Refactoring Dynamic Languages

Rafael Reia
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
rafael.reia@tecnico.ulisboa.pt

António Menezes Leitão
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

ABSTRACT

The need to create better support tools rises with the growing importance of programming as a skill among areas non related with software development. One important tool of an IDE is the refactoring tool which allow the users to safely improve their program quality.

In this paper, we present a refactoring tool for beginner programmers which we implemented in DrRacket IDE. Our tool provides simple refactoring operations for the typical mistakes made by beginners.

1. INTRODUCTION

The support provided to beginner programmers is increasing and there are already several languages known to be suited for the initial contact with programming, such as Scheme, Racket, Python and JavaScript which are used in introductory courses around the world. In addition, there are integrated development environments (IDEs) targeted for users with little or no previous contact with programming [1]. The pedagogically-aware IDE provides the tools and the means to create better programs while simplifying the complexity of a typical IDE [2].

After learning the fundamentals of programming a beginner programmer should have the capacity to transform the code systematically. In order to help with that there are the refactoring tools, which provides support to refactoring operations intended to improve the design of an existing code base [3] without changing the behavior of the program. Languages used in introductory courses already have refactoring tools available, however they were made for more advanced users and not for beginners. The lack of refactoring operations for beginner programmers makes those refactoring tools unfit for a beginner.

A refactoring tool for beginner programmers needs to improve code made by them, with refactoring operations for the typical errors made, and simple enough to be used by a beginner programmers. Automatic detection of possible refactorings would also help the users to know which refactoring operations available and where they are applicable.

In contrast, the typical refactoring tools do not provide any support for the detection of code which might or should be refactored. On top of that, the IDEs in which most of those tools are embedded, such as Eclipse[4], IntelliJ [5], NetBeans [6], VisualStudio[7], Vim[8], and Emacs[9], are too complex for beginners to use, requiring the user to understand several complex menus or to learn the special commands to properly use the IDE. As a result this IDEs have a steep learning curve, which makes it difficult for beginners to explore the tool. For instance, Eclipse has around 300 menu bar options and Visual studio 280, which is a massive amount of options for the user to select.

In this paper we present a refactoring tool aimed for beginner programmers which typically had one introductory course. Our tool helps to detect the typical programming mistakes made by students and in addition can make suggestions showing the possible refactoring operations found. This refactoring tool brings a new set of options for the beginners to use in order to safely improve their code.

We implemented our idea for a refactoring tool in DrRacket, formerly known as DrScheme is a pedagogical IDE [10] [11]. DrRacket was tailor made for the Racket programming language, which currently has only one simple refactoring operation which allows renaming a variable. Our implementation significantly extends the set of refactoring operations available in Drracket.

2. RELATED WORK

Several refactoring tools were analyzed to guide our development. Our focus was on dynamic languages which are used in introductory courses or that have similarities with Racket.

2.1 Scheme

A refactoring tool [12] for scheme, implemented in Lisp that uses two forms of information, AST (Abstract Syntax Tree) 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 dependency information of the program and relying only in this information to represent the program could create problems. For example, if two semantically unrelated statements they could be placed arbitrarily with respect to each other. Using the AST as the

WOODSTOCK '97 El Paso, Texas USA
© 2016 ACM. ISBN 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

main representation of the program ensures that statements are not arbitrarily reordered. And the PDG is only used as a notation to prove that transformations preserve the meaning and as a quick way to retrieve needed dependence information. Contours are used with the PDG providing scope information, which is 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.

2.2 Haskell

HaRe [13] 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.

2.3 Python

2.3.1 Rope

Rope[14, p. 109] is a Python refactoring tool written in Python, which works like a Python library. In order to make it easier to create refactoring operations, Rope assumes that a Python program only has assignments and functions calls. Thus, by limiting the complexity of the language it reduces the complexity of the refactoring tool. Rope uses a Static Object Analysis, which analyses the modules or scope to get information about functions. Rope only analyses the scopes when they change and it only analyses the modules when asked by the user, because this approach is time consuming.

Rope also uses a Dynamic Object Analysis that requires running the program in order to work. The Dynamic Object Analysis gathers type information and parameters passed to and returned from functions in order to get all the information needed. It stores the information collected by the analysis in a database. If Rope needs the information and there is nothing on the database the Static object inference starts trying to infer the object information. This approach makes the program run much slower, thus it is only active when the user decides.Rope uses an AST in order to store the syntax information about the programs.

