduced the usefulness of the HaRe refactoring tool to practising programmers.

- Type checking in Programmatica can be slow. The implementors of the HaRe tool would like to make use of type information when performing some refactorings, but obtaining such information using Programmatica can become a performance bottleneck.

- Programmatica is not distributed with any compilers. The Programmatica tool kit is not distributed with any of the commonly used compilers and thus HaRe must either include Programmatica in its distribution or users must download Programmatica themselves. However this is a relatively minor issue.

1.2 GHC: The Glasgow Haskell Compiler

The most commonly used Haskell compiler for use in non-trivial projects is GHC, the Glasgow Haskell Compiler. Indeed, GHC Haskell has become the de facto standard Haskell dialect. GHC is an advanced optimising Haskell compiler implemented in Haskell. Initially designed as a 'work bench' in which researchers could try out implementation ideas, GHC has since become a production quality compiler that supports a large number of sophisticated language extensions and optimisations and as a result has become rather complex.

The Haskell community would like to be able to build on the GHC infrastructure in order to produce their own tools (refactoring tools, documentation tools, editor plugins, etc), but the complex nature of GHC represents a significant hurdle to reusing its code. Recognising this, the GHC development team has begun to provide an API with which Haskell programmers can gain access to internal GHC functionality.

The API, sometimes known as 'package GHC' or 'GHC as a package', exposes the entire internal structure and functionality of GHC to user programs. This is a large and complicated selection of functions, and as such its main benefit is the ease of using parts of GHC, relative to modifying or extracting parts of GHC for one's own use. However, the GHC team is in the process of producing a more simplified API intended for use by utility writers.

The three main benefits of the GHC API over the Programmatica system are:

- It provides support for all GHC compiler extensions, allowing utilities to be used with most Haskell source code.

- It is well maintained by the GHC compiler development team, and should therefore always contain support for the latest language extensions.

- In the future it will provide a layer of insulation against changes to the internal organisation of GHC.

2 The Project

Because of the uncertainty over the future of the Programmatica system, as well as its lack of support for extensions to the Haskell 98 standard, it is desirable to port the HaRe tool from the Programmatica API to the GHC API. Such a move would both secure the future support for HaRe, as well as encourage the wider adoption of the HaRe tool by enabling its use on a greater range of programs. This project aims to assess the effort required to perform such a port.

2.1 Issues to consider

When attempting to port the HaRe tool to a new front-end there are a number of issues that must be addressed. The Programmatica front-end is focused on providing facilities for a wide range of tools, whereas GHC front-end is focused on providing facilities only for a compiler. In addition to this, the GHC front-end contains a number of optimisations to reduce its memory usage or execution time, and thus often program information in GHC is not retained past the point at which it is used. These issues lead to Programmatica and GHC having quite different APIs.

Fortunately, the HaRe tool contains an abstraction module which hides some, but not all, of the Programmatica API. Thus it should be possible to port just the abstraction module to the GHC API to gain most of the functionality of HaRe. Generally, the changes required for porting HaRe to the GHC API do not appear to be widely distributed in the source code. However, although the necessary changes are fairly localised, porting to GHC requires most of the abstraction module to be rewritten. Additionally, HaRe manipulates the abstract syntax tree of the program being refactored. The abstract syntax tree representations used by Programmatica and the GHC API are different, but the syntax tree representation is not hidden by the HaRe abstraction module and so this issue will need to be addressed in each individual refactoring. The following examples shows one case where Programmatica and GHC use different representations of Haskell syntax.

```
-- The representation of a Haskell function binding in Programmatica
-- with some data constructors ellided for clarity.
-- D e      expression recursion type
--   p      pattern recursion type
--   ds     declaration recursion type
--   t      type recursion type
--   c      context recursion type
--   tp     type pattern recursion type
data DI i e p ds t c tp
    = ...
    | HsFunBind     SrcLoc [HsMatchI i e p ds]
    | HsPatBind ...
    | ...

-- The representation of a Haskell function binding in GHC with some
-- constructors ellided for clarity
data HsBind id
  = FunBind     (Located id)
Bool -- True => infix declaration
(MatchGroup id)
  | PatBind ...
  | ...
```

As well as this general issue, there were a number of smaller technical issues which arose during the project. These smaller issues are described in the next section.

