Identifying Clones in Functional Programs for Refactoring

Nuno Rodrigues[1,2] and João L. Vilaça[1,3]

[1] DIGARC
Polytechnic Institute of Cávado and Ave
4750-810 Barcelos, Portugal
[2] DI-CCTC
University of Minho
4710-057 Braga, Portugal
[3] Life and Health Sciences Research Institute
School of Health Sciences, University of Minho
4710-057 Braga, Portugal

Abstract. Clone detection is well established for imperative programs. It works mostly on the statement level and therefore is ill-suited for functional programs, whose main constituents are expressions and types. In this paper we introduce clone detection for functional programs using a new intermediate program representation, dubbed Functional Control Tree. We extend clone detection to the identification of non-trivial functional program clones based on the recursion patterns from the so-called Bird-Meertens formalism.

1 Introduction

Each software system has a considerable part of duplicated code, i.e. code clones, which can be exact or modified copies of code bits. These clones are the result of copy-paste-modify actions taken by developers, usually employed when developers want to re-use previously implemented functionality, and many times, because of short implementation times imposed by industry demands. Consequently, no software industry branch is immune from code clone, and it is more prevalent in areas where large software projects are developed.

Although code clone can speed up development on short term, which is a fundamental aspect for companies, on long term, code clone implications can be very negative. In fact, whether the code was duplicated on purpose or whether the duplicates emerged by chance—they impose increased difficulties and costs in maintenance, adaptability, and extendability. This has given rise to research in *source code refactoring*, including techniques to support that process, in particular code clone detection approaches. Refactoring is a solid software transformation technique with given proofs over the popular object-oriented (OO) and imperative paradigm to enhance modularity and in turn ease maintenance of software.

In the same manner as the OO paradigm benefits from refactoring, it is our believe that functional programs have also something to profit from similar transformation principles. In particular, the encapsulation of, e.g., recursive patterns such as catamorphisms and successive function invocations provide reusable higher-order functions. This paper presents our idea to identify structural and functional patterns in Haskell programs in order to foster refactoring as well as re-use of identified and refactored patterns.

Contributions. We present an approach to automatically detect clones and recursive patterns in functional programs. It builds upon a new intermediate program representation, called functional control tree (Sections 3 and 4). The identified recursive patterns are formalised by the functional calculus of Bird-Meertens (Section 5).

2 Functional Control Tree

The basic idea presented before, motivates that an intermediate program representation as graph seems sensible in order to facilitate *automatic* pattern detection and refactoring. However, functional programs may have more and very different kinds of clone/refactoring scenarios as the simple one that we just discussed. At least, we want to consider the following five scenarios.

- *Re-defining* a function if it is used in another function definition seems trivial but can be used later on with more complex scenarios.
- *Inlining* of a function B in the calling function A if B has no recursive calls. As it is a de-modularisation, it is only a very simple refactoring case that may not be sensible on its own.
- *Pattern matching re-use* takes parts that are used/shared by two or more functions and encapsulates the pattern matching schema in a third function. The only requirement is that the pattern matching has to agree on the outer data type.
- *Successive Invocations* that are identified, i.e., one function successively calls another function, can be refactored, e.g., by applying `foldr` to the list of arguments on the successive invocations.
- Most importantly in functional programming, *recursions*—such as catamorphisms—for a particular data type can be encapsulated into a single function.

Considering these situations, we define the previously coarse-grained FCT now refined. In particular, we distinguish the different kinds of nodes that we use to build the function control tree. The formal definition is as follows:

Definition 21 (Functional Control Tree) *A Functional Control Tree (FCT) is a directed graph $G = \langle \mathcal{V}, \mathcal{E} \rangle$ with $\mathcal{V}$ set of vertices denoting program code expressions, and $\mathcal{E}$ set of edges representing control dependencies between the expressions.*

Each FCT has its dedicated entry node that carries the name of the function that is represented by that particular functional control tree. The edges may be labelled true or false, stating whether the previous conditional expression had to evaluate to true or false in order to follow this path.

In order to express all properties of a function appropriately for the later described clone detection algorithms, we need to distinguish five kinds of vertices. In detail, the vertices in our functional control tree are of either of the following kinds:

constant nodes that map constant expressions,
function nodes that constitute calls to functions,
recursive nodes that represent recursion,
conditional nodes that form conditional expressions, or
pattern nodes that represent pattern-matching clauses.

