HaRe
The *Ha*skell *Re*factorer

Huiqing Li
Claus Reinke
Simon Thompson

Computing Lab, University of Kent
www.cs.kent.ac.uk/projects/refactor-fp/

Outline

- Introduction
- HaRe: The Haskell Refactorer
- Demo
- The Implementation of HaRe
- Current Work
- Future Work

Outline

- **Introduction**
- HaRe: The Haskell Refactorer
- Demo
- The Implementation of HaRe
- Current Work
- Future Work

Refactoring

- What? Changing the structure of existing code … … without changing its meaning.
- Essential part of the functional programming process.
- Where? Development, maintenance, …
  - *-- to make the code easier to understand and modify*
  - *-- to improve code reuse, quality and productivity.*
- Not just programming … also proof, presentation, …

A Simple Example

- The original code

```
module Main where

pow = 2

f [] = 0
f (h:t) = h^pow + f t

main = print $ f [1..4]
```

- Refactoring 1: rename f to sumSquares to make the purpose of the function clearer.

A Simple Example (cont.)

- Code after renaming

```
module Main where

pow = 2

sumSquares [] = 0
sumSquares (h:t) = h^pow + sumSquares t

main = print $ sumSquares [1..4]
```

- Refactoring 2: demote the definition of pow to make its scope narrower.

A Simple Example (cont.)

- Code after demoting

```
module Main where

sumSquares [] = 0
sumSquares (h:t) = h^pow + sumSquares t
  where
    pow = 2

main = print $ sumSquares [1..4]
```

7

Refactoring vs Program Optimisation

- Refactoring
  - -- source-to-source
  - -- functionality-preserving
  - -- improve the design of a program
  - -- diffuse and bureaucratic
  - -- bi-directional
- Program optimisation
  - -- source-to-source
  - -- functionality-preserving
  - -- improve the efficiency of a program
  - -- focused
  - -- unidirectional

8

How to apply refactoring?

- By hand

  Tedious, error-prone, depends on extensive testing

- With machine support

  Reliable

  Low cost: easy to make large changes.

  Just as easy to un-make large changes.

  Exploratory

9

Refactoring Functional Programs

- 3-year EPSRC-funded project
  - Explore the prospects of refactoring functional programs
  - Catalogue useful refactorings
  - Look into the difference between OO and FP refactoring
  - A real life refactoring tool for Haskell programming
  - *A formal way to specify refactorings*
  - *A set of formal proofs that verify the implemented refactorings are functionality-preserving*
- Mid-project, the second HaRe release: module-aware.

10

Outline

- Introduction
- **HaRe: The Haskell Refactorer**
- Demo
- The Implementation of HaRe
- Current Work
- Future Work

11

HaRe – The Haskell Refactorer

-- A prototype tool for refactoring Haskell programs

-- Driving concerns: usability and solid basis for extensions.

-- Implemented in Haskell, using Strafunski and Programatica.

-- Full Haskell 98 coverage

-- Integrated with the two program editors: Emacs and Vim

-- Preserves both comments and layout style of the source

12

Refactorings in HaRe: Move Definition

- Move a definition
  --Demote a definition: move a definition down in the scope hierarchy to make its scope narrower.
  --Promote a definition: move a definition up in the scope hierarchy to widen its scope.

e.g. demote/promote the definition of f

```
module Main where
f [] = 0
f (h:t) = h^2 + f t
main = print $ f [1..4]
```

<=>

```
module Main where
main = print $ f [1..4]
  where
    f [] = 0
    f (h:t) = h^2 + f t
```

13

Refactorings in HaRe: Generalise

- Generalise a definition
  -- select a sub-expression of the rhs of the definition and introduce that sub-expression as a new argument to the function at each of its call sites.

e.g. generalise definition f on sub-expression 0 with new parameter name n.

```
module Main where
f [] = 0
f (h:t) = h^2 + f t
main = f [1..4]
```

=>

```
module Main where
f n [] = n
f n (h:t) = h^2 + f n t
main = f 0 [1..4]
```

14

Refactorings in HaRe … others

- Renaming
- Introduce a new definition
- Inline a definition
- Duplicate a definition
- Delete a definition
- Add or Remove an argument
- Move a definition to another module (not yet released)

15

Outline

- Introduction
- HaRe: The Haskell Refactorer
- **Demo**
- The Implementation of HaRe
- Current Work
- Future Work

16

Demo

```
module Demo(sumSquares) where
sq x = x ^ 2
sumSquares [] = 0
sumSquares (x:xs) = sq x + sumSquares xs
anotherFun = sumSquares [1..4]
```

17

Generalise Definition

```
module Demo(sumSquares) where
sq x = x ^ 2
sumSquares [] = 0
sumSquares (x:xs) = sq x + sumSquares xs
anotherFun = sumSquares [1..4]
```

18

Generalise Definition

```
module Demo(sumSquares) where
sq x = x ^ 2
sumSquares [] = 0
sumSquares (x:xs) = sq x + sumSquares xs
anotherFun = sumSquares [1..4]
```

name of new parameter?

