Bananas, Dark Worlds, and AspectH
(Group 5 Discussion)

Silvia Breu, Nuno Rodrigues, Marc Schlickling

Abstract

This report summarises our idea of code clone detection in Haskell code and refactorings based on identified clones as it evolved in our group-of-three discussion.

1 Motivation

Looking at the code example below, we observe a certain level of *redundancy*, in particular structural redundancy, such as $\boxed{\text{pattern matching}}$, $\underline{\underline{\text{if-then-else}}}$-constructs, and $\boxed{\boxed{\text{recursion}}}$.

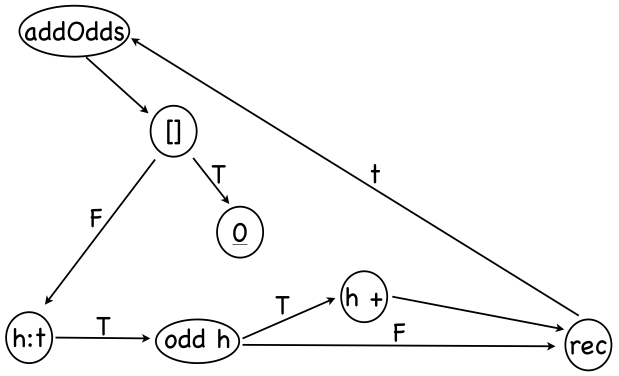
```
addOdds :: Integral a => [a] -> a
addOdds []    = 0
addOdds (h:t) = if odd h then h + (addOdds t) else addOdds t

remNeg :: (Ord a, Num a) => [a] -> [a]
remNeg []    = []
remNeg (h:t) = if h >= 0 then h : (remNeg t) else remNeg t

getReflex :: Eq a => [(a,a)] -> [(a,a)]
getReflex []         = []
getReflex ((x, y):t) = if (x == y) then (x, y) : (getReflex t)
                                   else getReflex t
```

But how can we automate the process to find such recurring and somewhat redundant patterns? Should we investigate the source code directly, or is it better to find an intermediate representation, that abstracts the source code to a level where these patterns can be identified more easily, detached from the actual name of variables etc.?

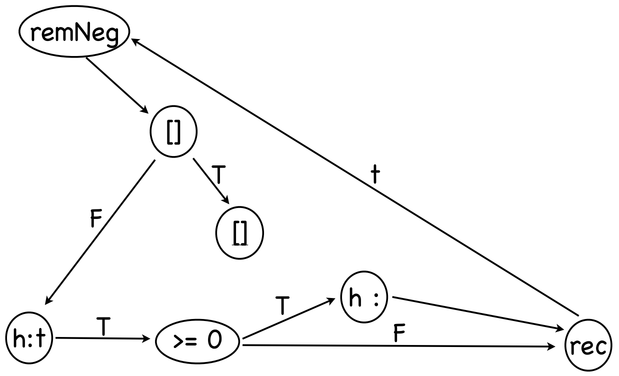
2 Functional Control Graph and Functional Control Tree

As it would be very inconvenient to identify such redundancy by hand in the code, especially these structural similarities, we propose a special structure called *functional control graph.* Its entry node carries the name of the function, its other nodes each represent an expression from the Haskell code of that function. The edges can but may not necessarily be labeled: T (true)/F (false) for a step that requires the previous expression to be true/false, and parameter names in case any parameters are passed along.

Considering the Haskell code example above, we get the following functional control graph for the function `addOdds` which takes a list of integers and returns the sum of all odd integers in that list:

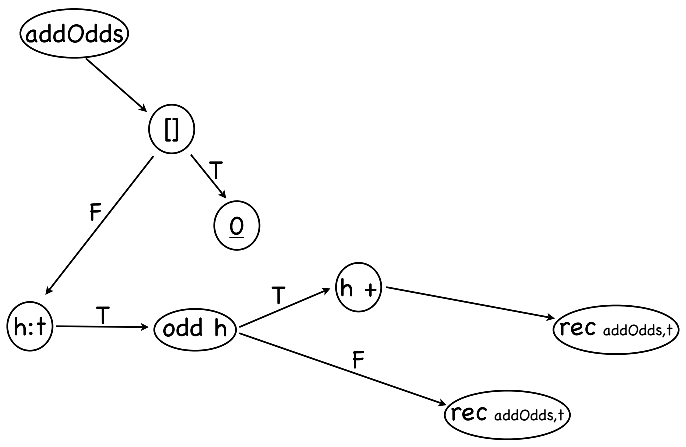


For the function `remNeg` which takes a list of integers, removes all negative numbers within, and returns the adjusted list, the functional control graph looks as follows:



Comparing the two graphes, we can recognise a common structure between them, that represent again the (structural) redundancy in the code. To find code clones, e.g., in C code, there already exist several approaches. [Kri01] identifies similar subgraphs in attributed directed graphs, namely in program dependence graphs. This is done by mapping identical nodes and identically labeled edges. Another approach was presented by [BYM+98] and works on abstract syntax trees. There, all nodes in the level before the last get assigned a hash value which are then propagated up the tree. Thus, the same types of nodes (e.g., variable nodes, constant nodes, ++ operator) get the same hash value and code clones are uncovered by identical hash values. Furthermore, there exist many algorithms in graph theory to identify isomorphic subgraphs, e.g., by [Bac01].