As calls to functions as well as recursive function calls will carry one or more arguments, it is important to handle these arguments correctly. Therefore, function and recursive nodes have a list attached in which all needed arguments are stored in the order as they have to be supplied. In the FCT itself, the list order is mapped by the natural left-to-right order in the tree.

3 Functional Clone Detection

As a prove of concept we have developed a simple clone detector based on FCTs. Keeping faithful to the functional paradigm, the clone detector is implemented in Haskell [1] and makes heavy use of some functional programming machinery, like high order functions, lazy evaluation and polymorphism.

3.1 Implementing the FCT

Capturing the previously formalised notion of an FCT, we have the following Haskell data type:

```
data FCT a = Constant a
           | Function a [FCT a]
           | Recursive [FCT a]
           | If (FCT a) (FCT a) (FCT a)
           | PM [PatternClause a]
           deriving (Eq, Show)

type PatternClause a = ([Pattern a], FCT a)

data Pattern a = PConstant a
               | PFunction a [Pattern a]
               deriving (Eq, Show)
```

where an `FCT` defines a tree containing several kinds of nodes. Each node kind represents a different entity from the source code and each constructor is self-explanatory to understand the kind of entities it represents. However, the last node type represent not so obvious code entities, which are pattern-matching clauses. Thus, a `PM` node has a list of pairs where each one of these pairs represents a pattern-matching clause. Each of these pairs is constituted by a list of patterns, which represent all the curry patterns of the clause, and an `FCT` which represents

the function definition associated to that clause. Note that, due to pattern-matching evaluation limitations, pattern nodes have fewer types available when compared to regular nodes.

Since we deliberately adopted a more general specification of the FCT data type, in order to keep it as simple as possible, it can give rise to the existence of valid `FCT` instances that do not respect the previous FCT formal definition. Thus, it is necessary to have some data type invariants in order to avoid invalid FCT instances. Therefore, to ease these invariants' understanding, one will present them in an informal way, although the clone detector implementation has concrete Haskell functions constraining FCT instances. Accordingly, an FCT must respect the following invariants:

- Every node of type `PM` must have at least one pattern clause.
- Every node of type `PFunction` must have at least. one pattern

In order to illustrate the use of the FCT data type, we present the FCT instance for function `mySum`:

```
mySum :: (Num a) => [a] -> a
mySum []    = 0
mySum (h:t) = h + (mySum t)
```

```
mySumFct =
   PM [ ([PConstant "[]"], Constant "0"),
        ([PFunction ":" [PConstant "h",
                         PConstant "t"]],
         Function "+" [Constant "h",
                       Recursive [Constant "t"]])]
```

3.2 Limitations of the Data Type FCT

By adopting such a data type definition to capture FCT instances, we are restricting ourselves to a subset of the entire Haskell language. This way, we do not have a direct way of capturing `where` and `let` clauses nor the monadic `do` notation. These limitations may seem to restrictive, but a simple refactoring reveals that this is not the case. In fact, most `where` and `let` clauses serve only to better organise the source code in a more comprehensive manner. Thus, we can eliminate these expressions by substituting in the core of the function the new variables they introduce by the expression they capture. Still, `let` and `where` expressions that perform more complex pattern-matching, can not be eliminated in this way, thus representing truly cases with no direct instance in our FCT data type which we do not consider. Concerning the monadic `do` notation, we can easily capture them in our FCT data type by using the semantic of the notation which is in fact just a syntax convenience to the appliance of the monadic combinatorial functions `(>>)` and `(>>=)`.

3.3 Clone Detection Algorithms

Given the above FCT data type implementation, we still have to define suitable clone detection strategies. Nevertheless, in order to do so, we first need a precise notion of what do we mean by a functional code clone. In fact, this is a pertinent definition that will drive all the latter analysis algorithms. We consider

two functional expressions as being clones, if they can be captured by a third functional expression that, when used in the place of the first two expressions, does not change the semantic of the initial expressions.

In particular this means that we may have syntactically fairly different expressions that by our definition are still regarded as clones. This approach to code clones, also gives rise to the appearance of a vast number of clones which can be categorised according to some specific characteristics they expose. Nevertheless, such an issue is beyond the scope of this paper.

