Refactoring Dynamic Languages

Rafael Reia and António Menezes Leitão

Instituto Superior Técnico, Universidade de Lisboa
{rafael.reia,antonio.menezes.leitao}@tecnico.ulisboa.pt

1 Related-Work -Change

A refactoring tool [1] for scheme, implemented in Lisp that uses two forms of information, AST and PDG (program dependence Graph). The AST represents the abstract syntactic structure of the program. While the PDG explicitly represents the key relationship of dependence between operations in the program. The graph vertices's represent program operations and the edges represent the flow of data and control between operations. However the PDG only has information of dependencies of the program. And if there are two semantically unrelated statements they could be placed arbitrarily with respect to each other. Using the AST as the main representation of the program ensures that statements are not arbitrarily reorder. And the PDG is only used as a notation to prove that transformations preserve the meaning and as quick way to retrieve needed dependence information. Contours are used with the PDG providing scope information, non existent in the PDG, to help reason about transformations in the PDG. With these structures it is possible to have a single formalism to reason effectively about flow dependencies and scope structure.

HaRe [2] is a refactoring tool for Haskell that integrates with Emacs and Vim. The HaRe system uses an AST of the program to be refactored in order to reason about the transformations to do. The system has also a token stream in order to preserve the comments and the program layout by keeping information about the source code location and the comments of all tokens. It retrieves scope information from the AST, that allows to have refactoring operations that require binding information of variables. The system also allows the users to design their own refactoring operations using the HaRe API.

2 Architecture

The Refactoring tool uses mainly two sources of information, the def-use-relations and the AST of the program.

The def-use-relations are visually represented in a form of Arrows in Dr-Racket, that information is especially relevant for refactoring operations such as Extrac-Function, Add-Prefix, Organize-Imports, etc.

The AST is represented by the s-expressions that compose the racket program. In Racket everything is a syntax-expression and therefore accessing the list (tree) of the syntax-expressions has all the information that a normal AST provides.

[INSERT IMAGE HERE]

2.1 Def-Use-Relations

DrRacket already uses the def-use-relations in the system that are visually represented by arrows in the GUI. The def-use-relations is computed by the online-compiler that runs in the background However it is only processed when a program is syntactically correct. (e.g. if a program has syntax errors there are no arrows produced in DrRacket)

Def-Use-Relations utility Def-Use-Relations are important in order to produce correct refactoring operations because they can be used to check whether or not there will be a duplicated name or even to compute the arguments of a function to be extracted.

2.2 Syntax Expressions

The Syntax expressions (s-exp) list are already being produced and used by the Racket language and in DrRacket. They represent the program and are computed in order to provide error information to the user. DrRacket already provides functions in order to create s-exp lists and uses some of those functions in the online check syntax and in the check syntax button callback.

[insert images here explaining that]

Syntax Expression tree forms DrRacket provides functions to compute the s-exp list in two different formats. One format is the expanded program, this format is used by the Check Syntax and the online check syntax, and computes the program expanded. The other format is the non-expanded program and computes the program unexpanded.

The expanded program has the macros expanded and the identifier information computed, however it is harder to extract the relevant information when compared with the non expanded program.

For example, the following program is represented in the expanded program, and in the non expanded program.

Listing 1.1. "example"

```
#lang racket
(if (= (+ 1 2) 1)
      #f
      #t)
```

Listing 1.2. "Syntax from Example"

```
#<syntax:3:0 (if (< 1 2) #t #f)>
```

Listing 1.3. "Expanded Syntax from example"

```
#<syntax:1:0 (module anonymous-module racket
  (#%module-begin (module configure-runtime
    (quote #%kernel)#%module-begin
      (#%require racket/runtime-config)
        (#%app configure (quote #f))))
          (#%app call-with-values (lambda ()
            (if (#%app < (quote 1) (quote 2))
              (quote #t) (quote #f))) print-values)))>
```

The expanded program transforms the "and", "or", "when" and "unless" forms into ifs and that makes refactoring operations harder to implement.

Racket adds internal representation information to the expanded-program which for most refactoring operations are not needed.

However, the expanded program has important information regarding the binding information that is not available in the non-expanded form and is rather useful to detect if two identifiers refer to the same binding. In addition to that, the expanded program has a format that is likely to change in the future because Racket is an evolving language and the expanded form is a low level and internal form of representation of the program in Racket.

All those combined make it desirable to use the non expanded form for the refactoring operations as much as possible and use the expanded form only for the necessary operations.

