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 of HaRe
- The Implementation of HaRe
- Current Work
- Future Work

Outline

- Introduction
- HaRe: The Haskell Refactorer
- Demo of HaRe
- 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]
```

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

How to apply refactoring?

- By hand

  Tedious
  Error-prone
  Depends on extensive testing

- With machine support

  Reliable
  Low cost: easy to make and un-make large changes
  Exploratory: a full part of the programmers' toolkit

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, and a set of formal proofs that the implemented refactorings are correct.
- Currently end of second year: the second HaRe is module-aware.

Outline

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

HaRe – The Haskell Refactorer

-- A prototype tool for refactoring Haskell programs

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

-- Implemented in Haskell, using Programatica’s frontends and Strafunski’s generic programming technique.

-- Full Haskell 98 coverage

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

-- Preserves both comments and layout style of the source

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
```

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.

```haskell
module Main where

f [] = 0
f (h:t) = h^2 + f t

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

=>

```haskell
module Main where

f n [] = n
f n (h:t) = h^2 + f n t

main = f 0 [1..4]
```

Refactorings in HaRe … others

Released:

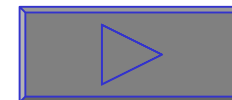
- Rename
- Introduce definition
- unfold
- Duplicate definition
- Delete definition
- Add/Remove an argument

Not yet released:

- Move definition to another module
- Clean imports
- Make imports explicit
- Add/Remove entity to/from exports
- From algebraic data type to ADT (in progress)

Outline

- Introduction
- HaRe: The Haskell Refactorer
- **Demo of HaRe (hosted in Emacs)**
- The Implementation of HaRe
- Current Work
- Future Work



Demo

```
module Demo(sumSquares) where

sq x = x ^ 2

sumSquares [] = 0
sumSquares (x:xs) = sq x + sumSquares xs

anotherFun = sumSquares [1..4]
```

Generalise Definition

```
module Demo(sumSquares) where

sq x = x ^ 2

sumSquares [] = 0
sumSquares (x:xs) = sq x + sumSquares xs

anotherFun = sumSquares [1..4]
```

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 DemoMain where
import Demo
ints = [1..10]
main = print $ sumSquares ints
```

Generalise Definition

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

Move definition to another module

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

Destination module name?

Move definition to another module

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

Destination module name? Demo

Move definition to another module

```
module DemoMain where

import Demo

main = print $ sumSquares sumSquares_gen ints
```

```
module Demo(ints,sumSquares, sumSquares_gen) where

ints = [1..10]

sq x = x ^ 2
sumSquares f [] = 0
sumSquares f (x:xs) = f x + sumSquares f xs

sumSquares_gen = sq

anotherFun = sumSquares sq [1..4]
```

Demo end

Outline

- Introduction
- HaRe: The Haskell Refactorer
- Demo of HaRe
- 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 defined at top level here or in 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 so, generalise to add these parameters and change type signature.