As it would be expected, all clone discovery algorithms we present are based on the programs corresponding FCT instances. The approach we take, consists first of the generation of the FCT for each function of the analysing program. Given this FCTs we proceed by calculating every sub-tree of the FCTs previously calculated. Since a complete sub-tree calculation can lead to a vast number of cases to compare, we truncate it by using a threshold consisting of a minimum number of nodes in each sub-tree. The next step consists of hashing each sub-trees in order to create buckets of similar sub-trees. This hashing depends on the clone detection algorithm that one will use in the following phase. The final phase consists of comparing each sub-tree inside a particular bucket, using one of the clone detection algorithms.

Exact Clone Detection. Concerning the clone detection algorithms, we start by analysing the simplest case of clone detection, which is the detection of exact clones. We then continue in an incremental way by analysing more complex non-trivial clones. Every strategy presented here is implemented in the clone detector prototype as FCTs comparison function(s).

The first strategy, and the simplest one, consists on finding exact clones, i.e., exact syntactic code copies except for the use of white characters. In this case, the Haskell equality operator `==` suffices to give an implementation such that all FCT data types derive from the `Eq` class. Since we are using `String` as the FCT data type parameter and all FCT data types derive from `Eq`, we already have this exact comparison for free.

Bringing some light into the categorisation of the code clones we are targeting in this case, we categorise this clones as *higher order clones*. The justification behind this name comes from the fact that the refactoring process associated to the elimination of these clones makes heavy use of higher order functions.

Exact Clone Detection up to Constants. The second clone detection case is given by an exact clone detection up to constant nodes, i.e., we are looking for exact clones, like in the previous case, except that now we do not force constant nodes to be identified by the same identifier in both expressions. In practice, this translates to finding exact functional expressions except for the name of pattern-matching variables and their following uses in the core of the function definition. Such code clones can be found by using the former exact clone detection function `(==)` after applying a colouring constant node function to both comparing expressions. This constant colouring node function attributes a different colour to every constant node, by the order of appearance in the tree, followed by a

substitution of every occurrence of that variable in the core of the function by the same colour.

At the implementation level this is accomplished by function `renameConsts` which makes use of function `rnConsts` that substitutes constant nodes by `ID` values identified by integers.

As suitable hashing criterion for this algorithm, we use the number of constant nodes in a tree. Nevertheless, for large programs, other kinds of hashing functions may be needed and even tuned according to the context of the program.

Exact Clone Detection up to Every Node Kind. Building up on top of the previous approach, one finds the discovery of code clones up to every kind of node. Translating this to our FCT terminology, one is looking for FCTs with the same shape and the same node colours, only this time instead of using a colouring function that only colours constant nodes, we will use a colouring function that colours every node type. After having applied this colouring function to both comparing expressions, the clone detection subsumes to an exact tree equality comparison with the `(==)` function.

Code Clone Detection Tuning. A fourth kind of clone detection, which is actually more a specialisation to all previous algorithms, consists of taking into account the fact that certain operators are commutative. Thus, this approach takes as extra input a list of commutative functions and discovers code clones that may commute the arguments of such functions. In particular, we apply this principle to all other approaches.

Sub-tree Analysis. As for the sub-tree generation, identified earlier as the second phase of the clone detection process, it is particularly useful in the way that it permits to find sub-expressions inside a function that were already declared by another function or another function sub-expression. At the implementation level, such a specialisation is mainly accomplished due to the following function which calculates every sub-tree of a given FCT with a minimum number of nodes `i`. Notice that we exclude from the sub-tree list the sub patterns of a given pattern-matching clause, since they do not point out clones. Nevertheless, the FCT associated to each pattern must be decomposed in the several sub-trees, which in fact may point out possible cases of code clones.

```
fctSubTrees :: (Ord a) => Int -> FCT a -> [FCT a]
fctSubTrees i n@(Constant a) =
    if i <= 1 then [n] else []
fctSubTrees i n@(Function a ts) =
    if fctSize n >= i
        then n : (concat .
                  map (fctSubTrees i) $ ts)
        else []
fctSubTrees i n@(Recursive ts) =
    if fctSize n >= i
        then n : (concat .
                  map (fctSubTrees i) $ ts)
        else []
fctSubTrees i n@(If tt t1 t2) =
    if fctSize n >= i
    then n : (fctSubTrees i tt ++
              fctSubTrees i t1 ++
              fctSubTrees i t2)
    else []
fctSubTrees i (PM []) = []
fctSubTrees i n@(PM ((pt, t):pts)) =
    if fctSize n >= i
    then n : (fctSubTrees i t ++
              fctSubTrees i (PM pts))
     else []
```

4 Identifying Functional Patterns

Using the 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 a large amountof function programs share. Still, we have to investigate more what these underlying commonalities are, and how we may take advantage of them whenever they appear in the code.

