Tool Support for
Refactoring Functional Programs

http://www.cs.kent.ac.uk/projects/refactor-fp/

Huiqing Li
Claus Reinke
Simon Thompson

Computing Lab, University of Kent
Refactoring Functional Programs

• 3 year EPSRC-funded project at the University of Kent:
  – explore the prospects for refactoring in functional languages
  – catalogue useful refactorings and prototype tool support
  – look into differences between OO and FP refactoring
  – **concrete focus: Haskell refactoring** (would like to add Erlang)
  – collect and document Haskell design patterns (each refactoring changes from one design option to an alternative, implicitly documenting the pros and cons for both)
  – *we’d like a real life refactoring tool for Haskell programming*

• Now at end of year one; have focussed on:
  1. refactoring case studies, initial catalogue
  2. securing suitable infra-structure for building a refactorer for H98
  3. first prototype exists (simple, local refactorings to stress-test 2)

http://www.cs.kent.ac.uk/projects/refactor-fp/
btw, what is this “Refactoring”? 

• Refactoring, the process: 
  – is about “improving the design of existing code” (Fowler) 
  – systematically changing program structure, without changing program functionality 
  – representation-level implementation of design changes 

• Refactorings, individual steps: 
  – meaning-preserving program transformations ..

• Refactoring, context: 
  – software maintenance (separate functional and structural changes, simplify the former by supporting the latter) 
  – agile development processes: continuous design improvement and adaptation favoured over monolithic upfront design

$\delta \text{Bug} = \delta \text{Feature} = 0$
Transformations, transformations,..

- *operational semantics*, reduction to *whnf*
- *program optimisation*, source-to-source transformations to get more efficient code
- *program derivation*, calculating efficient code from obviously correct specifications
- *refactoring*, transforming code structure
  - ..

related themes, with substantial overlap, and common theory, but with different intentions
Development by transformation

- specification $\xRightarrow{\text{refine, transform}}$

- derive $\xRightarrow{}$

- program $\xRightarrow{\text{optimise}}$

- de-sugar $\xRightarrow{}$

- core program $\xRightarrow{\text{optimise}}$

- reduce $\xRightarrow{}$

- result (program) $\xRightarrow{\text{compile}}$

- code+initial state $\xRightarrow{\text{execute}}$

- code+final state $\xRightarrow{\text{de-compile}}$

where does Refactoring come in?
Between development variants

- **derive**
  - program
    - de-sugar
    - core program
    - reduce
    - variant A
  - **refactor**
    - common specification
    - program
    - de-sugar
    - core program
    - reduce
    - common result
    - variant B

- **derive**
  - program’
    - de-sugar
    - core program’
    - reduce
    - variant B
Refactoring vs modification

The better the support for refactoring,
The less look-ahead guess-work needed to anticipate functional changes
A harmless little program

```
sum []   = 0
sum (h:t) = h + sum t
main = print $ sum [1..4]
```
But: shouldn’t you write it like this?

fold c n []    = n
fold c n (h:t) = h `c` fold c n t

sum = fold (+) 0

main = print $ sum [1..4]
Or like this?

```
fold c n l | null l = n
fold c n l | otherwise = head l `c` fold c n (tail l)
sum = fold (+) 0
main = print $ sum (cons 1 (cons 2 (cons 3 (cons 4 nil))))
```

```
fold c n [] = n
fold c n (h:t) = h `c` fold c n t
sum = fold (+) 0
main = print $ sum [1..4]
```

```
sum [] = 0
sum (h:t) = h + sum t
main = print $ sum [1..4]
```
Or like what?
Or like what?

Do not try to guess ahead:
* design minimally now, refactor if necessary
Soft Ware

*design for now, refactor later (if necessary)*

- **traditional assumptions:**
  - code freezes into first form
  - any change is expensive and error prone
  - you have to get everything right first time
  
  ➔ *substantial investment in upfront analysis & design needed*

- **refactoring changes those assumptions:**
  - *code remains malleable*
  - *structural changes can be inexpensive and safe*
  - functional changes can be less expensive and safer
  - what is “right” can emerge and evolve after coding has started
  
  ➔ *incremental analysis & continuous adaptive redesign*

Sounds nice, but can that work?
mini demo
module Sum where

    sum [] = 0
    sum (h:t) = h + sum t

main = sum [1..4]
generalise definition

module Sum where

sum [] = 0
sum (h:t) = h + sum t

main = sum [1..4]
generalise definition

module Sum where

sum [] = 0
sum (h:t) = h + sum t

main = sum [1..4]

name for new parameter?
module Sum where

sum [] = 0
sum (h:t) = h + sum t

main = sum [1..4]

name for new parameter? n
generalise definition

module Sum where

sum n []    = n
sum n (h:t) = h + sum n t

main = sum 0 [1..4]
generalise definition

module Sum where

sum n [] = n
sum n (h:t) = h + sum n t

main = sum 0 [1..4]
generalise definition

module Sum where

sum n []    = n
sum n (h:t) = h + sum n t

main = sum 0 [1..4]

name for new parameter?
module Sum where

sum n []     = n
sum n (h:t)  = h + sum n t

main = sum 0 [1..4]

name for new parameter? c
module Sum where

sum c n [] = n
sum c n (h:t) = h `c` sum c n t

main = sum (+) 0 [1..4]
module Sum where

sum c n []    = n
sum c n (h:t) = h `c` sum c n t

main = sum (+) 0 [1..4]
module Sum where

\[\begin{align*}
\text{sum } c \ n \ [] & = n \\
\text{sum } c \ n \ (h:t) & = h \ `c` \ \text{sum } c \ n \ t
\end{align*}\]

main = sum (+) 0 [1..4]
module Sum where

\texttt{sum} \ c \ n \ [] \quad = \quad n
\texttt{sum} \ c \ n \ (h:t) \quad = \quad h \ `c` \ \texttt{sum} \ c \ n \ t