3 Experiences

Performing this project required a number of tasks to be performed. These fall into the following categories.

- Familiarisation with “Package GHC”.
- Familiarisation with HaRe and Strafunski.
- Attempt to port an existing refactoring to the GHC API.

In the rest of this section we describe in more detail the individual tasks completed during the project.

3.1 Familiarisation with “Package GHC”

Before attempting to port HaRe to the Package GHC API, it was necessary to first familiarise ourselves with the facilities it provides. Very early in the project a number of the HaRe project members visited the GHC developers in Cambridge, UK. This meeting had a number of aims:

- Demonstrate to the GHC team the interest of the HaRe project in using the GHC API.
- To gain insight into the structure of the API
- To indicate to the GHC team the typical uses the HaRe tool might make of the API, with a view to shaping the higher-level, and still developing, GHC API.

Subsequently to the meeting with the GHC developers, we began to experiment with Package GHC. The first task was build Package GHC, which initially required additional options to be enabled in the CVS version of the GHC 6.5 source code. However, later CVS versions have the necessary options enabled by default.

3.2 Familiarisation with HaRe and Strafunski

The HaRe tool makes extensive use of the Strafunski library. Strafunski provides a library of generic programming routines. Generic programming is particularly useful for implementing HaRe, which traverses the abstract syntax tree of the program being refactored. Typically, implementing such syntax tree traversal routines requires a great deal of boilerplate code which is tedious to write. Strafunski provides a mechanism for generically specifying traversal routines, such that one has only to write the “interesting” functionality, with all the boilerplate code being produced by Strafunski.

Because Strafunski is such a vital building block for HaRe, it was necessary to ensure that it could be used on the data types of the GHC API, in particular the abstract syntax representation types which are directly manipulated by the refactorings. To enable Strafunski to generically traverse arbitrary data types it requires data types to be instances of two type classes, `Data` and `Typeable`.

The GHC data types did not have instances of these classes, so it was therefore necessary to add such instances and recompile GHC. However, this lead to two problems.

- GHC uses mutually recursive modules for many of its abstract syntax types. In order to break these cyclic dependencies, special `.hi-boot` files, which specify place holders for identifiers used in the cycles, must be written. However, `.hi-boot` files cannot contain instance definitions, making it difficult to add the necessary `Data` and `Typeable` instances.

  Simon Peyton-Jones kindly added support for instances in `.hi-boot` files in GHC, but using this feature would require either a three stage build process for Package GHC, or the liberal use of CPP conditions to hide the `Data` and `Typeable` instances from versions of GHC which do not have the necessary support.

  After discussions with Simon Marlow regarding this issue, it was suggested that the structure of the offending modules be refactored to break the recursion in the modules. This was done very quickly but crudely by placing all the recursive data types into a single module. This allowed the project to continue, but would obviously require further work before it could be considered for inclusion in future versions of GHC.

- For performance reasons, several data structures used by GHC incorporate unboxed types. Unfortunately GHC does not support automatically deriving instances of type classes for unboxed types. It was therefore necessary to write instances for such types manually.

3.3 An improved hasktags

GHC is distributed with a program called `hasktags`, which can be used to generate a list of the location of definitions in a Haskell source file. However, `hasktags` does not parse Haskell source code, but instead looks for type signatures. As such, it fails to list definitions without type signatures, as well as various other definitions such as instance declarations. In order to test the use of Strafunksi on the GHC data types we implemented a simple `hasktags` program that uses Package GHC to parse and type check Haskell sources files and Strafunski to extract the locations of various definitions. Although our implementation was very quickly written, it nonetheless still generates a more complete list of definition locations than the GHC `hasktags` program. The source code for our hasktags implementation is shown in Appendix A.

