Refactoring Dynamic Languages

Rafael Reia and António Menezes Leitão

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

1 Architecture

Refactoring Tool work flow: (add a brief explanation of what does) The Refactoring tool uses two types of information, the def-use-relations and the AST.

It gets the def-use-relations directly from DrRacket, which is visually represented in a form of Arrows, and are used in refactorings such as Extract- Method, Add-Prefix, Organize-Imports, etc.

The AST is acquired after using the check-syntax button. It was intended to be acquired automatically, because DrRacket has an online check syntax, but it was not possible. After that it cleans the AST removing information that is not necessary, at least for now, for the refactoring operations. When the AST is cleaned it passes the exact part of the AST that was selected. And tries to match rules, using syntax-parse against that syntax.

1.1 Def-Use-Relations

DrRacket has def-use-relations in the system that are visually represented by arrows. The information is computed in the online-comp.rkt that is where expands and computes the information needed to know the def-use-relations and whether the program is syntactically correct. There is an online expansion monitor that calls the online-comp, the online-comp then expands the program (to the expanded program) and does a trace to get the information needed for the arrows (def-use-relations). Afterwards it sends back that information to the caller of the online-comp to do the replay of the trace did by the compile comp and to process the information about the arrows.

1.2 Def-Use-Relations utility

Arrows can be used to check whether or not there is a duplicated name or to compute the arguments of a function to be extracted.

Arrow Evolution In previous versions of the online-comp information about the type of arrows was available in the structure. However in the recent versions that type of information was removed. With the previous version of the online-comp the framework was computing the argument for the newly extracted function using the type information: ”one of ’lexical, ’top-level, ’import”. That change created a problem while trying to check whether an argument was imported from the language or from another file/library that had at least two solutions: -Arrows + Text information -Arrows + sExpressions

The first approach is simpler 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.

1.3 Get Syntax Expressions

The syntax expression tree is how the program is represented in Racket.

DrRacket product DrRacket produces syntax from the program in two places. In the Online compilation time and via the Check Syntax represented by a button. Online-comp is the racket's online check syntax. Besides being online we do not know what calls the online syntax expander. Because of that we can not find a way of storing the AST information that we need to do the refactoring operations. Not knowing what calls the online syntax expander makes it impossible to store the information because the instance that receives the AST information is not accessible, it disappears. Having an instance of gui.rkt in online-comp.rkt / code-walker.rkt does not work either because we can not have 2 instances of the same thing running at the same time.

Therefore there are three possible solutions:

Write to a file Find out what is calling/starting the handler and get the results from there Use Check Syntax Button The last one was temporarly chosen because the others failed, either because we wasn't able to store syntax objects in a file (in a "small" number of time) or because we could not find what is calling that (in a "small" number of time).

The solution chose can be improved by working *automatically*. To do that we Have two options: Use the Online check syntax Automate the "call" to the check syntax button. Possibility the automate the call to the check syntax button will be chosen. This simplifies the process, however it increases the time that the user has to wait, and can make it unusable. This approaches besides not being ituitive for the user, or hard to do. They have a problem when doing several refactoring operations. Because If the user forgot to call the check syntax (by clicling on the button), the refactoring operation would affect an outdated syntax expression tree and produce wrong results. The results could be weird and created syntax errors but sometimes could be syntatic correct and insert bugs/ change the meaning of the program.

A better approach is for each refactoring to compute the Syntax empressions tree. With this it makes the call more intuitive for the user and essures that each refactoring operation is made with an updated syntax expression tree.

1.4 Syntax Expression tree form

Racket has two forms of syntax tree. An expanded form, with the macros expanded and a non expanded form that is after a read-syntax without the macros expanded.

