Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶

Go Back
Full Screen
Close

Refactoring for Comprehension

Gustavo Villavicencio
Facultad de Matemática Aplicada
Universidad Católica de Santiago del Estero
Santiago del Estero, Argentina

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶
Page 2 of 22
Go Back
Full Screen
Close

1. Refactoring

Concept from external link

Refactoring is about “improving the design of existing code” and as such, it has been practised as long as programs have been written. The term refactoring specifically refers to a common activity in programming and software maintenance: changing the structure of a program without changing its semantics.

Or maybe more precise, **restructuring** [1]

Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system’s external behavior (functionality and semantics).

2. Program comprehension/Program Understanding/Reverse Engineering

Reverse engineering is the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at higher level of abstraction

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶

Go Back
Full Screen
Close

3. Refactoring: Some techniques (HaRe)

- Structural refactorings: Generalisation
- Renaming a definition
- Changing the scope of a definition
- Adding/Removing an argument

Weakness:

- Techniques applied in isolated and intuitive way

We are looking for a systematic refactoring strategy

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶
Page 5 of 22
Go Back
Full Screen
Close

4. Our approach for refactoring

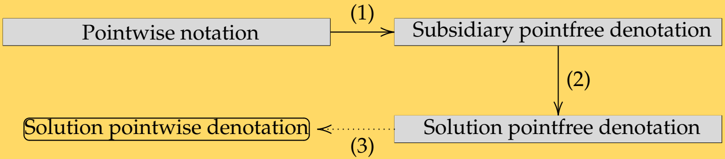

Figure 1: The reverse program calculation process

- Phase (1): Source-to-source transformations
  - removing parameter accumulation
- Phase (2): Formal refactoring
  - point-free calculus
  - pattern driven
- Phase (3): Reimplementation
  - Haskell
  - VDM-SL

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶
Page 6 of 22
Go Back
Full Screen
Close

5. Source-to-source transformations

During our experimentation we found that one of the most useful source-to-source transformation is *removing parameter accumulation*. We show an example from [3].

```
reset0t([],test0,(possum,negsum)) = ([],test0,(possum,negsum))
reset0t(n:l,test0,(possum,negsum)) =
 reset0t(l,test0,set_sum(n,test0,(possum,negsum)))

set_sum(n,test0,(ps,ns)) =
   if n==0 and test0 then
      if ps>ns then
         (0,ns)
      else
         (ps,0)
   else
     (ps,ns)
```

Figure 2: Program example with two accumulation parameters

```
reset0tt([],test0)  = (0,0)
reset0tt(n:l,test0) = set_sum(n,test0,reset0tt(l,test0))
```

Figure 3: Program after removing accumulators

So, figure 3 may be handled by mutual recursion law, etc.

Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶
Page 8 of 22
Go Back
Full Screen
Close

6. Formal Refactoring

Many laws and properties applied during the refactoring phase are taken from the *point-free calculus*. But, here, for not to be tedious, we show some of these related to *co-product* and *exponential* only.

6.1. Co-product

To combine functions as $f : C \longleftarrow A$ and $g : C \longleftarrow B$, we need *injectors*

$$\begin{tikzcd} A \arrow[r, "i_1"] & A + B & B \arrow[l, "i_2"'] \end{tikzcd}$$

defined as

$$\begin{aligned} i_1\ a &= (t_1, a) \\ i_2\ b &= (t_2, b) \end{aligned} \tag{1}$$

Therefore, we combine f and g as follow

$$\begin{array}{lcl} [f,g] & : & A+B \longrightarrow C \\ [f,g]\,x & \overset{def}{=} & \begin{cases} x = i_1\,a & \Rightarrow \quad f\,a \\ x = i_2\,b & \Rightarrow \quad g\,b \end{cases} \end{array} \tag{2}$$

operator named *either*. By mean of this, we define the *co-product of functions*

$$f + g \overset{def}{=} [i_1 \cdot f, i_2 \cdot g] \tag{3}$$

Properties

- Cancellation

$$\begin{array}{lcl} [f,g] \cdot i_1 & = & f \\ [f,g] \cdot i_2 & = & g \end{array} \tag{4}$$

- Reflection

$$[i_1, i_2] = id_{A+B} \tag{5}$$

- Fusion

$$f \cdot [g,h] = [f \cdot g, f \cdot h] \tag{6}$$

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶
Page 10 of 22
Go Back
Full Screen
Close

- Absorption

$$[f, g] \cdot (i + j) = [f \cdot i, g \cdot j] \tag{7}$$

- Functor

$$(f \cdot g) + (i \cdot j) = (f + i) \cdot (g + j) \tag{8}$$

- Functor-id

$$id_A + id_B = id_{A+B} \tag{9}$$

6.2. Exponential

To combine functions $f : C \times A \longrightarrow B$ and $g : A \longrightarrow B \ldots$ we "frozen" the C argument

$$\begin{aligned} f_c &: A \longrightarrow B \\ f_c a &\overset{def}{=} f(c, a) \end{aligned}$$