2.3.2 Bicycle Repair Man

Bicycle Repair Man Bicycle Repair Man is a Refactoring Tool for Python written in Python. This refactoring tool can be added to IDEs and editors, such as Emacs, Vi, Eclipse, and Sublime Text. Bicycle Repair Man is an attempt to create the refactoring browser functionality for Python and has the following refactoring operations: extract method, extract variable, inline variable, move to module, and rename.

The tool has an AST to represent the program and a database that has information about several program entities and dependency information.

2.3.3 Pycharm-edu

Pycharm Educational Edition[1], or Pycharm edu, is a IDE for Python created by JetBrains, the creator of IntelliJ. The IDE was specially designed for the educational purpose, for programmers with little or no previous coding experience. Pycharm EDU is a simpler version of Pycharm community which is the free python IDE created by JetBrains. Therefore it is very similar to their normal IDEs and it has interesting features such as code completion, version control tools integration. However it has a simpler interface when compared with Pycharm Community and other IDEs such as Eclipse or Visual Studio.

It has integrated a python tutorial and the big advantage is the possibility of the teachers creating tasks/tutorials for the students. However the Refactoring Tool did not received the same care as the IDE itself. The refactoring operations are exactly the as the Pycharm community IDE which were made for more advanced users. Therefore it does not provide specific refactoring operations to beginners. The embedded refactoring tool uses the AST and Def-use relations in the refactoring operations.

2.4 Javascript

There are few refactoring tools for JavaScript but there is a framework [15] for refactoring JavaScript programs. In order to guarantee the correctness of the refactoring operation, the framework uses preconditions, expressed as query analyses provided by pointer analysis. Queries to the pointer analysis produces over-approximations of sets in a safe way to have correct refactoring operations. For example, while doing a rename operation, it over-approximates the set of expressions that must be modified when a property is renamed in a safe manner. To prove the concept, three refactoring operations were implemented, namely rename, encapsulate property, and extract module. By using over-approximations it is possible to be sure when a refactoring operation is valid. However, this approach has the disadvantage of not applying every possible refactoring operation, because the refactoring operations for which the framework cannot guarantee behavior preservation are prevented. The wrongly prevented operations accounts for 6.2% of all rejections.

3. ARCHITECTURE

In order to create correct refactoring operations, the refactoring tool uses two sources of information, the def-use-relations and the AST of the program. The def-use-relations represent the definition of an identifier and their usage and is visually represented in a form of Arrows in DrRacket.

The AST is represented by the syntax expressions (s-exp) which composes 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.

Figure 3 summarizes the work flow of the refactoring tool where the Reader produces the non expanded program while the Expander produces the expanded program. Afterwards the Code walker parses the AST produced by the Reader or the Expander depending the cases. In order to produce the Def-Use-Relations it is necessary to use the code produced by the Expander because it has the correct dependency information. The Transformer uses both information from the

[1]https://www.jetbrains.com/pycharm-edu/

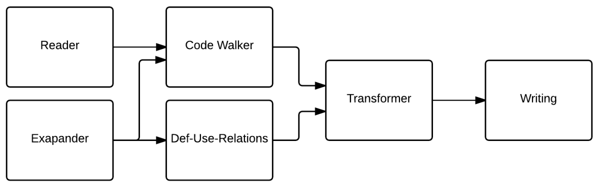

Figure 1: Information flow between modules.

Code Walker and the Def Use Relations to transform the code. Then it goes to the Writing module that produces the output in the Definitions area.

3.1 Syntax Expressions

The s-exp list represent the AST, which provides information about the structure of the program. The s-exp list it is already being produced and used by the Racket language and in DrRacket in order to provide error information to the user. DrRacket already provides functions which computes the program's s-exp list and uses some of those functions in the online check syntax and in the check syntax button callback.

3.1.1 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 with all the macros expanded. The other format is the non-expanded program and computes the program with the macros unexpanded.