\texttt{main} \ = \ \texttt{sum} \ (+) \ 0 \ [1..4]

ew \ name? \ \texttt{fold}
rename definition

module Sum where

fold c n [] = n
fold c n (h:t) = h `c` fold c n t

main = fold (+) 0 [1..4]
introduce definition

module Sum where

fold c n []    = n
fold c n (h:t) = h `c` fold c n t

main = fold (+) 0 [1..4]
module Sum where

fold c n [] = n
fold c n (h:t) = h `c` fold c n t

main = fold (+) 0 [1..4]

name of new definition?
module Sum where

fold c n []    = n
fold c n (h:t) = h \ `c` \ fold c n t

main = fold (+) 0 [1..4]

name of new definition? sum
module Sum where

fold c n []    = n
fold c n (h:t) = h `c` fold c n t

main = sum [1..4]
    where
        sum = fold (+) 0
lift definition

module Sum where

fold c n [] = n
fold c n (h:t) = h `c` fold c n t

main = sum [1..4]
    where
        sum = fold (+) 0
lift definition

module Sum where

fold c n []    = n
fold c n (h:t) = h `c` fold c n t

main = sum [1..4]

sum = fold (+) 0
demo end
Tool support for refactoring

recap:

– source-level representation of design changes
– meaning-preserving program transformations

⇒ need to manipulate source-code, but not as text (semantic editing)

1. Gathering semantic info
   – lexical/syntactic/static/type analyses

2. Editing I: analyses/program transformations
   – conditional rewrite rules + rewrite strategies

3. Editing II: interaction/integration
   – retranslation/faithful presentation at source-level
   – navigation/interaction/simple editing
Tool support for Haskell Tools (1)

**gathering semantic information**

- **Ideal:** standard interface to semantic info in your favourite Haskell implementation? 😞😞 not there yet..
- Reuse code from one of the implementations/hack your own **tool-specific frontend**? 😞 common practice
- Write or find **reusable Haskell-in-Haskell frontend** for meta-programming and Haskell tool development
  - parser/prettty printer: hsspacer 😊 (haskell 98)
  - Type analysis: thih 😊 (haskell 98 + some variations)
  - p/pp+ta: hatchet 😊😊 (haskell 98, somewhat in limbo)
  - p/pp+ta+static analysis: programatica frontend 😊😊😊
    (haskell 98 + some first extensions, under active development; see Thomas’ demo this afternoon)
Tool support for Haskell Tools (2)

**Program analyses/transformation**

- you’ve got your annotated AST (scopes, types, ..)
- what about tool-specific analyses/transformations?
  - idea from optimiser implementations: combine rewrite rules and rewrite strategies in *strategic programming* dsl; implement your own traversals on top ☺ Stratego, Strafunski (the latter provides a Haskell library)
  - abstract Haskell grammar is complex and many-typed: if handwritten, the essence of traversals disappears in an unmaintainable deluge of boilerplate code 😞
  - ☻ Strafunski already addresses this problem, providing a *generic strategy library* as well as pre-processor support to instantiate it
Tool support for Haskell Tools (3)

user interaction/integration in development environment

- Refactoring is a form of semantic editing, and needs to be integrated with standard development tools and processes
- Write-your-own Haskell editor/browser:
  - full control
  - zero acceptance
  - substantial extra work
- Interface to standard editor (Emacs/Vim):
  - restricted control, divergent standards
  - easier acceptance
  - reduced extra work
Tool support for Haskell Refactorer

1. Gathering semantic info
   - Programatica’s Haskell-in-Haskell frontend

2. Editing I: program transformations/analyses
   - Strafunski: strategy library and generic programming support (currently pre-processor-based)

3. Editing II: interaction/integration
   - Text interface to refactoreer proper, used via shallow script bindings/GUI elements from Emacs and Vim

... 

retranslation/faithful presentation at source-level
Theory vs practice, an example

retranslaction/faithful presentation at source-level

• initial (bad) idea: *no problem*
  – parse/analyse ➔ transform ➔ pretty print
  – most frontends throw away aspects of your code that you’d find quite essential (comments, layout)

• revised idea: *that needs some thinking*
  – preserve layout in annotated AST?
  – extract layout “style” and imitate that in pretty-printer?
  ➔ use abstract syntax for abstract tasks, concrete syntax for concrete tasks; AST auxiliary, not intermediate representation; concrete updates on token stream
Conclusions

- Refactoring Functional Programs:
  - 3-year project at U of Kent; at the end of first year
  - Prototype *(Haskell Refactorer)*, initial release after PLI (don’t use on production sources just yet, but try it out and give us feedback)
  - *Over the next 2 years, prototype should develop into real-life tool* – neither perfect, nor complete, but in daily use
  - *Think about refactoring: it’ll change your programming*, and we’d welcome your suggestions (we have our own unbounded list of more complex refactoring candidates, though;-)
  - Practice makes the difference (implementing ideas is important!)

- Connections to non-refactoring transformations
  - Should we provide an API for extensions (so you can extend our tool for program derivation, optimisation, or ...)?

- Infrastructure for Haskell tool development is improving
  - time to undust your good ideas and implement them?