Generalise Definition

```
module Demo(sumSquares) where
sq x = x ^ 2
sumSquares [] = 0
sumSquares (x:xs) = sq x + sumSquares xs
anotherFun = sumSquares [1..4]
```

name of new parameter? f

Generalise Definition

```
module Demo(sumSquares, sumSquares_gen) where
sq x = x ^ 2
sumSquares f [] = 0
sumSquares f (x:xs) = f x + sumSquares f xs
sumSquares_gen = sq
anotherFun = sumSquares sq [1..4]
```


Generalise Definition

```
module Main where
import Demo
main = print $ sumSquares [1..10]
```


Generalise Definition

```
module Main where
import Demo
main = print $ sumSquares sumSquares_gen [1..10]
```


Demo end

Outline

- Introduction
- HaRe: The Haskell Refactorer
- Demo
- The Implementation of HaRe
- Current Work
- Future Work

The Implementation of HaRe

- An example: Promote the definition of sq to top level.

```
-- This is an example
module Main where
sumSquares x y = sq x + sq y
   where sq :: Int->Int
         sq x = x ^ pow
         pow  = 2 :: Int
main = sumSquares 10 20
```


The Implementation of HaRe

- An example: Promote the definition of sq to top level.

```
-- This is an example
module Main where
sumSquares x y = sq x + sq y
   where sq :: Int->Int
         sq x = x ^ pow
         pow  = 2 :: Int
main = sumSquares 10 20
```

Step 1 : Identify the definition to be promoted.

The Implementation of HaRe

- An example: Promote the definition of sq to top level.

```
-- This is an example
module Main where
sumSquares x y = sq x + sq y
   where sq :: Int->Int
         sq x = x ^ pow
         pow  = 2 :: Int
main = sumSquares 10 20
```

Step 2: Is sq already defined at top level in this or other importing modules? Is sq imported from other modules?

The Implementation of HaRe

- An example: Promote the definition of sq to top level.

```
-- This is an example
module Main where
sumSquares x y = sq x + sq y
   where sq :: Int->Int
         sq x = x ^ pow
         pow  = 2 :: Int
main = sumSquares 10 20
```

Step 3: does sq use any identifiers locally defined in sumSquares?

The Implementation of HaRe

- An example: Promote the definition of sq to top level.

```
-- This is an example
module Main where
sumSquares x y = sq pow x + sq pow y
   where sq :: Int->Int->Int
         sq pow x = x ^ pow
         pow  = 2 :: Int
main = sumSquares 10 20
```

Step 4: If the answer to Step 3 is yes, then add parameters to sq and change type signature if necessary.

The Implementation of HaRe

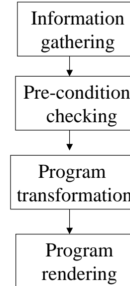
- An example: Promote the definition of sq to top level.

```
-- This is an example
module Main where

sumSquares x y = sq pow x + sq pow y
   where pow  = 2 :: Int

sq :: Int->Int->Int
sq pow x = x ^ pow

main = sumSquares 10 20
```

Step 5: Move sq to top level.

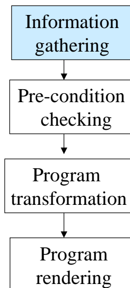
The Implementation of HaRe

- Basic steps



The Implementation of HaRe

- Basic steps



The Implementation of HaRe

- Information required … present in Programatica.

-- Abstract Syntax Tree (AST): for finding syntax phrases, e.g. the definition of sq. (need parser & lexer)

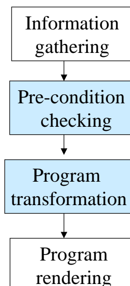
-- Static semantics: for the scope of identifiers.

-- Type information: for type-aware refactorings. (need type-checker)

-- Module information: for module-aware refactorings. (need module analysis system)

The Implementation of HaRe

- Basic steps



The Implementation of HaRe

- Pre-condition checking and program transformation

-- Our initial experience
   -- A large amount of boilerplate code for each refactoring
   -- Tiresome to write and error prone.

-- Why?
   -- The large size of the Haskell grammar: about 20 data types and 110 data constructors
   -- Both program analysis and transformation involve traversing the syntax tree frequently.

