Refactoring Functional Programs

Claus Reinke
Simon Thompson

TCS Seminar, 1 October 2001

Overview

Refactoring: what does it mean?

The background in OO ... and the functional context.

Simple examples.

Discussion.

Taxonomy.

More examples.

Case study: semantic tableau.

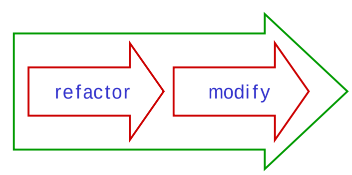
What is refactoring?

Improving the design of existing code ...

... without changing its functionality.

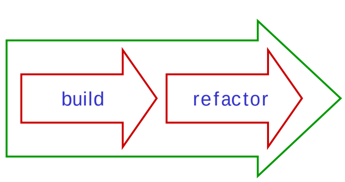
When refactor?

Prior to changing the functionality of a system.



When refactor?

After a first attempt: improving 'rubble code'.



What is refactoring?

• There is no one correct design.

• Development time re-design.

• Understanding the design of someone else's code.

Refactoring happens all the time ...

... how best to support it?

FP + SE

Functional programming provides a different view of the programming process ...

... however, it's not *that* different.

Software engineering ideas + functional programming

- design;
- testing;
- metrics;
- refactoring.

Refactoring functional programs

Particularly suited: build a prototype: revise, redesign, extend.

Semantic basis supports verified transformations.

Strong type system useful in error detection.

Testing should not be forgotten.

Genesis

Refactoring comes from the OO community ...

... associated with Martin Fowler, Kent Beck (of extreme programming), Bill Opdyke, Don Roberts.

`http://www.refactoring.com`

`http://st-www.cs.uiuc.edu/users/brant/Refactory/`

Loosely linked with the idea of design pattern.

Genesis (2)

'Refactoring comes from the OO community'

In fact the SE community got there first ...

... Program restructuring to aid software maintenance, PhD thesis, William Griswold, 1991.

OO community added the name, support for OO features and put it all into practice.

Design pattern

A stereotypical piece of design / implementation.

Often not embodied in a programming construct ...

... might be in a library, or

... more diffuse than that.

What refactoring is not

Changing functionality.

Transformational programming ... in the sense of the squiggol school, say.

Generalities

Changes not limited to a single point or indeed a single module: diffuse and bureaucratic.

Many changes bi-directional.

Tool support very valuable.

Simple examples

Simple examples from the Haskell domain follow.

Idea: refactoring in practice would consist of a sequence of small - almost trivial - changes …

… come back to this.

Renaming

```
f x y = …
```

```
findMaxVolume x y = …
```

⇐ Name may be too specific, if the function is a candidate for reuse.

⇒ Make the specific purpose of the function clearer.

Lifting / demoting

```
f x y = … h …
        where
        h = …
```

```
f x y = … (h x y) …

h x y = …
```

⇐ Hide a function which is clearly subsidiary to `f`; clear up the namespace.

⇒ Makes `h` accessible to the other functions in the module (and beyond?).

Naming a type

```
f :: Int -> Char
g :: Int -> Int
…
```

```
type Length = Int
f :: Length -> Char
g :: Int -> Length
```

⇐ Reuse supported (a synonym is transparent, but can be misleading).

⇒ Clearer specification of the purpose of `f`,`g`. (Morally) can only apply to lengths.

Opaque type naming

```
f :: Int -> Char
g :: Int -> Int
…
```

```
type Length
  = Length {length::Int}
f' :: Length -> Char
g' :: Int -> Length

f' = f . length
g' = Length . g
```

⇐ Reuse supported.

⇒ Clearer specification of the purpose of `f`,`g`. Can only apply to lengths.

The scope of changes

```
f :: Int -> Char          f :: Length -> Char
g :: Int -> Int           g :: Int -> Length
```

Need to modify …

… the calls to **f**

… the callers of **g**.

Bureaucracy

How to support these changes?

Editor plus type checker.

The rôle of testing in the OO context.

In the functional context much easier to argue that verification can be used.

Machine support

Different levels possible

Show all call sites, all points at which a particular type is used …

… change at all these sites.

Integration with existing tools (vi, etc.).

More examples

More complex examples in the functional domain; often link with data types.

Three case studies

- shapes: from algebraic to existential types;
- a collection of modules for regular expressions, NFAs and DFAs: heavy use of sets;
- a collection of student implementations of propositional semantic tableaux.

Algebraic or abstract type?

```
data Tr a
 = Leaf a |
   Node a (Tr a) (Tr a)

flatten :: Tr a -> [a]

flatten (Leaf x) = [x]
flatten (Node s t)
 = flatten s ++
   flatten t
```

```
isLeaf, isNode,
leaf, left, right,
mkLeaf, mkNode

flatten :: Tr a -> [a]

flatten t
 | isleaf t = [leaf t]
 | isNode t
  = flatten (left t) ++
    flatten (right t)
```


Algebraic or abstract type?