The implementation of our `hasktags` program showed that the necessary GHC data types now had the required instances for Strafunski to be used, and thus paved the way to start porting a refactoring.

3.4 Porting a refactoring

To begin assessing the ease or difficulty of porting HaRe to the GHC API, we chose the simplest refactoring, renaming[4, 7], as our case study. The porting process was performed by systematically attempting to compile the renaming refactoring against the GHC API, and fixing the compile errors as they occurred. This process showed that there were three main steps to porting the refactoring, which are summarised below.

- Determine how to initialise the GHC API with the source files of the program to be refactored.

- Determine how to extract GHC internal data such as bindings and scoping information.
- Determine the differences in the abstract syntax representation data types between Programmatica and the GHC API.

The porting process was centered on four modules, RefacLocUtils.hs, RefacUtils.hs, RefacTypeSyn.hs and RefacUtils.hs. Much of the porting effort involved determining how to extract information from the GHC API which is readily and easily available from Programmatica. For instance, the token stream of a source file is easily extracted from Programatica using the parseSourceFile' function, which takes a file path and returns various pieces of information about the source file, including its token stream as a list of tokens. The GHC API, however, requires a state monad to be primed with the necessary flags extracted from the GHC global state, the lexer function to be repeatedly called until the end-of-file token is returned, and finally the list of tokens extracted from the state monad. The difference between these two approaches is illustrated in the sample code below. It is also worth noting that generating the ModSummary parameter from a FilePath in the GHC example below requires several lines of code.

```
-- Extracting a token stream with Programmatica
progGetTokenStream :: FilePath -> [PosToken]
progGetTokenStream fp = ts
  where (ts,_) = parseSourceFile' fp

-- Extracting a token stream with the GHC API
ghcGetTokenStream :: Session -> ModSummary -> IO [Located Token]
ghcGetToken ses modSum =
  do
    dflags <- getSessionDynFlags ses
    let
        -- Get the string buffer containing the source code
        Just sb = ms_hspp_buf modSum
        -- Setup the monadic lexer state
        st = mkPState sb (mkSrcLoc (moduleFS (ms_mod modSum)) 0 1) dflags
        -- Extract the token stream from the lexer monad
        tokStrm :: [Located Token]
        tokStrm = case unP (lexer strmFunc) st  of { POk _ s -> s ; _ -> [] }
        -- Generate the token stream
        strmFunc :: Located Token -> P [Located Token]
        strmFunc lt@(L _ ITeof) = return [lt] -- End of file token
        strmFunc lt = do { strm <- lexer strmFunc ; return $ lt:strm }
    return tokStrm
```

Although the GHC API is obviously more verbose, it would be reasonably straightforward to construct an abstraction module that could provide a uniform interface to both the GHC and Programmatica front-ends.

Due to the limited time available for this project the renaming refactoring has not been completely ported to the GHC API. The renaming refactoring consists of two main parts, the part that extracts all the necessary information to check the preconditions and the part that performs the actual manipulation of the abstract syntax tree. The first part has been ported to the GHC API, but the second part remains as work to be completed. Completing this second part should be straightforward because the use of Strafunski minimises the coupling to the abstract syntax data types, and it would be expected that completing the port of the renaming refactoring could be completed within one or two full-time person months.

4 Final thoughts

This project attempted to port the HaRe refactoring tool to a new compiler front-end, which offers significant benefits over the system currently used by the HaRe tool.

Unfortunately, we were unable to complete the porting of the renaming refactoring in the time available to the project. It is estimated that half the renaming refactoring has been ported to the GHC API, and no major technical challenges are expected in porting the remainder of the refactoring.

It is worth noting that much of the work involved in porting the renaming refactoring is generic across all the refactorings, and therefore porting the first refactoring is most of the effort of porting HaRe.