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶

Go Back
Full Screen
Close

thus, we have fc is a value of type B, but $f_c \in B^A$ is a function!

$$B^A \overset{def}{=} \{g | g : A \longrightarrow B\} \tag{10}$$

From here, we design the *apply* operator

$$\begin{aligned} ap &: B^A \times A \longrightarrow B \\ ap(f, a) &\overset{def}{=} fa \end{aligned}$$

- Cancellation

$$\begin{tikzcd} B^A \times A \arrow[r, "ap"] & B \\ C \times A \arrow[u, "\overline{f} \times id"] \arrow[ur, "f"'] & \end{tikzcd} \qquad f = ap \cdot (\overline{f} \times id) \tag{11}$$

- Reflexion

$$\begin{tikzcd} B^A \times A \arrow[r, "ap"] & B \\ B^A \times A \arrow[u, "id_{B^A} \times id"] \arrow[ur, "ap"'] & \end{tikzcd} \qquad \overline{ap} = id_{B^A} \tag{12}$$

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶
Page 12 of 22
Go Back
Full Screen
Close

- Fusion

$$\begin{tikzcd}
B^A \times A \arrow[r, "ap"] & B \\
C \times A \arrow[u, "\overline{g}\times id"] \arrow[ur, "g"'] & \\
D \times A \arrow[u, "f\times id"] \arrow[uur, "g\cdot(f\times id)"'] &
\end{tikzcd}
\qquad \overline{g \cdot (f \times id)} = \overline{g} \cdot f \tag{13}$$

- Absorption

$$\begin{tikzcd}
D^A \times A \arrow[r, "ap"] & D \\
B^A \times A \arrow[u, "f^A\times id"] \arrow[r, "ap"] & B \arrow[u, "f"'] \\
C \times A \arrow[u, "\overline{g}\times id"] \arrow[ur, "g"'] &
\end{tikzcd}
\qquad \overline{f \cdot g} = f^A \cdot \overline{g} \tag{14}$$

where we use another functional combinator

$$(f^A)g \overset{def}{=} f \cdot g \tag{15}$$

- Functor

$$(g \cdot h) = g^A \cdot h^A \tag{16}$$

- Functor-id

$$id^A = id \tag{17}$$

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶
Page 14 of 22
Go Back
Full Screen
Close

7. (Pattern-driven) Formal Refactoring

The calculated patterns lead the transformational process.

- For list

$$\begin{aligned} \langle\!|h|\!\rangle Nil &= h_1 \\ \langle\!|h|\!\rangle Cons &= h_2^* \cdot \tau_{A,L} \cdot (id \times \langle\!|h|\!\rangle) \end{aligned} \tag{18}$$

- For Binary tree

$$\begin{aligned} (\!|h|\!) Leaf\ a &= h_1\ a \\ (\!|h|\!) Join\ t_1\ t_2 &= h_2^* \cdot \psi \cdot ((\!|h|\!) \times (\!|h|\!))(t_1, t_2) \end{aligned} \tag{19}$$

8. An example

The example use a list datatype involving the monad **State**. But more experiments we have carried out on binary tree, for example, and handling other side effects.

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶
Page 16 of 22
Go Back
Full Screen
Close

A commutative diagram is often used as a graphical tool to get a quick view of the function we are interested in.

$$\begin{tikzcd}
L \arrow[d, "\llparenthesis sms \rrparenthesis"'] & & 1 + Int \times L \arrow[ll, "in"'] \arrow[d, "id+id\times\llparenthesis sms \rrparenthesis"] \\
(Int \times S)^S & ((1 + Int \times Int) \times S)^S \arrow[l, "\overline{(h\times id_S)}^{*}"] & 1 + Int \times (Int \times S)^S \arrow[l, "\delta^L_{Int}"]
\end{tikzcd} \tag{20}$$

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶

Go Back
Full Screen
Close

The pattern-driven calculational/transformational process applying, among others, properties and laws from the point-free calculus.