$\Leftarrow$

Pattern matching syntax is more direct …

… but can achieve a considerable amount with field names.

Other reasons?

$\Rightarrow$

Allows changes in the implementation type without affecting the client: e.g. might memoise values of a function within the representation type (itself another refactoring…).

Allows an invariant to be preserved.

Migrate functionality

```
isLeaf, isNode,
leaf, left, right,
mkLeaf, mkNode

depth :: Tr a -> Int

depth t
 | isleaf t = 1
 | isNode t
   = 1 +
     max (depth (left t))
        (depth (right t))
```

```
isLeaf, isNode,
leaf, left, right,
mkLeaf, mkNode, depth
```


Migrate functionality

⇐

If the type is reimplemented, need to reimplement everything in the signature, including `depth`.

The smaller the signature the better, therefore.

⇒

Can modify the implementation to memoise values of `depth`, or to give a more efficient implementation using the concrete type.

Algebraic or existential type?

```
data Shape
  = Circle Float |
    Rect Float Float …

area :: Shape -> Float
area (Circle f) = pi*r^2
area (Rect h w) = h*w

perim :: Shape -> Float
 …
```

```
data Shape
= forall a. Sh a => Shape a

class Sh a where
 area :: a -> Float
 perim :: a -> Float

data Circle = Circle Float

instance Sh Circle
 area (Circle f)  = pi*r^2
 perim (Circle f) = 2*pi*r
```


Algebraic or existential?

⇐

Pattern matching is available.

Possible to deal with binary methods: how to deal with `==` on `Shape` as existential type?

⇒

Can add new sorts of `Shape` e.g. `Triangle` without modifying existing working code.

Functions are distributed across the different `Sh` types.

Replace function by constructor

```
data Expr = Star Expr |
       Then Expr Expr | …

plus e = Then e (Star e)
```

```
data Expr = Star Expr |
            Plus Expr |
       Then Expr Expr | …
```

⇐

`plus` is just syntactic sugar; reduce the number of cases in definitions.

[Character range is another, more pertinent, example.]

⇒

Can treat `Plus` differently, e.g.

```
literals (Plus e)
  = literals e
```

but require each function over `Expr` to have a `Plus` clause.

Set comprehensions (Haskell-specific)

```
makeSet
 [ f x y |
    x <- flatten xs,
    y <- flatten ys,
    p x y ]
```

```
do x <- xs
   y <- ys
   guard (p x y)
   return (f x y)
```

⇐

The notation looks more like {f x y | x<-xs, y<-ys, p x y}; the monadic notation has a different connotation.

⇒

Doesn't require the abstraction to be broken by `flatten :: Set a -> [a]`

Might not be possible to define a `flatten` function.

Other examples ...

Modify the return type of a function from `T` to `Maybe T`, `Either T T'` or `[T]`.

Would be nice to have field names in `Prelude` types.

Add an argument; (un)group arguments; reorder arguments.

Move to monadic presentation.

Flat or layered datatypes (`Expr`: add `BinOp` type).

Various possibilities for error handling/exceptions.

What now?

Grant application: catalogue, tools, cast studies.

Online catalogue started.

Develop taxonomy.

A 'live' example?

A classification scheme

- name (a phrase)
- label (a word)
- left-hand code
- right-hand code
- comments
  - l to r
  - r to l
  - general
- primitive / composed
- cross-references
  - internal
  - external (Fowler)

- category (just one) or …
- … classifiers (keywords)
- language
  - specific (Haskell, ML etc.)
  - feature (lazy etc.)
- conditions
  - left / right
  - analysis required (e.g. names, types, semantic info.)
  - which equivalence?
- version info
  - date added
  - revision number

Case study: semantic tableaux

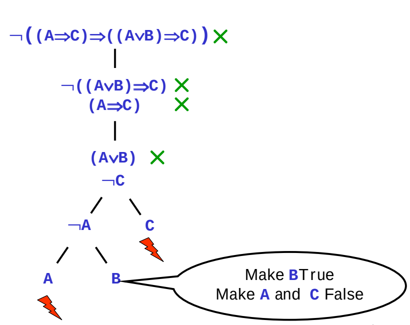
Take a working semantic tableau system written by an anonymous 2nd year student …

… refactor as a way of understanding its behaviour.

Nine stages of unequal size.

Reflections afterwards.

An example tableau



v1: Name types

Built-in types

```
[Prop]
[[Prop]]
```

used for branches and tableaux respectively.

Modify by adding

```
type Branch = [Prop]
type Tableau = [Branch]
```

Change required throughout the program.

Simple edit: but be aware of the order of substitutions: avoid

```
type Branch = Branch
```

v2: Rename functions

Existing names

```
tableaux
removeBranch
remove
```

become

```
tableauMain
removeDuplicateBranches
removeBranchDuplicates
```

and add comments clarifying the (intended) behaviour.

Add `test` datum.

Discovered some edits undone in stage 1.

Use of the type checker to catch errors.

