Refactoring Dynamic Languages

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

ABSTRACT

Typically, beginner programmers do not master the style rules of the programming language they are using and, frequently, do not have yet the logical agility to avoid writing redundant code. As a result, although their programs might be correct, they can also be improved and it is important for the programmer to learn about the improvements that, without changing the meaning of the program, simplify it or transform it to follow the style rules of the language. These kinds of transformations are the realm of refactoring tools. However, these tools are typically associated with sophisticated integrated development environments (IDEs) that are excessively complex for beginners. On the other hand, there are several different programming languages being used in introductory courses to teach beginner programmers and, it is not expected that the languages used in these introductory courses will converge into one single language in the near future.

In this thesis, we present a refactoring tool designed for beginner programmers, which we made available in DrRacket, a simple and pedagogical IDE. Our tool provides several refactoring operations for the typical mistakes made by beginners and is intended to be used as part of their learning process. We also present a framework designed to simplify the creation of refactoring tools for dynamic languages, which we evaluate by creating refactoring tools for Python and Racket.

Keywords

Refactoring Tool, Pedagogy, Framework, Racket

1. INTRODUCTION

In order to become a proficient programmer, one needs not only to master the syntax and semantics of a programming language, but also the style rules adopted in that language and, more important, the logical rules that allow him to write simple and understandable programs. Given that beginner programmers have insufficient knowledge about all these rules, it should not be surprising that their code reveals what more knowledgeable programmers call "poor style," or "bad smells". As time goes by, it is usually the case that the beginner programmer learns those rules and starts producing correct code written in an adequate style. However, the learning process might take a considerable amount of time and, as a result, large amounts of poorly-written code might be produced before the end of the process. It is then important to speed up this learning process by showing, from the early learning phases, how a poorly-written fragment of code can be improved.

After learning how to write code in a good style, programmers become critics of their own former code and, whenever they have to work with it again, they are tempted to take advantage of the opportunity to restructure it so that it conforms to the style rules and becomes easier to understand. However, in most cases, these modifications are done without complete knowledge of the requirements and constraints that were considered when the code was originally written and, as result, there is a serious risk that the modifications might introduce bugs. It is thus important to help the programmer in this task, so that he can be confident that the code improvements he anticipates are effectively applicable and will not change the meaning of the program. This has been the main goal of *code refactoring*.

Code refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [1]. Nowadays, any sophisticated IDE includes an assortment of refactoring tools, e.g., renaming variables or methods, for extracting methods, or for moving methods along a class hierarchy. It is important to note, however, that these IDEs were designed for advanced programmers, and that the provided refactorings require a level of code sophistication that is not present in the programs written by beginners. This may make the refactoring tools inaccessible to beginners.

In addition, since the choice of the language to teach in an introductory course often differs from faculty to faculty and has evolved throughout the years, there are several different programming languages being used to teach beginner programmers. Furthermore, even the requirements used to decide the language in the introductory courses have changed [2]. Considering all these factors, it is not expected that the languages used in these introductory courses will converge into one language in the near future.

In order to correctly refactor a program, refactoring tools often require the same kind of information about the program. Therefore they often have similar architecture. This similarity between refactoring tools creates the possibility of reusing some modules instead of creating every module from scratch, thus making the development of refactoring tools development faster. Grouping all those modules in a refactoring framework would make them more easily accessible Moreover, it is also possible to share features for the refactoring tools thus implemented, improving the support available for the users of those refactoring tools.

In this paper, we present a tool that was designed to ad-

dress the above problems. In particular, our tool (1) is usable from a pedagogical IDE designed for beginners [2, 3], (2) is capable of analyzing the programmer's code and inform him of the presence of the typical mistakes made by beginners, and finally, (3) can apply refactoring rules that restructure the program without changing its semantics.

We also present a framework designed for creating refactoring tools for dynamic languages. This Framework (4) helps the developer to create a new refactoring tool by providing functions to support the refactoring tool, (5) is capable of sharing the features once created for one of the refactoring tools, and finally (6) provides automatic tests to maintain the refactoring tools created.

To evaluate our proposal, we implemented a refactoring tool in DrRacket, a pedagogical IDE [4, 5] used in schools around the world to teach basic programming concepts and techniques. Currently, DrRacket has only one simple refactoring operation which allows renaming a variable. Our work significantly extends the set of refactoring operations available in DrRacket and promotes their use as part of the learning process. We also implemented a framework for creating refactoring tools in DrRacket which already have implemented several languages besides Racket, such as Python and Processing. [1] Our work significantly simplifies the creation of new refactoring tools and provides useful features to the developer of the refactoring tool.