However, often NP-hard problems on graphs, such as identifying isomorphic subgraphs, become easier on special kinds of graphs such as trees. Thus, we re-model our functional control graph slightly to a *functional control tree*. For each incoming edge in a node of the control graph that represents a recursive call to another function, we build its own destination node. Instead of then adding an edge from each new node to the entry node of the recursively called function, we annotate the recursive call node with the name of the recursively called function, and—if necessary—any arguments that are passed on. The previously existing edge from the recursive call node to the entry node of the recursively called function is deleted. This results for the function `addOdds` in the following tree:



3 Functional Patterns

By using the above graph and tree representation of functional programs and by combining it with other well known clone detection techniques, we were able to discover some structural basis that all programs from our example shared. Still we have to investigate more about what are this underlying commonalities and how may we take advantage of them whenever they appear in code.

The functional control graphs we are using capture the way data flows inside functions. Since one of the main characteristics that decides how data flows in functional programs is recursion, maybe one of the main common functional aspects we are identifying is recursion patterns. Indeed, if we look at the recursion patterns used in our examples, we realise that they all use toail recursion. This tail recursion is mainly responsible for the overall similarities in the layout that all the graphs share.

Recursion patterns have been well studied in the past, for which [BM97, MFP91] are good examples of. In fact, some of the recursion patterns presented in [BM97] fit very well in the patterns we have previously identified.

For the `addOdds`, `remNeg`, and `getReflex` functions, one can find out which recursion pattern fits better by noticing that all edges returning to the entry node (the recursive edges), came from the $h : t$ node and that they only re-utilise the tail t of the input list.

This is a very common strategy in functional programming and can be captured by a recursion pattern presented in [BM97] called catamorphism. The following diagram shows how the pattern works and that the only thing it needs to derive working solutions is a definition for the *gene* f. Now the explicit recursion is hidden from the gene f, which just has to calculate the intended result B from the case where the list is empty (represented by the 1 in the notation) and from the case where it has to combine an element of the list (A) and a list already calculated with the desired result (B).

$$\begin{tikzcd}
\mathsf{F}A \arrow[r, "out"] \arrow[d, "(\!|f|\!)_F"'] & 1 + A \times \mathsf{F}A \arrow[d, "id+id\times(\!|f|\!)_F"] \\
B & 1 + A \times B \arrow[l, "f"]
\end{tikzcd}$$

Nevertheless, the above formalism only captures part of the similarities that we found in the functional control graphs. In fact, we are only treating the redundancy referring to the recursive edges, but there can be other kinds of similarities in the graphs. For instance, in the presented example functions, every graph has a control node based on a predicate, in the case of non empty lists, which decides how the recursive call should be made. Such similarities can also be captured and formalised, augmenting the above diagram to the following one.

$$\begin{tikzcd}
\mathsf{F}A \arrow[rr, "out"] \arrow[d, "\llparenthesis f \rrparenthesis_F"'] & & 1 + A \times \mathsf{F}A \arrow[d, "id+id\times\llparenthesis f \rrparenthesis_F"] \\
B & & 1 + A \times B \arrow[ll, "f"] \arrow[dl, "id+(\pi_2\cdot(p\times id))?"] \\
& 1 + ((A \times B) + (A \times B)) \arrow[ul, "\underline{c}+[g,\pi_2]"] &
\end{tikzcd}$$

By encapsulating the test p in the pattern, our gene f definition is even more specific and captures more of the similarities previously identified by the graphs analysis. With the newly discovered pattern definition, we can now define all functions just by filling in the missing parts p, $\underline{c}$ and g in the following definition of the catamorphism gene.

$$f = (\underline{c} + [g, \pi_2]) \cdot (id + (\pi_2 \cdot (p \times id))?)$$

p is the desired test predicate, $\underline{c}$ is the constante being applied when the input is an empty list, and g the function that combines an element of the list with the result of applying the defining function to the rest of the list whenever the predicate succeeds.

Besides having isolated the redundancy in function definitions, the above method also delivers a formal definition over the functional calculus presented in [BM97]. This can then be used to formally calculate properties over programs, or to refine the analysed programs.