Macros usage could make the refactoring operations incorrect by modifying the program behavior. This is not considered part of the scope of this refactoring tool capabilities. This is not considered part of the scope problem because this refactoring tool is aimed at unexperienced programmers (that had one semester of programming classes) and those type of programmers do not use macros.

If we intended to create a tool that supports macros the non expanded program is insufficient and the expanded program must be used. However there are no guarantees that would be enough to ensure the correctness of such refactoring operations due to the reflection capabilities of Racket.

2.3 Code-walker

The code-walker is used to parse the syntax tree represented by a syntax elements that is a list of s-exp in racket. A syntax element can contain either a symbol, a syntax-pair, a datum (number, boolean or string) or an empty list. While a syntax-pair is a pair containing a syntax object as it first element and a syntax pair, a syntax element or an empty list as the second argument. Each syntax-object has information about the line where they are defined and this information is used by the code-walker to find the correct elements.

Most of the time using the code-walker we are searching for a specific syntax element and we use the location information contained in the syntax-object in order to skip the syntax blocks that are before the syntax element wanted in the first place.

The Code-walker is a core part of the refactoring tool ensuring that the selected syntax is correctly fed to the refactoring operations.

3 Writting

3.1 Syntax-Parse

The Syntax-Parse function provided by Racket is rather useful for the refactoring operations regarding mainly syntax information. It has a wide range of options to help matching the correct syntax it also have backtracking. With Backtracking it is possible to have several rules to be match in the same syntax parser which helps to create more sophisticated rules.

Literal vs Datum-literal One of the options in the syntax-parse is to specify if an element is a literal. The #:literals option specifies identifiers that should be treated as literals rather than pattern variables. This option helps to ensure that a refactoring operation made is targeted only to the correct elements of the language. There is also another option that is very similar to the literals know as datum-literals. Datum-literals match symbols instead of an identifier and can be rather useful as the literals option because it provides a wider range of options.

However because of an unknown bug the literals option only works with the expanded-program and we are limited to use the datum-literals option. This could possible create incorrect refactoring transformations when the user renames the literals of the language. e.g(renaming the if, cond, let, defines, syntax, when, unless, etc)

3.2 Pretty Printing

Pretty Printing (E.g. Cond lets etc) The racket makes it easy to modify syntax using the syntax-parse to transform the AST into another AST. In order to produce indented code we choose to use a pretty-printer already incorporated in the language. However this pretty-printer does not follow the convention in the cond clauses should be surrounded by [] parenthesis. This is not considered a problem because Racket supports both representations. One possible solution is to use a different pretty-printer in order to keep the language convention.

4 Refactoring operations

4.1 Semantic problems

There are known semantic problems that might occur after doing a refactoring operation. One of those problem occurs when removing the and of the following example

Listing 1.4. "Example"

```
(and (< 1 (foo 2)) (< (foo 2) 3))
```

The refactoring transforms the code into this:

Listing 1.5. "Example"

```
(< 1 (foo 2) 3)
```

This refactoring has semantic problems if the function "foo" has collateral effects. And instead of applying those collateral effects twice it will only apply once therefore changing the semantic of the program.

We still decided to keep this refactoring operation because for the vast majority of the cases this does not change the semantic of the program and the solution would limit too much this kind of refactoring. Because of the reflection capabilities of racket we could only safely apply this refactoring when the arguments of the "¡" were datums (number, boolean or string).

4.2 Extract Functions

Extract function is an important refactoring operation that every refactoring tool should have. However is not that simple to implement and there are some things to take into account. In order to extract a function it is necessary to compute the arguments needed to the correct use of the function. While naming the new function it is an interesting feature to check if that name is duplicate. Then computing the body and replacing it by the call should be straightforward. Another problem is where should the function be extracted to. A function can not be defined in an expression, (e.g inside a let) but it could be defined in the top-level or in any other level that is accessible from the top level.

e.g: When extracting the (+ 1 2) to a function where should it be defined? Top-level? level-0 level-1 or in the current level, level-2?

Listing 1.6. "Example"

```
;;top-level
(define (level-0)
  (define (level-1)
    (define (level-2)
      (+ 1 2))
    (level-2))
  (level-1))
```

The fact is that is extremely difficult to know the answer to this question, and we think that the best solution to this question is to let the user decide where he want the function defined.

Computing the arguments In order to compute the arguments we have to know which variables are being define inside or outside the extracted function. The candidates to arguments of the function to be created are the variables defined outside, however imported variables, whether from the language or from other libraries does not have to be passed as arguments. There are no information

in the def-use-relations that indicates whether an variable is imported or not. We considered two possible solutions: -Def-use-relations + Text information -Def-use-relations + sExpressions

The first approach is simpler to implement and more direct than the second one. However it is less tolerant to changes and to errors. The second one combines the Arrow information with the syntax information to check whether it is imported from the language or from other library.