The Implementation of HaRe

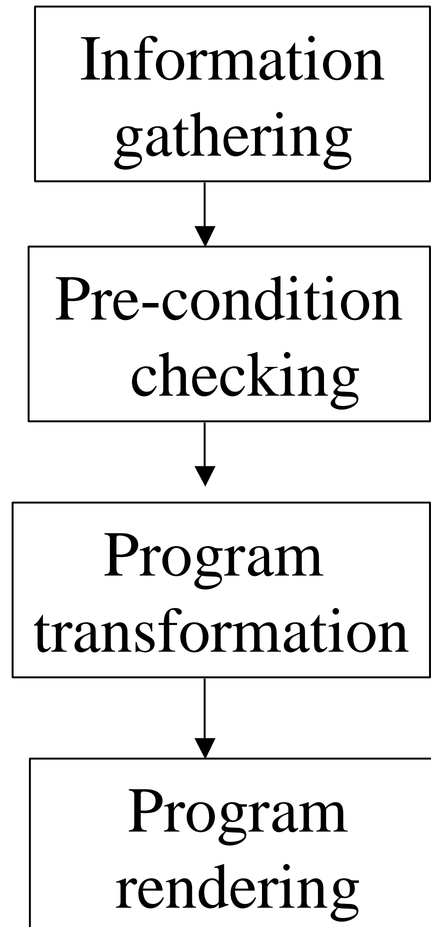
- An example: Promote the definition of sq to top level.

```
-- 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.

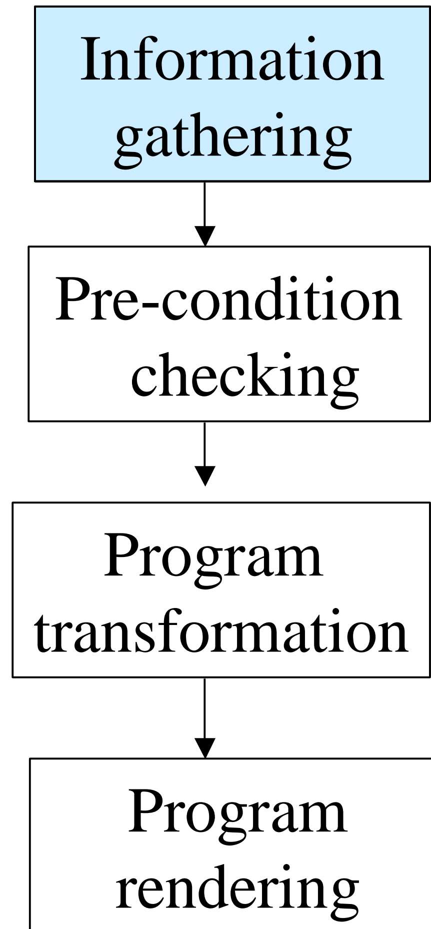
The Implementation of HaRe

- Basic steps



The Implementation of HaRe

- Basic steps



The Implementation of HaRe

- Information required

  -- 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)

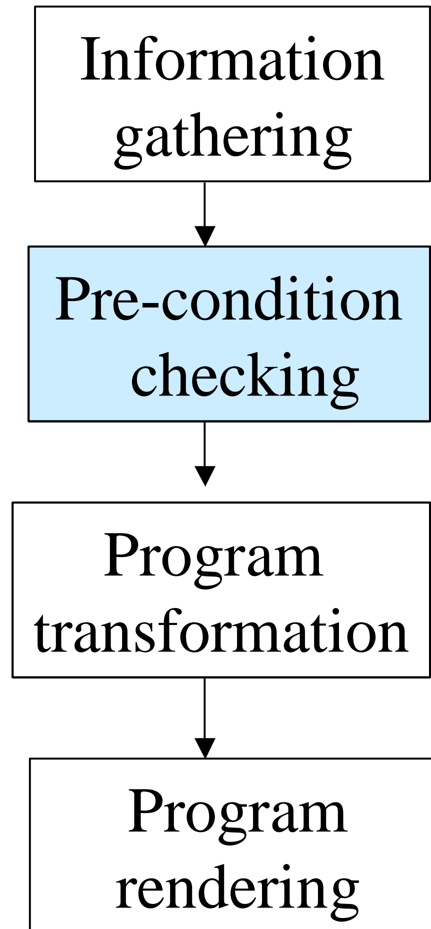
Programatica

Integrating Programming, Properties, and Validation

- Project at OGI to build a Haskell system …
- … with integral support for verification at various levels: assertion, testing, proof etc.
- The Programatica project has built a Haskell front end in Haskell, supporting syntax, static, type and module analysis, and a lexer that preserves location info.
- … freely available under BSD licence.

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 algebraic data types and the sum of 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

- Programatica's support for generic programming

-- A small selection of generic traversal operators.

-- Defined as type class instances.

-- 2-level scheme data type definitions.

-- Sensitive to changes in grammars or traversals.

The Implementation of HaRe

- Strafunski's support for generic programming

-- 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 – a generative tool.

... Strafunski: Lämmel and Visser

... DriFT: Winstanley, Wallace

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

- Our experience of using Strafunski

  -- Traversal combinators are extensively used during the development of refactorings.

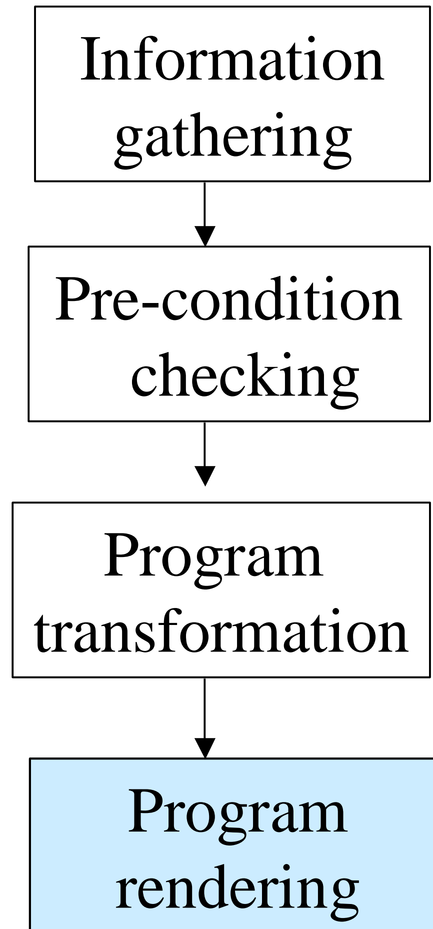
  -- Strafunski-style of programming makes the code concise. (average 200 lines per primitive refactoring). Much of the code lies on comment&layout preservation.

  -- A combinator which combines TP(type-preserving) and TU(type-unifying) would be helpful.

  -- Generic zipping is helpful too. (supported by the boilerplate approach).

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
```