$$
\begin{array}{ll}
 & (\!|sms|\!) \cdot in \\
= & \{(20)\} \\
 & \overline{h \times id} \bullet \delta^{L}_{Int} \cdot (id + id \times (\!|sms|\!)) \\
= & \{\text{distribution law definition}\} \\
 & \overline{h \times id} \bullet [\widehat{i_1}, \widehat{i_2} \bullet \tau_{Int,L}] \cdot (id + id \times (\!|sms|\!)) \\
= & \{\text{kleisli composition definition}\} \\
 & (\overline{h \times id})^{*} \cdot [\widehat{i_1}, \widehat{i_2} \bullet \tau_{Int,L}] \cdot (id + id \times (\!|sms|\!)) \\
= & \{(6)\} \\
 & [\overline{(h \times id)}^{*} \cdot \widehat{i_1}, \overline{(h \times id)}^{*} \cdot \widehat{i_2} \bullet \tau_{Int,L}] \cdot \\
 & (id + id \times (\!|sms|\!)) \\
= & \{\text{lifting functor definition}\} \\
 & [\overline{(h \times id)}^{*} \cdot (unit \cdot i_1), \overline{(h \times id)}^{*} \cdot (unit \cdot i_2) \bullet \tau_{Int,L}] \cdot \\
 & (id + id \times (\!|sms|\!)) \\
= & \{\text{associativity and second kleisli triple property}\}
\end{array}
$$

$$[\overline{(h \times id)} \cdot i_1, \overline{(h \times id)} \cdot i_2 \bullet \tau_{Int,L}] \cdot$$
$$(id + id \times \llparenthesis sms \rrparenthesis)$$

$= \{(14)$ in reverse$\}$

$$[\overline{(h \times id) \cdot (i_1 \times id)}, \overline{(h \times id) \cdot (i_2 \times id)} \bullet \tau_{Int,L}] \cdot$$
$$(id + id \times \llparenthesis sms \rrparenthesis)$$

$= \{$"bi-distribution" of $\times$ with respect to composition in reverse$\}$

$$[\overline{(h \cdot i_1) \times (id \cdot id)}, \overline{(h \cdot i_2) \times (id \cdot id)} \bullet \tau_{Int,L}] \cdot$$
$$(id + id \times \llparenthesis sms \rrparenthesis)$$

$= \{$identity and h definition$\}$

$$[\overline{([\underline{0}, +] \cdot i_1) \times id}, (\overline{([\underline{0}, +] \cdot i_2) \times id}) \bullet \tau_{Int,L}] \cdot$$
$$(id + id \times \llparenthesis sms \rrparenthesis)$$

$= \{(4)\}$

$$[\overline{\underline{0} \times id}, (\overline{+ \times id}) \bullet \tau_{Int,L}] \cdot (id + id \times \llparenthesis sms \rrparenthesis)$$

$= \{(7)$ and kleisli composition definition$\}$

$$[\overline{\underline{0} \times id}, (\overline{+ \times id})^{*} \cdot \tau_{Int,L} \cdot (id \times \llparenthesis sms \rrparenthesis)]$$

Since $in = [Nil, Cons]$ we can conclude that

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶

Go Back
Full Screen
Close

```
sms l = \s -> mfoldL(return 0, \x y -> do {c <- tick;
                                           return(x+y)}) l
```

Figure 7: sms function refactored by mfoldL operator

```
nmfoldL :: Monad m => (m a, b -> m a -> m a) -> [b] -> m a
nmfoldL (h1,h2) = mfl
 where mfl []      = h1
       mfl (a:as) = h2 (a) (mfl as)
```

Figure 8: mfold operator for lists without distribution law

$$\begin{aligned} (\!|sms|\!) Nil &= \overline{\underline{0} \times id_S} \\ (\!|sms|\!)(Cons) &= \overline{(+ \times id_S)}^* \cdot \tau_{Int,L} \cdot (id \times (\!|sms|\!)) \end{aligned} \tag{21}$$

Matching (21) and (18) . . .

8.1. An alternative refactoring

But, in this case, we can show another way to refactor leaded by other pattern.

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶
Page 20 of 22
Go Back
Full Screen
Close

```
sms = \s -> nmfoldL(return 0,\e r -> do {c <- tick;
                                         x <- r; return(e+x)})
```

Figure 9: sms refactored by nmfoldL operator

9. Future directions

- To analyze more complex cases involving monad transformers
- To apply more abstract patterns as mentioned by [2]
- Patterns for specific domain problems?
- Funtional setting for reengineering imperative code?

10. Conclusions

- The refactoring process is pattern driven
- We can calculate specification
- The patterns are calculated . . . not designed

Refactoring
Program . . .
Refactoring: . . .
Our approach . . .
Source-to- . . .
Formal . . .
(Pattern- . . .
An example
Future directions
Conclusions
Home Page
Print
Title Page
◀◀ ▶▶
◀ ▶

Go Back
Full Screen
Close

References

[1] E.J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[2] Jeremy Gibbons. Design patters as higher-order datatype generic programs. In *Workshop on Generic Programming*, Portland, Oregon, USA, September. ACM SIGPLAN.

[3] G. Villavicencio. Reverse program calculation by conditioned slicing. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pages 368–378, Benevento, Italy, March 2003. IEEE CS Press, California, USA.