The functional control trees 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, 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 tail recursion. This tail recursion is mainly responsible for the overall similarities in the layout that all the trees share.

Recursion patterns have been well studied in the past, for which [2, 3] are good examples of. In fact, some of the recursion patterns presented in so-called Bird-Meertens formalism [2] fit very well in the patterns we have previously identified.

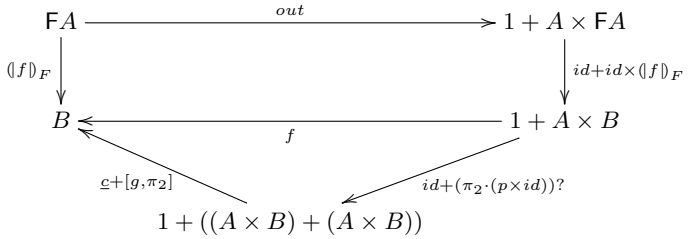
For the `addOdds`, `remNeg`, and `getReflex` functions, one can find out which recursion pattern fits better by noticing that all edges returning to the entry node (the recursive edges), came from the $h : t$ node and that they only re-utilise the tail t of the input list.

This is a very common strategy in functional programming and can be captured by a recursion pattern presented by the Bird-Meertens formalism [2] as a 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 value already calculated with the desired result B (achieved from applying the desired operation to the tail of the list).

$$\begin{array}{ccc} \mathsf{F}A & \xrightarrow{out} & 1 + A \times \mathsf{F}A \\ {\scriptstyle (\!|f|\!)_F}\downarrow & & \downarrow{\scriptstyle id+id\times(\!|f|\!)_F} \\ B & \xleftarrow[f]{} & 1 + A \times B \end{array}$$

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.



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 constant 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 [2]. This can then be used to formally calculate properties over programs, or to refine the analysed programs.

5 Refactoring in Haskell

In order to refactor found patterns, 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.

For the exact clone detection case, the refactoring associated consists of isolating the clone expression in a third independent function and then reusing this last function in place of the previous two expressions. However, the third function definition must keep the pattern matching clauses that refer to any variable used in the sub-expression clones.

For the second clone detection case, which detects exact code clones up to constant nodes, the previous refactoring suffices to eliminate the clones. Notice that this refactoring eliminated the sub-expression variables and created a new function capturing the common pattern-matching variables.

For the third case, the associated refactoring is a bit more elaborate since it as to be sensible to the amount of equal function nodes. Thus, the refactoring begins just like the previous one, but introduces a new pattern-matching parameter for each function node that differs in the FCTs. Once this is accomplished, the clones may be eliminated by calling the new created function using as arguments not just the constant nodes but also the different function nodes.

6 Conclusions and Consequences

We have introduced a new intermediate data representation model for functional programs, the functional control tree. Our FCT definition combines what we believe are the main aspects of functional programs regarding functional program analysis and transformation, i.e., the pattern-matching clauses, the recursive calls, and the control flow influenced by the previous two. In particular, we have shown the application of FCTs in code clone detection, refactoring, and recursive pattern discovery over functional programs.

The discovered recursive patterns identified by our method can be easily formalised in the functional calculus of Bird-Meertens, giving a strong basis and soundness to the programs obtained by the recursive pattern discovery algorithm.

As a prove of concept of the more theoretical ideas over FCTs, we presented the guidelines for a code clone and recursive pattern discovery system implemented in Haskell.

As being what we believe a first incursion in Haskell code clone detection, there are numerous points for future work following what we have presented here. In particular, the development of automatic extractors from Haskell source code to FCT instances would be a very important step in order to test our ideas over real examples. Further research of minimal sets of code entities that positively point out the occurrence of recursive patterns would also be welcome to augment our current recursive pattern cases.

Finally the articulation of our algorithms to fully functional refactorers like HaRe [4] also constitute a primordial step towards our objective of implementing a fully functional code clone detector and refactorer.

References

1. Bird, R.: Functional Programming Using Haskell. Series in Computer Science. ph (1998)
2. Bird, R., Moor, O.: The Algebra of Programming. Series in Computer Science. ph (1997)
3. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In Hughes, J., ed.: Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture, Springer Lect. Notes Comp. Sci. (523) (1991) 124–144
4. HaRe project webpage. : (http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html)