2. RELATED WORK

The large majority of these tools were designed to deal with large statically-typed programming languages such a Java or C++ and are integrated in the complex IDEs typically used for the development of complex software projects, such as Eclipse or Visual Studio.

On the other hand, it is a common practice to start teaching beginner programmers using dynamically-typed programming languages, such as Scheme, Python, or Ruby, using simple IDEs. As a result, our focus was on the dynamic programming languages which are used in introductory courses and, particularly, those that promote a functional programming paradigm.

In the next sections, we present an overview of the refactoring tools that were developed for the languages used in introductory programming courses.

2.1 Scheme

In its now classical work [6], Griswold presented a refactoring tool for Scheme that uses two different kinds of information, namely, an AST and a PDG.

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, two semantically unrelated statements can be placed arbitrarily with respect to each other. Using the AST as the main representation of the program ensures that statements are not arbitrarily reordered, allowing the PDG to be used to prove that transformations preserve the meaning and as a quick way to retrieve needed dependence information. Additionally, contours are used with the PDG to provide scope information, which is non existent in the PDG, and 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 Python

Rope[7, 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 function calls. Thus, by limiting the complexity of the language it reduces the complexity of the refactoring tool.

Rope uses Static Object Analysis, which analyses the modules or scopes to get information about functions. Because its approach is time consuming, Rope only analyses the scopes when they change and it only analyses the modules when asked by the user.

Rope also uses Dynamic Object Analysis, requiring running the program in order to work. Dynamic Object Analysis gathers type information and parameters passed to and returned from functions. 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 it. This approach makes the program run much slower. Thus, it is only active when the user allows it. Rope uses an AST in order to store the syntax information about the programs.

Bicycle Repair Man[2] is another refactoring tool for Python and is written in Python itself. This refactoring tool can be added to IDEs and editors, such as Emacs, Vi, Eclipse, and Sublime Text. It attempts 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 uses an AST to represent the program and a database to store information about several program entities and their dependencies.

Pycharm Educational Edition,[3] or Pycharm Edu, is an IDE for Python created by JetBrains, the creator of IntelliJ. The IDE was specially designed for educational purposes, 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. It is very similar to their complete IDEs and it has interesting features such as code completion and integration with version control tools. However, it has a simpler interface than Pycharm Community[4] and other IDEs such as Eclipse or Visual Studio.

Pycharm Edu integrates a Python tutorial and supports teachers that want to create 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 same 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 the dependencies between the def-

[1] www.processing.org

[2] https://pypi.python.org/pypi/bicyclerepair/0.7.1

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

[4] https://www.jetbrains.com/pycharm/

Table 1: Data Structures

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

inition and the use of variables, known as def-use relations.

2.3 Javascript

There are few refactoring tools for JavaScript but there is a framework for refactoring JavaScript programs [8]. 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 produce 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.

2.4 Analysis

Table 1 summarizes the data structures of the analyzed refactoring tools. 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 contains mainly information about the def-use-relation of the program. The PDG has also control flow information of the program.

Some tools, like the one build by Griswold, focus on the correctness of the refactoring operations and therefore need more information about the program, such as the information provided by the PDG. Others, focus on 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 proving formally correct the refactoring operations or provide refactoring operations only used in advanced and complex use cases.

We intend to have simple, useful, and correct refactoring operations to correct the typical mistakes made by beginners. With this, we exclude from the refactoring tool scope, macros, classes, and other complex language structures not often used by beginners.

2.5 Language-independent Refactoring

Some refactoring operations make sense in different languages, such as the rename, the move or even the extract function. In order to use that similarity there are some tools that aim to create refactoring operations independently of the language.

Refactoring has been mainly applied to sequential programs in detriment of parallel ones, and to introduce and help fine tune parallelism parallel refactoring, a language independent parallel refactoring framework for C/C++ and Erlang [9] was proposed. This tool uses refactoring combined with parallel design patterns that will introduce parallelism into the programs to help the users creating better parallel programs.