The expanded program has the macros expanded and the identifier information correctly 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: "example"

```
(and #f #t)
```

Listing 2: "Expanded program"

```
#<syntax:2:0
 (#%app call-with-values
  (lambda () (if (quote #f)
   (quote #t) (quote #f)))
   print-values)>
```

Listing 3: "Non-expanded program"

```
#<syntax:2:0 (and #f #t)>
```

The expanded program transforms the `and`, `or`, `when`, and `unless` forms into `if`s which 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, the expanded program has a format that is likely to change in the future. Racket is an evolving language and the expanded form is a low level and internal form of representation of the program. Additionally we do not consider macros as part of a code that could be refactored, since the refactoring tool is targeted at unexperienced programmers macros will not be used often and therefore it is not considered part of the scope of this refactoring tool capabilities.

Therefore it is desirable to use the non expanded form for the refactoring operations whenever possible and use the expanded form only for the necessary operations. Nevertheless, if we intended to create a tool that gives support to refactoring macros using the non expanded program would let to errors making the expanded program the representation of the program used. Even considering that are no guarantees that would be enough to ensure the correctness of such refactoring operations due to the reflection capabilities of Racket.

3.2 Def-Use-Relations

Def-Use-Relations holds an important information in order to produce correct refactoring operations. 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. DrRacket already uses the def-use-relations in the system and they 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)

3.3 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 either 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 to search for the correct elements.

Most of the time using the code-walker we are searching for a specific syntax element and location information contained in the syntax-object is used 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.

3.4 Pretty-printer

Producing correct output is an important part of the refactoring tool. It is necessary to be careful to produce indented code and we decided to use a pretty-printer that is 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.

3.5 Comments preservation

Preserving the comment after a refactoring transformation is an important task of the refactoring tool. If the comment in determined place of the program changes hits location affecting another structure it could confuse the programmer. However comment preservation is not implemented yet, making it a limitation of this prototype.

One possible solution is to modify the syntax reader and add a comment node to the AST. While the new node will not be used during refactoring transformations it is used during the output part of the refactoring operation preserving the comment with the correct syntax expression.

3.6 Syntax-Parse

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

4. REFACTORING OPERATIONS

In this section we explain some of the more relevant refactoring operations and some limitations of the refactoring tool.

4.1 Semantic problems

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

Listing 4: ”And example”

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

The refactoring transforms the code into this:

Listing 5: ”Example”

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

Listing 6: ”Foo”

```
(define (foo arg)
  (displayln "foo"))
```

Instead of applying the side effect that is displaying the the string ”foo” it will only display the it once. Therefore changing the meaning of the program.

We still kept this refactoring operation because in the vast majority of the cases this refactoring operation does not change the semantic of the program. Furthermore, the possible solution would limit excessively this refactoring operation. Considering Racket's reflection capabilities we would only apply this refactoring operation safely when the arguments of the `and` expression, in this case `<` and the function `foo` were datums (number, boolean or string). This line of thought is also applicable to the `or` form.

E.g.:

Listing 7: ”Code sample”

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

The two different refactoring transformations are possible:

Listing 8: ”Refactoring option 1”

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

Listing 9: ”Refactoring option 2”

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

The first refactoring option changes the meaning of the program, because if the test expression, in this case ?x, is false the result of the when expression is #<void>. However, the programmer may still 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 and do not care for the result of the expression may prefer the `when` version.