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
```

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

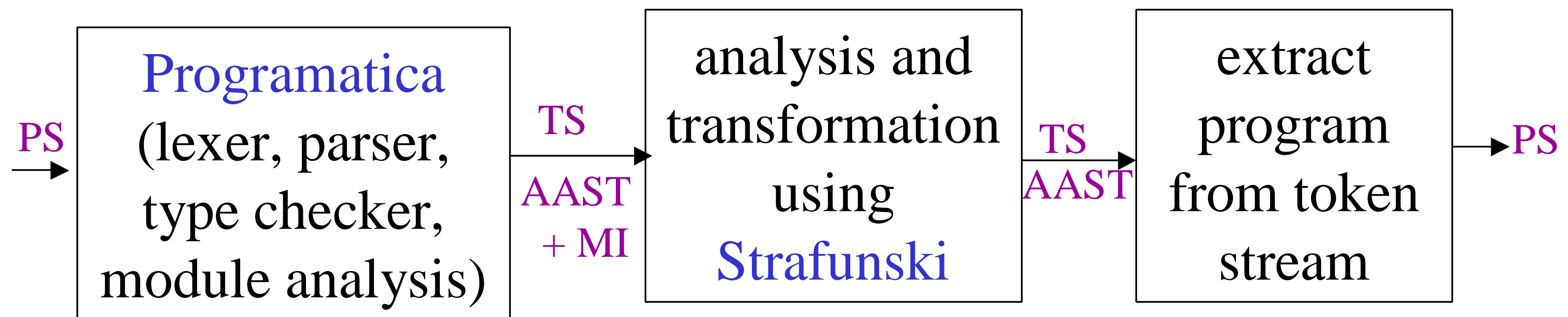
-- the modification in the AST guides the modification of the token stream.

-- 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.

The Implementation of HaRe

- The current implementation architecture



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

Outline

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

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?

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.

From Algebraic Data Type to ADT

- A large-scale refactoring.
- Can be decomposed into a series of primitive refactorings:
  - -- Introduce field labels
  - -- Create discriminators
  - -- Create constructors
  - -- Remove nested patterns
  - -- Remove patterns
  - -- Move a set of declarations to a new module
- Need to compose primitive refactorings into one composite refactoring.

Outline

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

Future work

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

-- ‘Not quite refactorings’ and transformations …

-- An API for do-it-yourself refactoring.

-- A language for composing refactorings.

-- More complex interactions between the refactorer and the user.

-- Use HaRe in teaching.