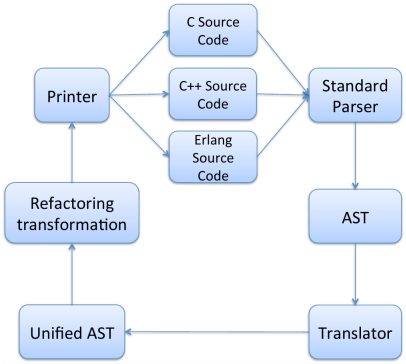

Figure 1: Parallel Refactorer architecture

Figure 2.5 describes an overview of the architecture of the system. It parses the user's code, in C/C++ or Erland, into an AST which is then translated into a unified intermediate language. This intermediate language is what allows to have a language independent refactoring tool. It is then refactored and then pretty printed into the program's source language.

3. ARCHITECTURE

In this section, we present the architecture of our refactoring tool, developed for the Racket programming language and, more specifically, for the DrRacket IDE.

Racket is a language designed to support meta-programming and, in fact, most of the syntax forms of the language are macro-expanded into combinations of simpler forms. This has the important consequence that programs can be analyzed either in their original form or in their expanded form.

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 links between definition of an identifier and its usage. In the DrRacket IDE, these relations are visually represented as arrows that point from a definition to its use. The opposite relation, the use-def relation, is also visually represented as an arrow from the use of an identifier to its definition. The AST is the abstract syntax tree of the program which, in the case of the Racket language, is represented by a list of syntax-objects.

Figure 2 summarizes the workflow of the refactoring tool, where the Reader produces the non expanded AST of the program while the Expander expands the AST produced by the Reader. In order to produce the def-use relations, it is necessary to use the expanded AST produced by the Expander because it has the correct dependency information. The Transformer uses the Code Walker to parse the ASTs and the information of the def-use relations to correctly perform the refactoring operations. Then it goes to the Writing module to produce the output in DrRacket's definitions pane.

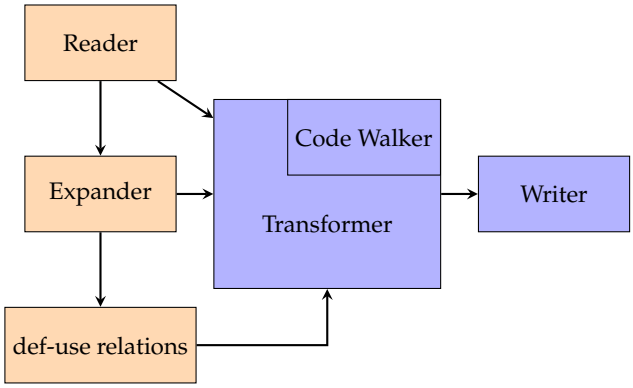

Figure 2: Main modules and information flow between modules. Arrows represent information flow between modules.

3.1 Syntax Expressions

The syntax-object list represents the AST, which provides information about the structure of the program. The syntax-object list 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 syntax-object list and uses some of those functions in the Background Check Syntax and in the Check Syntax button callback.

3.1.1 Syntax Expression tree forms

DrRacket provides functions to compute the syntax-object list in two different formats. One format is the expanded program, which 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 form, and in the non-expanded form.

Listing 1: Original Code

```
(and alpha beta)
```

Listing 2: Expanded program

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

Listing 3: Non-expanded program

```
#<syntax:2:0 (and alpha beta)>
```

Note that 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, is not necessary. 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. However, the expanded program has important information regarding the binding information that is not available in the non-expanded form, and this information might be useful, e.g., to detect if two identifiers refer to the same binding. Additionally, we do not consider macro definitions as part of the code that needs to be refactored, since the refactoring tool is targeted at unexperienced programmers and these programmers typically do not define macros.

Taking the previous discussion into consideration, it becomes clear that it is desirable to use the non-expanded form for the refactoring operations whenever possible and use the expanded form only when needed.

3.2 Def-use relations

Def-use relations hold important information needed in order to produce correct refactoring operations. They can be used to check whether there will be a duplicated name or to compute the arguments of a function that is going to be extracted.

Def-use-relations are computed by the compiler that runs in the background. However, they are only computed when a program is syntactically correct.

3.3 Code-walker

The code-walker is used to parse the syntax tree represented by a syntax element that is a list of syntax-objects in Racket. A syntax-object 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 used to search for the correct elements.