4.2 Extract Function

Extract function is an important refactoring operation that every refactoring tool should have. However there are some concerns to have into account. In order to extract a function it is necessary to compute the arguments needed to the correct use of the function. While giving the name to a function seams quite straightforward it is necessary to check for name duplication in order to produce a correct refactoring. e.g. having two identifiers with the same name and in the same scope produces an incorrect program and therefore modifying the meaning of the program. Then computing the body and replacing it by the call should be straightforward. Another problem is where should the function 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 current 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 10: ”Extract function levels”

```
;;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 because it depends on what the user is doing and the user interpretation. Accordingly we think that the best solution is to let the user decide where the user wants the function defined.

4.2.1 Computing the arguments

In order to compute the arguments we have to know in which scope the variables are being defined, in other words, if the variables are defined inside or outside the extracted function. The variables defined outside the function to be extract are the candidates to be the argument of that function, however imported variables, whether from the language or from other libraries does not have to be passed as arguments. We considered two possible solutions:

-Def-use-relations + Text information
-Def-use-relations + AST

The first approach is simpler to implement and more direct than the second one. However it is less tolerant to future 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 a more stable solution to compute correctly the arguments of the new function.

4.3 Let to Define

We noticed the similarity of a `let` expression with a function, therefore making it easy to choose one of them by mistake. Therefore we decided to provide a refactoring operation that would make that transition simpler.

There are several let forms, but we want to focus in the ones more similar to a function, namely `let`, `let*`, and `named let`. The ones that closely represent a function is the `let` and the `named let` because of the difference between the `let` and `let*` evaluate their values expressions. The `let` defines variables independently, while `let*` can use the value of the variable defined before.

The `let` and the `named let` can be directly mapped to a function, however the `named let` can be directly mapped to a define whereas the `let` can only be directly mapped to an anonymous function. However, we did not consider the transformation of a `let` to an anonymous function, `lambda`, making the code simpler and therefore it was not implemented yet.

However, the refactoring operation that transform a `named let` into a `define` function, could have syntax problems because 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 11: "Let in an expression"

```
(and (let xpto ((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 Wide-Scope Replacement

The Wide-Scope replacement brings the possibility to replace all the duplicated code with a function call. This is usually performed after an extract function refactoring.

The Wide-Scope replacement refactoring operation searches for the code that is duplicated of the extracted function and then it replaces for the call of the extracted function and it is divided in two steps:

- Detect duplicated code
- Replace 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.

4.4.1 Detecting duplicated code

Correctly detect code duplication is a key part for the correctness of this refactoring. Even the simplest form of duplicated code detection, when it only detects code duplication when the code is exactly equal, may 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 already provides 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.

Therefore, in order to compute the correct bindings, it is necessary to use the expanded form of the program.

The naive solution is to use 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 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.

5. FEATURES

This section describes some of the features that improve the utility of this refactoring tool to beginner programmers.

5.1 User FeedBack

It is important to give proper feedback to the user while the user is atempting or preforming refactoring operations. Instead of just disabling the refactoring operation button the tool should provide information about the steps needed in order to be possible to apply that refactoring operation. However, after an analysis it was clear that these situations rarely occur in Racket language and therefore it was not implemented. However if a language is statement based instead of expression based the situation changes and the importance of the User Feedback increases.

Previewing the outcome of a refactoring operation is an efficient form to help the users understand the result of a refactoring before even applying the refactoring. Previewing works by applying the refactoring operation in a copy version of the AST and displaying those changes to the user.

5.2 Automatic Suggestions

Beginner programmers usually do not know which refactoring operations exist or which can be applied. By having a automatic suggestion of the possible refactoring operations

Table 1: Data Structures

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

available the beginner programmer can have an idea what refactoring operations can be applied or not.

In order to detect possible refactoring operations it 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.

To properly display this information 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)

6. ANALYSIS

The 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 implementation uses the same data structures, the AST and def-use-relations. The def-use-relation is often represented as a database, some refactoring tools annotate that information in the AST, some tools extract the information from the AST itself when it is possible and it is possible to extract that information from the PDG. Some tools have more information about the program, either because they need that information to perform the refactoring operation or because they need to prove that the refactoring is correct.

Some tools like the one build by Griswold focus on the correctness of the refactoring operations. Others, focus in offering refactoring operations for professional or advanced users. However, the goal of our refactoring tool is to provide refactoring operations designed for beginners. Therefore we are not interested in refactoring operations formerly proven correct or provide refactoring operations only used in advanced and complex use cases. We intend to have simple, useful, and correct refactoring operations for the usual use cases a beginner would use. With this set of scope we exclude macros usage, classes and other complex structures not used by beginners.

7. EVALUATION

In this section we present some code examples written by beginner programmers in their final project of an introductory course and their possible improvements using the refactoring operations available in our refactoring tool. The examples show the usage of some of the refactoring operations previous presented and here is explained the motivation for their existence. This code mistakes appear repeatedly in almost every project supporting the need to this kind of refactoring.

