The Haskell Refactorer, HaRe, and its API ⋆

Huiqing Li [a,1] Simon Thompson [a,1] Claus Reinke [b,1]

[a] *Computing Laboratory, University of Kent, Canterbury, UK*

[b] *Canterbury, UK*

Abstract

We demonstrate the Haskell Refactorer, HaRe, both as an example of a fully-functional tool for a complete (functional) programming language, and to show the API which HaRe provides for building source-level program transformations for Haskell. We comment on the challenges presented by the construction of this and similar tools for language frameworks and processors.

Key words: Haskell, refactoring, HaRe, program transformation API, source code, layout preservation, strategic programming, Strafunski, Programatica

Refactoring is the process of improving the design of a program whilst preserving its behaviour. Separating general software updates into functionality changes and refactorings has well-known benefits. This process is supported both by catalogues documenting the effects and side-conditions of refactoring steps and, more importantly, by tools. Tools can ensure the validity of refactoring steps by automating both the checking of the conditions for the refactoring (using various program analyses) and the application of the refactoring itself, thus making refactoring less painful and less error-prone.

As part of our project on 'Refactoring Functional Programs' [1], we have developed the Haskell Refactorer, HaRe [3], providing support for the working programmer to refactor Haskell programs. HaRe supports the full Haskell 98 language and it works within Haskell programmers' preferred tools (Emacs and Vim, according to a survey of the community). Crucially for usability, it preserves the appearance of source code programs. To do this requires comments and source code locations to be preserved (or synthesised, for 'new' code) by language processors; a challenge for existing language frameworks.

The initial release of HaRe contained a repertoire of scope-related single-module refactorings; multiple-module versions of these refactorings were added

⋆ This work is partially supported by EPSRC grant GR/R75052.

[1] Email: H.Li@kent.ac.uk, S.J.Thompson@kent.ac.uk, claus.reinke@talk21.com

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

```
-- Test1.hs
module Test1 where

g x = x : (g (x + 1))

-- Test2.hs
module Test2 where
import Test1

h y = g y
```

```
-- Test1.hs
module Test1 where

g m x = x : ((g m) (x + m))

-- Test2.hs
module Test2 where
import Test1

h y = g 1 y
```

Fig. 1. HaRe in action: generalise `g` over the subexpression `1`.

in HaRe 0.2, and various data refactorings were added in HaRe 0.3. This version restructures HaRe to expose an API to our infrastructure for implementing refactorings and more general transformations of Haskell programs.

HaRe is shown in action on a trivial example in Figure 1; the original code appears in the left-hand column and the refactored code in the right. Assuming that the expression `1` is selected within the definition of the function `g` in `Test1`, the user is prompted for the name of the variable to be added (here `m`). Applying the *generalise* refactoring will ensure that `g` is given an extra formal parameter in its definition, and that the expression selected is supplied as argument at applications of `g` throughout the project (in this case in modules `Test1` and `Test2`). Larger examples appear in our papers [3,5]

Generalisation is typical of the class of *structural* refactorings; others include renaming a definition, changing the scope of a definition to make it broader or narrower, and unfolding a definition. *Module* refactorings include various operations on import and export lists as well as supporting the move of a definition from one module to another. The *data* refactorings support a number of atomic steps which together build a transformation from a concrete `data` type to an abstract type. This requires the synthesis of various functions (selectors, discriminators) and multi-level pattern-matching removal.

Implementation

Implementing these refactorings requires information about a number of aspects of the program; we refer back to the example of Figure 1 here.

Syntax. At the heart of the transformation is the abstract syntax tree (AST) for the parsed program. To preserve comments and layout, information about comments and source code locations for all tokens is also necessary.

Static semantics. Before inserting the new formal parameter `m` it is necessary to check that this new binding does not capture any existing uses of the name. The binding analysis provides this information.

Module analysis. In a multi-module project, analysis must include all modules. For example, a binding of `m` may have been imported into `Test1`, and the static analysis must treat this possibility.

Type system. If a type declaration accompanies the definition of `g` then it has to be updated with the new type of `g` with the parameter `m` added.

Clearly, we require the full functionality of a Haskell front-end in order to implement the refactorings completely and safely. For pragmatic reasons, we build on other projects that provide a Haskell frontend and support for generic transformations of Haskell data types [3]. These projects use standard idioms for generating or embedding language processors and transformations in Haskell, which serves as an expressive integration framework for the collection of tools. We had to extend their basic functionality because losing layout information and comments is not acceptable for our application.

Preserving the appearance of source code as much as possible presents a challenge for the tool builder, since a typical compiler front end, such as Programatica [4], will discard comments and layout information at an early stage in program analysis. In order to retain the information, we work simultaneously with the token stream (for comments and layout) and the AST. In Programatica, each identifier's location information is kept in both the token stream and the AST, and the locations can serve as the bridge for connecting the tokens in the token stream with the syntax phrases in the AST.

The HaRe API for Implementing Refactorings

The architecture of HaRe has been evolved in order to expose an API to HaRe's infrastructure for implementing refactorings or general program transformations. This API contains a collection of auxiliary functions for program analysis and transformation, covering a wide range of syntax entities of Haskell 98, including identifiers, expressions, patterns, declarations, imports and exports and so forth, and provides functions such as free and declared variable analysis, simplification of multi-equation definitions, updating/adding/removing/swapping syntax phrases, etc. Together with Programatica's abstract syntax for Haskell 98 and the Strafunski [2] library for AST traversals, this API serves as the basis for implementing primitive refactorings.