`test` will be useful later?

v3: Literate → normal script

Change from literate form:

```
Comment …

> tableauMain tab
>    = ...
```

to

```
-- Comment …

tableauMain tab
   = ...
```

Editing easier: implicit assumption was that it was a normal script.

Could make the switch completely automatic?

v4: Modify function definitions

From explicit recursion:

```
displayBranch
  :: [Prop] -> String
displayBranch [] = []
displayBranch (x:xs)
   = (show x) ++ "\n" ++
     displayBranch xs
```

to

```
displayBranch
  :: Branch -> String
displayBranch
 = concat . map (++"\n") . map show
```

More abstract … move somewhat away from the list representation to operations such as `map` and `concat` which could appear in the interface to any collection type.

First time round added incorrect (but type correct) redefinition … only spotted at next stage.

Version control: undo, redo, merge, … ?

v5: Algorithms and types (1)

```
removeBranchDup :: Branch -> Branch
removeBranchDup [] = []
removeBranchDup (x:xs)
    | x == findProp x xs  = []  ++ removeBranchDup xs
    | otherwise           = [x] ++ removeBranchDup xs

findProp :: Prop -> Branch -> Prop
findProp z [] = FALSE
findProp z (x:xs)
    | z == x     = x
    | otherwise  = findProp z xs
```


v5: Algorithms and types (2)

```
removeBranchDup :: Branch -> Branch
removeBranchDup [] = []
removeBranchDup (x:xs)
    | findProp x xs     = []  ++ removeBranchDup xs
    | otherwise         = [x] ++ removeBranchDup xs

findProp :: Prop -> Branch -> Bool
findProp z [] = False
findProp z (x:xs)
    | z == x     = True
    | otherwise  = findProp z xs
```


v5: Algorithms and types (3)

```
removeBranchDup :: Branch -> Branch
removeBranchDup = nub

findProp :: Prop -> Branch -> Bool
findProp = elem
```

v5: Algorithms and types (4)

```
removeBranchDup :: Branch -> Branch
removeBranchDup = nub
```

Fails the `test`! Two duplicate branches output, with different ordering of elements.

The algorithm used is the 'other' `nub` algorithm, `nubVar`:

```
nub    [1,2,0,2,1] = [1,2,0]
nubVar [1,2,0,2,1] = [0,2,1]
```

The code is dependent on using lists in a particular order to represent sets.

v6: Library function to module

Add the definition:

```
nubVar = …
```

to the module

```
ListAux.hs
```

and replace the definition by

```
import ListAux
```

Editing easier: implicit assumption was that it was a normal script.

Could make the switch completely automatic?

v7: Housekeeping

Remanings: including `foo` and `bar` and `contra` (becomes `notContra`).

An instance of filter, `looseEmptyLists` is defined using `filter`, and subsequently inlined.

Put auxiliary function into a `where` clause.

Generally cleans up the script for the next onslaught.

v8: Algorithm (1)

```
splitNotNot :: Branch -> Tableau
splitNotNot ps = combine (removeNotNot ps) (solveNotNot ps)

removeNotNot :: Branch -> Branch
removeNotNot [] = []
removeNotNot ((NOT (NOT _)):ps) = ps
removeNotNot (p:ps) = p : removeNotNot ps

solveNotNot :: Branch -> Tableau
solveNotNot [] = [[]]
solveNotNot ((NOT (NOT p)):_) = [[p]]
solveNotNot (_:ps) = solveNotNot ps
```


v8: Algorithm (2)

`splitXXX` `removeXXX` `solveXXX`
are present for each of nine rules.

The algorithm applies rules in a prescribed order, using an integer value to pass information between functions.

Aim: generic versions of `split` `remove` `solve`

Have to change order of rule application …
… which has a further effect on duplicates.

Add `map sort` to top level pipeline prior to duplicate removal.

v9: Replace lists by sets.

Wholesale replacement of lists by a `Set` library.

`map`	`mapSet`	
`foldr`	`foldSet`	(careful!)
`filter`	`filterSet`	

The library exposes the representation: `pick`, `flatten`.
Use with discretion … further refactoring possible.

Library needed to be augmented with

```
primRecSet :: (a -> Set a -> b -> b) -> b -> Set a -> b
```

v9: Replace lists by sets (2)

Drastic simplification: no need for explicit worries about
… ordering and its effect on equality,
… (removal of) duplicates.

Difficult to test whilst in intermediate stages: the change in a type is all or nothing …
… work with dummy definitions and the type checker.

Further opportunities:
… why choose one rule from a set when could apply to all elements at once? Gets away from picking on one value (and breaking the set interface).

Conclusions of the case study

Heterogeneous process: some small, some large.

Are all these stages strictly refactorings: some semantic changes always necessary too?

Importance of type checking for hand refactoring …
… and testing when any semantic changes.

Undo, redo, reordering the refactorings … CVS.

In this case, directional … not always the case.

What next?

Put the catalogue into the full taxonomic form.

Continue taxonomy: look at larger case studies etc.

Towards a tool design.