There are two main areas of the GHC API where we would have wished for improvements. Firstly, the GHC API is very large and complicated, but there is little documentation to explain how the various functions should be used, or data structures initialised. The lack of such documentation creates a significant hurdle when beginning a project with the API.

The second area in which we would like to see improvements is the provision of a higher level API than is currently provided by GHC. For instance, such a high level API would hide as much detail about the various state mechanisms used by GHC as possible. Work is currently underway to provide such a high level API, and contributions and discussion is being actively sought by the GHC development team.

In conclusion, there do not appear to be any significant technical difficulties in porting the HaRe tool to the GHC API. Indeed the GHC API looks to be a significant contribution to the Haskell community and is likely to spawn many new projects.

Finally, we would like to thank Simon Peyton Jones and Simon Marlow for their enthusiastic support during this project.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. `http://www.refactoring.com/`.

[2] Thomas Hallgren. Haskell tools from the Programatica project. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 103–106, Uppsala, Sweden, 2003. ACM Press.

[3] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proceedings of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375, New Orleans, Louisiana, USA, January 2003. Springer-Verlag.

[4] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In Johan Jeuring, editor, *Proceedings of the ACM SIGPLAN 2003 Haskell Workshop*, pages 27–38, Uppsala, Sweden, August 2003. ACM Press.

[5] Simon Marlow and Simon Peyton Jones. The Glasgow Haskell Compiler. Microsoft Research, Cambridge, UK. `http://www.haskell.org/ghc/`, December 2004.

[6] Simon Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, Cambridge, UK, 2003. `http://www.haskell.org/definition/`.

[7] Simon Thompson, Claus Reinke, and Huiqing Le. The Refactoring Catalogue: Renaming. `http://www.cs.kent.ac.uk/projects/refactor-fp/catalogue/Renaming.html`, Oct 2005.