The first example is a very typical error made beginner programmers. Almost every project has one or more piece of code similar to this one

```
 (if (>= n_plays 35)
      #T
      #F)
```

It is rather a simple refactoring operation, but nevertheless it improves the code.

```
 (>= n_plays 35)
```

The next example is related with the conditional expressions, namely the `and` or `or` expressions. We decided to choose the `and` expression to exemplify a rather typical usage of this expression.

```
(and
  (and
   (eq? #t (correct-moviment? player play))
   (eq? #t (player-piece? player play)))
  (and
   (eq? #t (empty-destination? play))
   (eq? #t (empty-start? play))))
```

Transforming the code by removing the redundant `and` expression the code becomes cleaner and simpler to understand. Needless to say it is still a rather simpler refactoring operation.

```
(and (eq? #t (correct-moviment? player play))
     (eq? #t (player-piece? player play))
     (eq? #t (empty-destination? play))
     (eq? #t (empty-start? play)))
```

However, this code could still be improved, the `(eq? #t ?x)` is a redundant way of simple writing `?x`.

```
(and (correct-moviment? player play)
     (player-piece? player play)
     (empty-destination? play)
     (empty-start? play))
```

While a student is still learning and experiencing it is easier to confuse whether or not an expression needs the the "begin" expression or not. The `when`, `cond` and `let` expressions have a implicit `begin` expression and as a result it is not necessary to add the `begin` expression. Our refactoring tool checks for those mistakes and corrects them.

```
 (if (odd? line-value)
     (begin
       (let ((internal-column (sub1 (/ column 2))))
         (if (integer? internal-column)
             internal-column
```

```
            #f)))
    (begin
      (let ((internal-column (/ (sub1 column) 2)))
        (if (integer? internal-column)
            internal-column
            #f))))
```

It is a simple refactoring operation and it does not have a big impact, however it makes the code clear and helps the beginner programmer to learn that a `let` does not need a `begin`.

```
(if (odd? line-value)
      (let ((internal-column (sub1 (/ column 2))))
        (or (integer? internal-column)
            internal-column))
      (let ((internal-column (/ (sub1 column) 2)))
        (or (integer? internal-column)
            internal-column)))
```

The next example shows nested if expressions. This kind of nested if expressions is difficult to understand, error prone, and is a debugging nightmare. It is much simpler to have a `cond` expression. In addition, every true branch of this nested if contains if expressions that are or expressions and by refactoring those if expressions to ors it makes the code simpler to understand.

```
(define (search-aux? board line column piece)
 (if
  (> column 8)
  #f
  (if
   (= line 1)
   (if
    (eq? (house-board
          board 1 column) piece)
    #t
    (search-aux? board line
                    (+ 2 column) piece))
   (if (= line 2)
    (if
     (eq? (house-board
            board 2 column) piece)
     #t
     (search-aux? board line
                    (+ 2 column) piece))
    (if (= line 3)
     (if
      (eq? (house-board
              board 3 column) piece)
      #t
      (search-aux? board line
                    (+ 2 column) piece))
     (if (= line 4)
      (if
       (eq? (house-board
                board 4 column) piece)
       #t
       (search-aux? board line
                      (+ 2 column) piece))
      (if (= line 5)
       (if
        (eq? (house-board
              board 5 column) piece)
        #t
        (search-aux? board line
                        (+ 2 column) piece))
       (if (= line 6)
        (if (eq? (house-board
                   board 6 column) piece)
         #t
         (search-aux? board line
                         (+ 2 column) piece))
        (if (= line 7)
         (if (eq? (house-board
                    board 7 column) piece)
          #t
          (search-aux? board line
                          (+ 2 column) piece))
         (if (= line 8)
          (if (eq? (house-board
                     board 8 column) piece)
           #t
           (search-aux? board
              line (+ 2 column) piece))
          null))))))))))
```