In principle, our API exposes the full abstract syntax and our domain-specific utility libraries embedded in the full Haskell programming language. In practice, as will be demonstrated, only small fragments of the abstract syntax types need be dealt with thanks to support for generic strategic programming provided by Strafunski, a Haskell library of functions embodying tree-traversal strategies of various sorts, implemented in a type-generic way; using Strafunski keeps the amount of 'boilerplate' code minimal. Scope and module information is readily at hand, and the details of layout and comment preservation are handled behind the scenes.

This hiding of layout and comment preservation in the program transformation functions eliminates a major source of programming errors which were possible in pre-API case studies: these functions now modify not only the AST but also the token stream. A collection of basic token stream manipulations, such as deleting or updating a list of tokens, getting the corresponding

```
module RefacCase(ifToCase) where
import RefacUtils

ifToCase fileName beginPos endPos
  = do (_, _, mod,ts) <- parseSourceFile fileName
       -- mod: the AST; ts: the token stream.
       let exp = locToExp beginPos endPos mod ts
       case exp of
        (Exp (HsIf _ _ _))
          -> do r<-applyRefac (worker exp) (Just(mod,ts)) fileName
                writeRefactoredFiles False [r]
        _ -> error "You haven't selected a conditional expression!"
  where
  worker exp = applyTP (once_buTP (failTP `adhocTP` inExp))
    where
    inExp exp1@((Exp (HsIf  e e1 e2))::HsExpP)
      |sameOccurrence exp exp1
      = let newExp =Exp (HsCase e
                    [HsAlt loc0 (nameToPat "True") (HsBody e1) [],
                     HsAlt loc0 (nameToPat "False")(HsBody e2) []])
        in update exp1 newExp exp1
    inExp _ = mzero
```

Fig. 2. An example refactoring: from *conditional* expression to *case* expression.

tokens of an AST syntax phrase and so on, have been implemented. Underneath these token stream manipulations is a layout adjustment algorithm which keeps the program's layout correct whenever the token stream has been updated. The refactored program source is extracted from the token stream rather than from the AST.

Demonstration

The demonstration will briefly show existing refactorings – generalisation and introduction of an abstract data type – illustrating that they can be applied across a multi-module project written in full Haskell 98. Then the API will be demonstrated by implementing a new refactoring, underlining the compactness and high level nature of the code produced.

Figure 2 shows the refactoring that transforms a user-selected conditional expression into a case expression. The body of the refactoring turns a textual selection into an expression `exp`, if possible, and then calls the `worker` function to effect the transformation, taking `exp` as an argument.

In `worker` the functions `applyTP`, `once_buTP`, `failTP` and `adhocTP` are type-preserving strategy combinators from Strafunski [2]. `once_buTP` performs a bottom-up traversal of the AST, terminating after its argument function succeeds at one node. In this case its argument fails at every type except `HsExpP`, where it calls function `inExp`. This latter function transforms the

current expression into a `case` expression if the expression refers to the same occurrence as the user-selected expression, otherwise, it fails. So, the overall effect is to transform the first (and only) occurrence of `exp`.

The API function `update` replaces a syntax phrase with a new syntax phrase of the same type in both the AST and the token stream. The functions `parseSourceFile`, `locToExp`, `applyRefac`, `writeRefactoredFiles`, `sameOccurrence` and `nameToPat` are also from the API, and their meaning can be found in the API documentation available from the HaRe webpage[1].

The example in Figure 2 is extremely simple, and involves neither scope analysis nor the module system. In the demonstration, we will illustrate how scope analysis and the module system are handled in HaRe by showing the implementation of a more complex refactoring which swaps the first two arguments of a function.

Conclusions

The Haskell Refactorer, HaRe, is a pragmatically motivated example of a fully-functional tool for a complete (functional) programming language. To ensure correctness of refactorings, HaRe builds on a complex infrastructure of program analysis and transformation tools, using Haskell both as an implementation language and as an integration framework for these tools. To facilitate the implementation of further refactorings, and to enable other Haskell transformation projects to reuse HaRe's infrastructure, we have recently published an API for building source-level program transformations for Haskell.

Taken together, the tools provide a high-level 'domain specific language' for writing transformation and their side-conditions. This alone does not guarantee correctness of program transformations, but the compactness and transparency of the code as well as the isolation of common error sources allows API users to have a high degree of assurance that the implementation is faithful to their intention. It would also be possible formally to verify aspects of the implementation; we are currently working on aspects of this.

References

[1] *Refactoring Functional Programs*, http://www.cs.kent.ac.uk/projects/refactor-fp/.

[2] Lämmel, R. and J. Visser, *Generic Programming with Strafunski* (2001), http://www.cs.vu.nl/Strafunski/.

[3] Li, H., C. Reinke and S. Thompson, *Tool Support for Refactoring Functional Programs*, in: *ACM Sigplan Haskell Workshop*, 2003.

[4] *Programatica*, http://www.cse.ogi.edu/PacSoft/projects/programatica/.

[5] Thompson, S. and C. Reinke, *A Case Study in Refactoring Functional Programs*, in: *Brazilian Symposium on Programming Languages*, 2003.