A An implementation of the hasktags program using Strafunski and the GHC API

```
module Main where

import System
import Control.Exception
import System.IO.Unsafe
import System.IO
import List

-- Package GHC stuff
import GHC
import ErrUtils
import Outputable
import HsSyn
import SrcLoc
import RdrName
import Name
import CRSynTypes (HsBind(..), Sig (..))

-- Strafunski stuff
import StrategyLib
import MonadRun

main :: IO ()
main = do
  args <- getArgs
  realMain args
realMain :: [String] -> IO ()
realMain [] =
  do
    putStrLn "Usage: hasktags files"
realMain args =
  (do
    args0   <- GHC.init $ "-B/usr/local/fptools/lib/ghc-6.5":args
    GHC.setMsgHandler (\m -> putStrLn ("hasktags: " ++ m))
    ses     <- GHC.newSession JustTypecheck
    dflags0 <- GHC.getSessionDynFlags ses
    (dflags1,fileish_args) <- GHC.parseDynamicFlags dflags0 args0
    dflags2 <- GHC.initPackages dflags1
    targets <- mapM (GHC.guessTarget) args
    mapM_ (GHC.addTarget ses) targets
    res     <- GHC.load ses LoadAllTargets
    case res of
      Failed ->
        do
          putStrLn "hasktags: Failed to load modules"
      Succeeded ->
        do
          putStrLn "hasktags: Succeded loading modules"
          checked <- GHC.checkModule ses (mkModule "Main")
            (\m -> ErrUtils.printErrorsAndWarnings m)
          case checked of
            Nothing   -> putStrLn "hasktags: Failed to check module."
            Just _ ->
              do
                putStrLn "hasktags: Checked module."
                -- Get the complete module graph
                modGraph <- getModuleGraph ses
                taglist <- sequence $ map
                  (\modSum ->
                    do
                      Just cmod <- GHC.checkModule ses (ms_mod modSum)
                                     (\m -> ErrUtils.printErrorsAndWarnings m)
                      let Just ps = renamedSource cmod
                      case tags ps of
                        Just ts -> return ts
                        Nothing -> return []
                  ) modGraph
                alltags <- return $ concat taglist
                sequence_ $ map (putStrLn . show) $ sort alltags

  ) ‘catchDyn‘ (\e ->
    putStr $ show (e::GhcException)
  )

-- Build a list of tags from a term
```

```
tags :: Term t => t -> Maybe [Tag]
tags m =  applyTU (stop_tdTU step) m
  where
    step = failTU `adhocTU` matchHsBind `adhocTU` matchHsModule
                            `adhocTU` matchTyClDecl `adhocTU` matchPostTcType
    matchHsModule ((HsModule (Just lmod) _ _ _ _)::HsModule Name) =
      let ss   = getLoc lmod
          mod  = unLoc lmod
          name = showSDoc $ pprModule mod
          file = tidyFileName $ show $ srcSpanFile ss
          line = srcSpanStartLine ss
          desc = "module " ++ name
      in return [Tag name file line desc]
    matchHsModule  _ = Nothing
    matchTyClDecl (ty :: TyClDecl Name) =
      case ty of
        ForeignType _ _ _ ->
          return $ doType "foreign type "
        TyData _ _ _ _ _ _ _ _ ->
          let cons = concatMap conDeclTag $ tcdCons ty
          in return $ cons ++ doType (
                                case tcdND ty of {
                                  DataType -> "data ";
                                  NewType  -> "newtype "
                              })
        TySynonym _ _ _ ->
          return $ doType "type "
        ClassDecl _ _ _ _ _ _ ->
          return $ doType "class "
      where
        conDeclTag con =
          let lnm = case unLoc con of
                    ConDecl ln _ _ _ -> ln
                    GadtDecl ln _ -> ln
              ss = getLoc lnm
              nm = unLoc lnm
              fqName = showSDoc (ppr nm)
              name    = showSDoc (ppr $ nameOccName nm)
              file = tidyFileName $ show $ srcSpanFile ss
              line = srcSpanStartLine ss
              desc = "const " ++ name
          in [ Tag fqName file line desc,
               Tag name file line desc
             ]
        doType desc =
          let ss = getLoc $ tcdLName ty
              nm = unLoc $ tcdLName ty
              fqName = showSDoc (ppr nm)
              name    = showSDoc (ppr $ nameOccName nm)
              file = tidyFileName $ show $ srcSpanFile ss
              line = srcSpanStartLine ss
          in [ Tag fqName file line (desc ++ name),
               Tag name file line (desc ++ name)
             ]
    matchHsBind (bnd :: HsBind Name) =
      case bnd of
        FunBind _ _ _ -> doFunc
        VarBind _ _  -> Nothing -- Not implemented yet
        _ -> Nothing

      where
        doFunc = case bindLN bnd of
            Nothing -> Nothing
            Just rn ->
              let ss = getLoc rn
                  nm      = unLoc rn
                  fqnName = showSDoc (ppr nm)
                  name    = showSDoc (ppr $ nameOccName nm)
                  file    = tidyFileName $ show $ srcSpanFile ss
                  line    = srcSpanStartLine ss
                  desc    = "func " ++ fqnName
              in return [Tag name file line desc, Tag fqnName file line desc]
    matchPostTcType (a :: PostTcType) = return []

bindLN :: HsBind a -> Maybe (Located a)
bindLN (FunBind ln _ _) = Just ln
bindLN _ = Nothing

-- Remove "./" from the start of a filepath
tidyFileName :: String -> String
tidyFileName ('.':'/':str) = str
tidyFileName str           = str

type TagName = String
type TagFile = String
type TagLine = Int
type TagDesc = String
-- A tag file entry
data Tag = Tag TagName TagFile TagLine TagDesc
  deriving (Eq)

instance Ord Tag where
  compare (Tag t1 _ _ _) (Tag t2 _ _ _) = compare t1 t2

instance Show Tag where
  show (Tag tag file line desc) =
    tag `sep` file `sep` (show line) `sep` ";\t\"" ++ desc ++ "\""
      where a `sep` b = a ++ '\t':b

---- END OF PROGRAM ----
```