4 Haskell Refactoring

For the refactoring of the found pattern, however, we do not need to introduce a new aspect-oriented extension such as AspectH, or whatever called. In the world of functional programming, Haskell is that powerful that we get it for free. Applying the previously identified recursion patterns to the code of our functions `addOdds`, `remNeg`, and `getReflex` we get the following refactored pattern `pat1` and the three "new" functions `addOdds'`, `remNeg'`, and `getReflex'`. The usage of the refactored pattern in those three functions is marked with a box ($\boxed{\boxed{\texttt{pat1}}}$).

```
pat1 :: (a -> Bool) -> b -> (a -> b -> b) -> [a] -> b
pat1 p n f []    = n
pat1 p n f (h:t) = if p h then f h (pat1 p n f t)  else pat1 p n f t

addOdds' :: Integral a => [a] -> a
addOdds' = pat1 odd 0 (+)

remNeg' :: (Ord a, Num a) => [a] -> [a]
remNeg' = pat1 (>= 0) [] (:)

getReflex' :: Eq a => [(a,a)] -> [(a,a)]
getReflex' = pat1 (uncurry (==)) [] (:)
```

5 Conclusions and Future Work

We have introduced a new intermediate data representation for functional programs, the functional control graph. Furthermore, we suggested the application of some well-known code clone detection techniques in order to identify recurring patterns in functional code. Some of these patterns can then be formalised in the functional calculus, giving a string basis and soundness to the program transformations that such patterns suggest.

By the analysis of the treated example, as well as some other more complex ones, we strongly believe that the recursion pattern identification can entirely be based on the functional control graph of the program. Even more, we believe that there can be other types of graph/tree similarities that can automatically be discovered and treated accordingly, similar to the predicate pattern found in the presented example.

The HaRe[1] project is related to the results of our preliminary work, where we target refactoring of functional programs. Nevertheless, their approach is quite different form ours. First, they do not make any use of explicit intermediate representations of the programs besides an annotated abstract syntax tree. Second, their approach is more focused on pure refactorings applicable to concrete places in the code while our approach focusses more on a general view of all the entities in a functional program and tries to identify common patterns in such definitions. These are then refactored to reduce the quantity and redundancy in the source code.

Future work will pass by the classification of common recurring (redundant) patterns, their respective transformations, and theoretic justification. The selection of the most relevant graph and tree clone analysis algorithms applied to our functional control graph will also be subject of future research.

[1]http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html

References

[Bac01] Sabine Bachl. *Isomorphe Subgraphen und deren Anwendung beim Zeichnen von Graphen.* PhD thesis, University of Passau, 2001.

[BM97] R. Bird and O. Moor. *The Algebra of Programming.* Series in Computer Science. 1997.

[BYM+98] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Lect. Notes Comp. Sci. (523), 1991.

A Second Code Example

Original version

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

preorder :: (Ord a, Num a) => Tree a -> [a]
preorder (Leaf a)            = if a > 0 then [a] else []
preorder (Node b left right) = if (b > 0)
                                  then b : (preorder left) ++ (preorder right)
                                  else (preorder left) ++ (preorder right)

evenTSum :: Integral a => Tree a -> a
evenTSum (Leaf a)            = if (even a) then a else 0
evenTSum (Node b left right) = if (even b)
                                  then b + (evenTSum left) + (evenTSum right)
                                  else (evenTSum left) + (evenTSum right)

minPos :: (Ord a, Num a) => Tree a -> a
minPos (Leaf a)            = if a > 0 then  a else -1
minPos (Node b left right) = if b > 0
                                then min (min b (minPos left)) (minPos right)
                                else min (minPos left) (minPos right)
```

Refactored version

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

pat2 :: (a -> Bool) -> b -> (a -> b) -> (a -> b -> b -> b) -> (b -> b -> b) ->
         Tree a -> b
pat2 p n g f h (Leaf a)            = if p a then g a else n
pat2 p n g f h (Node b left right) = if p b
                                        then f b (pat2 p n g f h left)
                                                 (pat2 p n g f h right)
                                        else h (pat2 p n g f h left)
                                               (pat2 p n g f h right)

preorder' :: (Ord a, Num a) => Tree a -> [a]
preorder' = pat2 (> 0) [] (\x -> [x]) (\x y z -> x : y ++ z) (++)

evenTSum' :: Integral a => Tree a -> a
evenTSum' = pat2 odd 0 id (\x y z -> x + y + z) (+)

minPos' :: (Ord a, Num a) => Tree a -> a
minPos' = pat2 (> 0) (-1) id (\x y z -> x `min` y `min` z) min
```