Most of the time, the code-walker is used to search for a specific syntax element and the location information contained in the syntax-object is used to skip the syntax blocks that appear 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 Syntax-parse

The syntax-parse [10] function provided by Racket is very useful for the refactoring operations. It provides a wide range of options to help matching the correct syntax, using backtracking to allow several rules to be matched in the same syntax parser, helping 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. The complete list of refactoring operations is available in Appendix **??**.

4.1 Semantic problems

There are some well-known semantic problems that might occur after doing a refactoring operation. One of them occurs

in the refactoring operation that removes redundant `and`s in numeric comparisons. Although rarely known by beginner programmers, in Racket, numeric comparisons support more than two arguments, as in `(< 0 ?x 9)`, meaning the same as `(and (< 0 ?x) (< ?x 9))`, where, we use the notation `?x` to represent an expression. Thus, it is natural to think about a refactoring operation that eliminates the `and`. However, when the `?x` expression somehow produces side-effects, the refactoring operation will change the meaning of the program.

Despite this problem, we support this refactoring operation because, in the vast majority of the cases, there are no side-effects being done in the middle of numerical comparisons, it is not often to have the same argument repeated in a comparison, and with the short-circuit evaluation there is no guarantee that the side effect will occur twice.

4.2 Extract Function

Extract function is an important refactoring operation that every refactoring tool should have. In order to extract a function, it is necessary to compute the arguments needed for the correct use of the function. While giving the name to a function seems quite straightforward, it is necessary to check for name duplication in order to produce a correct refactoring as having two identifiers with the same name in the same scope produces an incorrect program. After the previous checks, it is straightforward to compute the function body and replace the original expression with the function call.

However, the refactoring raises the problem of where should the function be extracted to. A function cannot be defined inside an expression, but it can be defined at the top-level or at any other level that is accessible from the current level.

As an example, consider the following program:

Listing 4: Extract function levels

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

When extracting `(+ 1 2)` to a function where should it be defined? Top-level, Level-0, level-1, or in the current level, the level-2? 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 intent. Accordingly, we decided that the best solution is to let the user decide where to define the function.

4.2.1 Computing the arguments

In order to compute the function call 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 extracted are candidates to be the arguments of that function. However, imported variables, whether from the language base or from other libraries, do not have to be passed as arguments. To solve this problem, 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 def-use relations 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 correctly compute the arguments of the new function.

4.3 Wide-Scope Replacement

The Wide-Scope replacement refactoring operation searches for the code that is the duplicate of the extracted function and then replaces it 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 need to compute the arguments for the duplicated code again.

Correctly detecting duplicated code is a key part for the correctness of this refactoring. Even the simplest form of duplicated code detection, where it only detects duplicated code when the code is exactly equal, may have some problems regarding the binding information. For example, if the duplicated code is inside a `let` that changes some bindings, that must be taken into consideration. We decided to use AST comparison to detect clones because it was the representation of the program used by our refactoring tool and it has very high precision, but currently has considerably higher costs in terms of execution time [11]. In order to solve the binding problem we can use functions already provided in Racket. However, that does not work if we use the program in the non-expanded form to do the binding comparisons 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 to the user. While this does not change the detection of the duplicated code, it adds unnecessary information that would have to be removed. In order to solve this, in a simple way, we can use the expanded code to correctly detect duplicated code and use the non-expanded program to compute which code will be replaced. The duplicated code detection is a quadratic algorithm which might have some performance problems for larger programs, however, for the programs typically written by beginners this is not a problem.

4.4 Features

This section describes some of the features created to improve the usability by providing sufficient feedback to the user, and way to inform the user of the presence of the typical mistakes made by beginners.

4.4.1 User Feedback

It is important to give proper feedback to the user while the user is attempting or preforming a refactoring operation. 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. It works by applying the refactoring operation in a copy of the AST and displaying those changes to the user.

4.4.2 Automatic Suggestions

Beginner programmers usually do not know which refactoring operations exist or which can be applied. By having an automatic suggestions of the possible refactoring operations available, beginner programmers can have an idea what refactoring operations can be applied.