Expanded Program Vs Not expanded Program #Talk about difficulty, durability, Python (other languages)

[Only in the expanded form] (and) is treated like an if which is bad and give problems The rest tends to work. testing: (if (= (+ 1 2) 1) #f #t) to (not (= (+ 1 2) 1))

#add an example of both to compare

Listing 1.1. "example"

```
(if (< 1 2)
    #t
    #f)
```

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)))>
```

Racket uses s-expressions to represent the program, basically it represents the abstract syntax tree (AST) of the program. Racket has two "trees" the fully expanded program, that has all the macros expanded and the non expanded program, that is more like the code the user programed. The fully expanded program loses some information about the program, for example And. Or, When, Unless are transformed into if forms. It also adds racket internal representation of the program into the AST. All these combine make it harder to create refactoring operations to the program, because it loses information and because it adds unnecessary syntax to the program. It is necessary to take into consideration that the internal representation of the program may change in the future and that could make the refactoring tool useless.

However, if the program contains macros using the non expanded program may result in semantic incorrect refactoring operations. This is not considered a 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 for more experienced programmers the non expanded program is insufficient and the expanded program must be used.

Using the expanded program might simplify the refactoring for other languages, for example it has a literal that says when the program returns in python, and racket does not have it. In the expanded form there is syntax that explicitly says where the program returns.

In the future it is possible to create a tool that uses both program expansions, making it possible to have macros and the relation with other languages for the more difficult problems and using the non expanded program for the simple cases.

Semantic problem (E.g. (and (¡ 1 (foo 2)) (¡ (foo 2) 3))) The refactoring operation that merges two ands into one, can change the semantic form of the program. because if foo has collateral effects (changing the value of a variable, printing some stuff, etc) it will only do that once, instead of twice. changing the semantic of the program.

2 Code-walker

It is used as a parser for the syntax tree. the way that the code-walker guarantees that goes to every important part of the tree is by having a "pointer" to the current selected syntax element, and have a stack that contains the remaining code to analyze. Each syntax element could be a syntax-pair, one syntax element or the empty list. If it is a pair of syntax elements they are organized by (stx . pair) In which the first element is a syntax element and the second one is a syntax pair. 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. Because of this structure it is necessary to have a stack in order to search the tree correctly and in the best way possible. Because Most of the time we are searching for a specific syntax element (e.g function, define, etc, because everything is a syntax element in Racket) and using this we can skip uninteresting syntax in order to get to the syntax that we want to use.

3 Syntax-Parse

Syntax-Parse Utility Racket gives a function that parses syntax. The Syntax-Parse provided by Racket is rather useful. because besides having a wide range of auxiliary functions that helps matching the syntax it also has 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.

4 Pretty Printing

Pretty Printing (E.g. Cond lets etc) The racket makes it easy to create syntax using the syntax-parse to transform the AST into another AST. however the syntax elements (s-expressions) lose some information about parenthesis. For example it is a convention that cond clauses should be surrounded by [] parenthesis but the syntax element does not store that information among others. One possible solution to this problem is by using a pretty-printer. There are several pretty-printers developed for Racket and even the Racket language has one incorporated. The one already incorporated does not use the [ ] parenthesis, however racket supports both representations.

4.1 Refactoring operations

Extract Functions Compute arguments

Body

Call

Paste (etc)

Let to Defines Refactoring Let to Defines Usefulness Vs implementation difficulty let* vs 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)])) This let, because it defines the variables independently the value of b is 11. However if it was used let* the value of b would be 2. let* simplifies the creation of defines and removes the necessity of using the keyword local to ensure semantic preservation.

While the keyword local helps semantic preservation, that is not used very often and can make confusion to the programmer.

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

named let. Named let is a let that has a name and can be called, like a function. It 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. Let 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.

Define to Let Refactoring Define to Let Usefulness Vs Implementation difficulty

Ambiguities Refactoring (if ?x (begin ?y ...) #f) could go either way: (when ?x ?y ...) or (and ?x (begin ?y ...))

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. 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.

Implemented Refactoring Operations:

5 Features

5.1 User FeedBack

5.2 Automated refactoring