This transformation consists in an aggregation of two refactoring operations, `if-to-cond` and `if-to-or`. The `if-to-cond` has the biggest impact and reduces drastically the number of lines. In addition the if-to-or helps to make the code clearer. It drastically reduced the lines of code, from 59 to 28 that accounts for a 52% reduction of the lines of code.

```
(define (search-aux? board line column piece)
 (cond
 [(> column 8) #f]
 [(= line 1)
  (or (eq? (house-board board 1 column) piece)
    (search-aux? board line (+ 2 column) piece))]
 [(= line 2)
  (or (eq? (house-board board 2 column) piece)
    (search-aux? board line (+ 2 column) piece))]
 [(= line 3)
  (or (eq? (house-board board 3 column) piece)
    (search-aux? board line (+ 2 column) piece))]
 [(= line 4)
  (or (eq? (house-board board 4 column) piece)
    (search-aux? board line (+ 2 column) piece))]
 [(= line 5)
  (or (eq? (house-board board 5 column) piece)
    (search-aux? board line (+ 2 column) piece))]
 [(= line 6)
  (or (eq? (house-board board 6 column) piece)
    (search-aux? board line (+ 2 column) piece))]
 [(= line 7)
  (or (eq? (house-board board 7 column) piece)
    (search-aux? board line (+ 2 column) piece))]
 [(= line 8)
  (or (eq? (house-board board 8 column) piece)
    (search-aux? board line (+ 2 column) piece))]
 [else null]))
```

Beginners often use error and trial approach in code writing which led to peaces of code like the presented above. If the users had a refactoring tool that highlighted their code in areas that could be improved, they probably would not have this kind of code.

8. CONCLUSION

The growing interest in programming as a skill combined with the need of areas non related with any computation field creating the need to improve the support given to the beginner programmer. Therefore a refactoring tool designed for beginners in a pedagogical environment such as DrRacket would benefit those users as it would help them in their first contact with a refactoring tool and improve their code safely.

Our solution tries to help those users to improve their programs and to facilitate the first contact with a refactoring tool. By having a refactoring tool designed for beginners in a pedagogical environment that suggests possible refactoring operations.

We also shown the practicability of the refactoring tool with simple refactoring operations that improved safely the beginners code.

There are some improvements that we consider important and in the future it would be a huge improvement detecting when a developer is refactoring in order to help the developer finish the refactoring by doing it automatically [16].

The detection of duplicated code is still very naive and improving to understand if two variables represent the same even if the names are different or even if the order of some commutative expressions is not the same would make a huge improvement on the automatic suggestion.

It is possible to improve the automatic suggestion of refactoring operations by having different colors for different types of refactoring operations. This would let the user know what refactoring is being suggested without having to select the area. Another improvement that could be made is the color intensity of the suggestion. With a lower intensity for low "priority" refactoring operations and a high intensity for higher "priority". Thus giving the user a better knowledge of what is a better way to solve a problem or what is a strongly recommendation to change the code.

9. REFERENCES

[1] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The bluej system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[2] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*, 39(4):204–223, 2007.

[3] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[4] David Carlson. *Eclipse Distilled*. Addison-Wesley Reading, 2005.

[5] Heiko Böck. Intellij idea and the netbeans platform. In *The Definitive Guide to NetBeans^{TM} Platform 7*, pages 431–437. Springer, 2011.

[6] Tim Boudreau, Jesse Glick, Simeon Greene, Vaughn Spurlin, and Jack J Woehr. *NetBeans: the definitive guide*. " O'Reilly Media, Inc.", 2002.

[7] Sara Ford and Zain Naboulsi. *Coding Faster: Getting More Productive with Microsoft Visual Studio*. Pearson Education, 2011.

[8] Bram Moolenaar. The vim editor. *See website at http://www. vim. org*, 2008.

[9] Richard M Stallman. *Gnu Emacs Manual: For Version 22*. Free Software Foundation, 2007.

[10] Clements J. Flanagan C. Flatt M. Krishnamurthi S. Steckler P. & Felleisen M. Findler, R. B. DrScheme: A programming environment for Scheme. *Journal of functional programming*, 12(2):159–182, 2002.

[11] Flanagan C. Flatt M. Krishnamurthi S. & Felleisen M. Findler, R. B. DrScheme: A pedagogic programming environment for Scheme. *Programming Languages: Implementations, Logics, and Programs*, 12(2):369–388, 1997.

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

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

[14] Siddharta Govindaraj. *Test-Driven Python Development*. Packt Publishing Ltd, 2015.

[15] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. *ACM SIGPLAN Notices*, 46(10):119–138, 2011.

[16] Xi Ge, Quinton L DuBose, and Emerson Murphy-Hill. Reconciling manual and automatic refactoring. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 211–221. IEEE, 2012.