We choose the second approach in order to provide the refactoring tool with a better and lastly solution to compute correctly the arguments of the new function.

4.3 Let to Define

Changing a let form to a define could be rather useful when the user notices that instead of a let form it should be a function.

There are several types of let forms, but the most common are the let and the let*. There is a subtle difference between this two keywords that influences directly the simplicity of the solution. the let defines variables independently, while let* can use the value of the variable defined before. e.g: There is a global variable a defined with value 10. in the let we define variable a with 1 and variable b = a + 1 (let ([a 1] [b (+ a 1)]))

Listing 1.7. "Example"

```
(define a 10)
(let ([a 1]
      [b (+ a 1)])
      ...
      )
```

This let, because it defines the variables independently the value of b is 11.

Listing 1.8. "Example"

```
(define a 10)
(let* ([a 1]
      [b (+ a 1)])
      ...
      )
```

However in the case of let* the value of b is 2.

Therefore only let* would be considered because it is a more directly representation of a function.

Named let is a let that has a name and can be called, like a function. The named let* is directly mapped as a function and therefore might be useful to transform to a function. The same applies from a function to a named let*.

However this has a problem, a let form can be used in expressions, but the define can not. In the vast majority of cases this refactoring is correct, but when a let is used in an expression it is not correct and it changes the meaning of the program, transforming a correct program in a incorrect one. e.g

Listing 1.9. "Example"

```
(and (let* test ((a 1)) (< a 2)) (< b c))
```

Modifying this named let* into a define would raise a syntax error because a define could not be used in an expression context.

This could be solved by using the local keyword that is an expression like the let form. However the local is not used very often and can confuse the users. This reason made us keep the refactoring operation without the local keyword that works for most of the cases.

4.4 Define to Let

Refactoring Define to Let Usefulness Vs Implementation difficulty Useful for when changing several defines and merging them into a let. the let would "swallow" all the range of the defines scope Interesting?

It would be let* and named let* If define is a function it does not work. Is that worth it?

4.5 Ambiguities

There are some cases where two types of syntax refactoring apply. Eg:

Listing 1.10. "Example"

```
(if ?x
    (begin ?y ...)
    #f)
```

The two different refactoring transformations are possible:

Listing 1.11. "Example"

```
(when ?x
      ?y ...)
```

Listing 1.12. "Example"

```
(and ?x (begin ?y ...))
```

The programmer could want in some situations choose one approach and in others choose the other one. For example if a programmer is creating a predicate may choose the and version, whereas if the programmer is using another control structure may prefer the when version.

This example shows how hard it is to have an semi-automated refactoring tool that gives suggestions. It could displays both possibilities, but that will create an precedent meaning that if a refactoring has several possibilities the tool has to display every one. Or it could only display that there is a refactoring opportunity. This requires further reflection to choose the best approach to the problem.

4.6 Implemented Refactoring Operations:

5 Features

5.1 User FeedBack

It is an important to give proper feedback to the user while the user is doing the refactoring operations. It was studied a way to inform users what were the conditions that do not allow a refactoring operation to occur. And also inform which are the steps in order to allow that refactoring operation to occur, instead of just disabling the refactoring button. However after an analysis it was clear that kind of problems rarely occur in Racket language and therefore not implemented. However if a language is statement based instead of expression based the situation changes and the importance of the User Feedback increases.

5.2 Wide-Scope Replacement

The Wide-Scope replacement brings the possibility to replace all the duplicated code with a function call. This is usually used after an extract function refactoring. This is a huge improvement on the utility of the refactoring regarding the use of the extract function refactoring operation.

This searches for the code that is duplicated of the extracted function and then it replaces for the call of the extracted function. This "refactoring-operation" is divided in two steps: detecting duplicated code Replacing the duplicated code

Replacing the duplicated code is the easy part, however the tool might has to compute the arguments for the duplicated code itself. The argument computation occurs when the code is the same, but it has different variable names. This is not yet in this version of the refactoring tool.

Detecting duplicated code Detecting correctly the duplicated code is a key part for the correctness of this refactoring. The simplest form of duplicated code detection, when each code is exactly equal, could have some problems regarding the bindings. For example, if the duplicated code is inside a let that changes some binding that must be taken into consideration. Racket has already functions that compute if the bindings are the same. However that does not work if we consider the program in the not expanded form because there is not enough information for those bindings to work.