Automated refactoring, after finishing a project to make it more beautiful. (could be interesting) This is already done, it calls every possible refactoring operation that the tool finds. This is kinda dangerous to use, however this might be useful for the user to see all the results of the refactoring operations and a good way to test if the refactoring operations do not change the meaning of the program.

Automatic Suggesting refactoring opportunities. 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 refactored or not.

For now this is only applicable for syntax refactoring and not for extract-function.

In order to show the possible refactoring operations it highlights the source-code indicating that there is a possible refactoring.

The Automatic suggesting can be easily extend to automatic refactoring. (and will be)

This features are a huge help to do the evaluation for two reasons. First the automated refactoring displays all the possible refactoring found thus making it easier to detect the refactoring operations in projects 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.

Wide-Scope-Replacement: 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. Doing that is not a trivial job and that might be in the future work.

Detecting duplicated code: This is the key for the correctness of this refactoring, even in the simplest way that the code must be exactly equal, same names, same bindings, etc. For example, if the duplicated code is in a let that changes some binding it must be taken into consideration. Racket has already functions that compute if the bindings are the same, however because the refactoring tool uses the program in the not expanded form there is not enough information for those bindings to work.

[TODO Recently racket changed binding expansion and it brought interesting improvements to racket and that might be useful for this, read paper before]

To solve this it is necessary to use the expanded form of the program in order to compute the correct bindings. The naive solution is instead of using the non 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 information to run the program itself that are not visible for the user. while this does not change the detecting of the duplicated code, because if the code is duplicated it will have the same extra information. It changes the way the tool computes which code will be replaced.

In order to solve this problem in a simple way we can use the non expand program to compute which code will be replaced. This runs after detecting the duplicated code, so the bindings are corrected identified.

Detect possible refactoring operations (extract funcion): This can be added to the automatic detection of possible refactoring operations to be applied. notifying the users of a possible extract function operation. 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.

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

6 Plugin

In order to improve the deployement and usage of the refactoring tool a plugin for DrRacket was createad. DrRacket has several advantages and the plugin support is another one, because it allows a simple integration with DrRacket itsel a simple mananing interface. Another advantage is the incorporation with git. A plugin can be a link for a git repository and because of that it becomes simple to keep the refactoring tool up to date.

However, the main reason to create a plugin is to become independent from DrRacket evolution. With a plugin that is possible and it is managable.

7 Evaluation

Case Study: (find a good ones) FP Project, Architecture Project. Correctness

8 Future work

Detect when a developer is refactoring in order to help the developer finish the refactoring by doing it automatically.

Wide-scope-replacement smarter and better

9 FrameWork

Python: [framework] One of the goals of this thesis is to create refactoring for dynamic languages in general, in other words, to create a Framework for refactoring tools. //by using the same structure (program, framework, choose one better) After creating several refactoring operations for Racket Python was chosen next. POC (prove of concept) It was created a prove of concept in Python consistent in several refactoring operations, such as: [TODO] say which refactoring operations were implemented. Python was chosen because there is already an implementation of Python for the DrRacket environment in which the refactoring tool for Racket was first developed.

The implementation for DrRacket is what does the trick, because by implementing the language with Racket syntax we are basically using as a meta-language that can represent several languages. Being the meta-language a language that are composed only by syntax elements is a huge advantage to compute effortless new refactoring operations for new languages when compared with the necessary effort to create the refactoring operations directly for that language.

how easy it is to add new refactoring operations and languages The framework it is simply to use, it is only necessary to have a specification file of each refactoring operation. That file must have a function that receives two arguments, one is the AST of the program and the other is the def-use-relation. This information makes it possible to have several refactoring operations that help the programmer. what was "reused" everything except the refactoring operations itself. the advantages of that This Framework makes it easier to implement refactoring operations for dynamic languages, with only the catch that they have to be implemented for DrRacket. Helping minimizing the problem of the difficulty and lacking of refactoring operations for dynamic languages. (for at least every language implemented for DrRacket)

References