The Implementation of HaRe

- Example: code for renaming an identifier

```
instance Rename HsExp where
 rename oldName newName (Exp (HsId id))
   = Exp (HsId (rename oldName newName id))
 rename oldName newName (Exp (HsLit x)) = Exp(HsLit x)
 rename oldName newName (Exp (HsInfixApp e1 op e2))
   = Exp (HsInfixApp (rename oldName newName e1)
                      (rename oldName newName op)
                     (rename oldName newName e2))
 rename oldName newName (Exp (HsApp f e))
   = Exp (HsApp (rename oldName newName f)
                (rename oldName newName e))
 rename oldName newName (Exp(HsNegApp e))
   = Exp (HsNegApp (rename oldName newName e))

 rename oldName newName (Exp(HsLambda ps e))
    =Exp (HsLambda (rename oldName newName ps)
                   (rename oldName newName e))
```

. . . (about 200 lines)

The Implementation of HaRe

- Strafunski

-- A Haskell library developed for supporting generic programming in application areas that involve term traversal over large ASTs.

-- Allow users to write generic function that can traverse into terms with ad hoc behaviour at particular points.

-- Offers a strategy combinator library StrategyLib and a pre-processor based on DrIFT.

- DrIFT

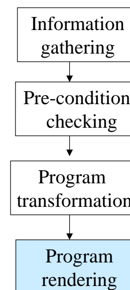
The Implementation of HaRe

- Example: renaming an identifier using Strafunski

```
rename:: (Term t)=>PName->HsName->t->Maybe t
  rename oldName newName = applyTP worker
    where
     worker = full_tdTP (idTP `adhocTP` idSite)

     idSite :: PName -> Maybe PName
     idSite v@(PN name orig)
         | v == oldName = return (PN newName orig)
     idSite pn = return pn
```


The Implementation of HaRe

- Basic steps



The Implementation of HaRe

- Program rendering

-- A real-life useful refactoring tool should preserve program layout and comments.

but,

-- layout information and comments are not preserved in AST

-- the layout produced by pretty-printer may not be satisfactory and comments are still missing

The Implementation of HaRe

- Program rendering -- example

-- program source before promoting definition sq to top level.

```
-- This is an example
module Main where
sumSquares x y = sq x + sq y
   where sq :: Int->Int
         sq x = x ^ pow
         pow  = 2 :: Int
main = print $ sumSquares 10 20
```

The Implementation of HaRe

- Program rendering -- example

-- program source from pretty printer after promoting .

```
module Main where
sumSquares x y
    = sq pow x + sq pow y where pow  = 2 :: Int
sq :: Int->Int->Int
sq pow x = x ^ pow
main = print $ sumSquares 10 20
```

43

The Implementation of HaRe

- Program rendering -- example

-- program source using our approach after promoting .

```
-- This is an example
module Main where
sumSquares x y = sq pow x + sq pow y
   where pow  = 2 :: Int
sq :: Int->Int->Int
sq pow x = x ^ pow
main = print $ sumSquares 10 20
```

44

The Implementation of HaRe

- Program rendering -- our approach

-- make use of the white space & comments in the token stream (the lexer output)

-- the refactorer takes two views of the program: the token stream and the AST

-- the modification in the AST guides the modification of the token stream.

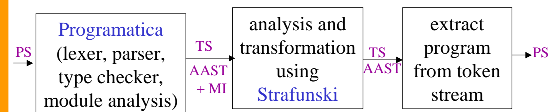
-- after a refactoring, the program source is extracted from the token stream instead of from the AST

-- use heuristics for associating comments and semantics entities.

45

The Implementation of HaRe

- The current implementation architecture



PS: program source ; TS: token stream;
AAST: annotated abstract syntax tree; MI: module information ;

46

Outline

- Introduction
- HaRe: The Haskell Refactorer
- Demo
- The Implementation of HaRe
- **Current Work**
- Future Work

47

Making refactorings module-aware

-- A refactoring may have effects in several modules

-- Effects and constraints can be subtle, choices have to be made.

-- A refactoring succeeds only if it succeeds on all affected modules in the project.

-- Built on top of Programatica's module analysis system

-- Information needed: module graph, entities imported by a module, entities exported by a module

-- What if the module is used by modules outside the project? Notify the user or create a wrapper?

48

Making refactorings module-aware

- Example: move a top-level definition f from module A to B.

-- Conditions:

-- Is f defined at the top-level of B?
-- Are the free variables in f accessible within module B?
-- Will the move require recursive modules?

-- The transformation:

-- Remove the definition of f from module A.
-- Add the definition to module B.
-- Modify the import/export in module A, B and the client modules of A and B if necessary.
-- Change the uses of A.f to B.f or f in all affected modules.
-- Resolve ambiguity.

49

Outline

- Introduction
- HaRe: The Haskell Refactorer
- Demo
- The Implementation of HaRe
- Current Work
- Future Work

50

Future work … possible collaboration

-- Other kinds of refactorings: type-aware, data-oriented, data abstraction, interface, structural, …

-- 'Not quite refactorings' and transformations …

-- An API for adding refactorings/transformations.

-- Composite refactorings and tactics, plans, strategies …

-- More complex interactions between the refactorer and the user

-- HaRe user trials: gurus, programmers, students, …

-- Semantic-aware editing, a Haskell IDE

51