In order to compute the correct bindings it is necessary to use the expanded form of the program to detect the bindings correctly.

The naive solution is instead of using the expanded program to detect the duplicated code and then use this information to do the replacing of the duplicated code. However when expanding the program Racket adds necessary internal information to run the program itself that are not visible for the user. While this does not change the detecting of the duplicated code, this adds unnecessary information that would have to be removed. In order to solve this problem in a

simple way we can use the expanded code to detect the correctly duplicated code and use the non expand program to compute which code will be replaced.

However this detection is a quadratic algorithm (TODO check this) which might have some performance problems for bigger programs.

Detecting duplicated code can be added to the automatic detection of possible refactoring operations to be applied. Notifying the users of a possible extract function operation if there is duplicated code. This is a rather useful notification because for programs that are bigger than the visible part of the screen. Which might be difficulty for the user to remember if a piece of code was duplicated or not.

Automatic Suggesting Automatic Suggesting refactoring opportunities like the name suggests it suggests possible refactoring operations to the users. This feature is very useful in order to have a general idea what possible refactoring operations can be done in a piece of software. It is also important for inexperienced users because with this, they can have an idea what refactoring operations can be applied or not.

This features parses the code from the beginning to the end and tries to check if a refactoring is applicable. To do that it tries to match every syntax expression using syntax parse. In other words it uses brute force to check whether a expression can be applied a refactoring operation or not. Automatic Suggesting is mainly applied to syntax refactoring operations, however it could be applied to the Extract-Function refactoring operation using the simple detection of duplicated code to inform the user that might be useful to extract a function instead of having duplicated code.

In order to show this information to the users it highlights the source-code indicating that there is a possible refactoring. This feature could be improved by having a set of colors for the different types of refactoring operations. And the color intensity could be proportional to the level of suggestion. (e.g the recommended level to use extract function refactoring increases with the number of duplicated code found)

5.3 Automated refactoring

Automatic refactoring is an option that the user decides to let the refactoring tool do every refactoring operation that it founds. It is a feature that usually used after the automatic suggesting and when the user wants that every refactoring operation found to be applied.

6 Tool Maintenance

Developer Point of view This feature combined with the automatic suggesting are a huge help to do the evaluation of this refactoring tool and to use in battery tests. First the automated refactoring displays all the possible refactoring found thus making it easier to detect the refactoring operations in projects. It makes it

easier to evaluate the need to do the refactoring and to evaluate if the refactoring tool (frameowork) is working correctly. Second the automatic refactoring makes it easier to test if the refactoring operations are correct (**** for a limited test cases ****) This is possible by running the test cases of the program before any refactoring. Then using automatic refactoring in the program that applies all the refactoring that the tool finds. And finally by running again the test cases of the program. After testing it with several (or huge) programs if every test passes (or the exact same tests, because refactoring operations should not fix bugs) we can consider that the refactoring operations are correct.

7 Analysis

Table 1. Data Structures

Name	AST	PDG	Database	Others
Griswold	X	X		
HaRe	X			
Rope	X		X	
Bicycle	X		X	

This table summarizes the data structures of the refactoring tools deeply analyzed. It is clear that the AST of a program is an essential part of the refactoring tool information with every Refactoring tool having an AST to represent the program. Regarding the PDG and Database it has mainly information about the def-use-relation of the program. The PDG has also control flow information among others.

HaRe only uses the AST as a source of information of the program. Thus, by not having the def-use-relation or a PDG it has less information to perform the refactoring operations. However because HaRe is for the Haskell program language that is purely-functional programming language that extra information is not necessary to perform a good set of refactoring operations correctly.

Our developed tool uses the same kind of information, namely the AST and the def-use-relations. Some tools like the one build by Griswold has more information available when compared to the one developed by us. However we find out that for expression based languages such as racket and focused in the functional part it is sufficient an AST and def-use-relations.

8 Evaluation

Case Study: (find a good ones) FP Project, Architecture Project. For SLATE Correctness: Here?

9 Conclusion

def-use-relations + AST is sufficient.
Importance of some refactoring operations
Maintainability of the refactoring tool?
Framework?

10 Future work

Detect when a developer is refactoring in order to help the developer finish the refactoring by doing it automatically. Improving automatic detection Wide-scope-replacement smarter and better, computation of the arguments

References

[1] William G Griswold. Program restructuring as an aid to software maintenance. 1991.

[2] Simon Thompson. Refactoring functional programs. In *Advanced Functional Programming*, pages 331–357. Springer, 2005.