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. In other words, it uses brute force to check whether an 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. Moreover, 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. FRAMEWORK

Refactoring tools often share a similar architecture since they usually require the same information about the program. They often require program information such as the AST and the def-use relations, in order to correctly reason upon the program. The use of the same program's information creates an architectural similarity between refactoring tools, which raises the possibility of reusing some modules instead of creating every module from scratch. Thus, by reusing those components that provide the program's information that refactoring tools need we are speeding up the refactoring tool development process. Therefore, a framework for creating refactoring tools increases the development speed and simplifies the development of a refactoring tool. In addition, our framework is able to reuse the features already available for the developed refactoring tools without extra implementation effort. Such features highly improve the utility of a refactoring tool, for example the highlight of possible refactoring operations, the previewing of the result and support to detect duplicated code.

Ideally, in order to achieve maximum re-usability, our framework should only have one implementation of each refactoring operation. This can be accomplishes by having a meta-language that represents all the languages supported by the refactoring tool combined with specialized pretty printers to output in the desirable language. However, such general representation would be too complex or, in order to keep it manageable it would reduce the language expressiveness. Instead of having a general representation that can represent all the supported languages and only have one implementation per refactoring, we have language-specific refactoring operations. In other words, we have language-dependent refactoring operations.

Nevertheless, we also have a meta-language that abstracts all the supported languages in order to have language-independent refactoring operations. However, it is only for a small set of refactoring operations, the simpler ones.

With this combination between a meta-language that has language-independent refactoring operations and language-specific refactoring operations we do not have a purely language-independent framework, but instead, we have a framework for creating refactoring tools for dynamic languages.

We decided to implement the framework in DrRacket since it were already some programming languages implemented which simplified development of the framework.

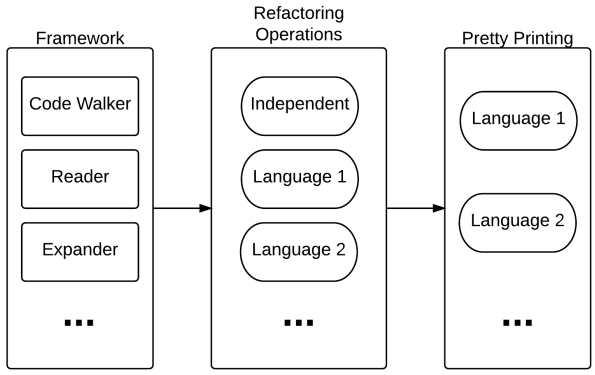

Figure 3: Framework Architecture

As described in Figure 5, we have a module that provides language-independent refactoring operations and a module for each supported language that has specialized refactoring operations. We also have pretty-printing modules for each supported language.

We already have refactoring operations for Racket, Python, and Processing. We selected these languages since they are already implemented in Racket and they are commonly used by beginners.

5.1 Language-independent refactoring operations

Creating a fully language-independent refactoring tool is rather complex since programming languages are semantically different from each other, and there are even some operations that are semantically equal for most of the cases, but not for all the cases, making it very difficult to do a general refactoring operation. However, for simple refactoring operations, which do not require much program semantics, it is possible to have language-independent refactoring operations in a meta-language that represents the semantics of the other implemented languages.

5.2 Language-dependent refactoring operations

It is necessary to have refactoring operations that are language-dependent, since some refactoring operations have particular cases for each supported language. By having a specific refactoring operations, language dependent ones, it is possible to implement some useful refactoring operations in a simple way.

5.3 Analysis

Combining the two approaches allows for the simple creation of refactoring operations for several languages targeted at beginner programmers. Some of these refactoring operations are rather simple, when compared to more advanced ones, and can even be reproduced in a meta-language and, therefore, implemented only once for several programming languages. For the rest of the refactoring operations, they can be added to the dependent language module in which it is possible to have the special cases covered.

With the combination usage of reader + expander + code-walker working for the several implemented languages, most of the work is already done and, when combined with the syntax-parser, it is possible to create refactoring operations using only one line of code, e.g. `[(p-not (p-gt a b)) #'(p-le a b)]`.

Even having some refactoring operations that are more complex it is way quicker and simpler to only have to implement the refactoring operation itself.

6. LANGUAGES

In this section, we present the languages, besides Racket, that have refactoring tools developed in our framework. We start by presenting Python, then we present Processing.

6.1 Python

Python is being promoted as a good replacement for Scheme and Racket in science introductory courses. It is an high-level, dynamically typed programming language and it supports the functional, imperative and object-oriented paradigms. Using the architecture of this framework and the capabilities offered by Racket, combined with an implementation of Python for Racket, called PyonR [12] [13], it is also possible to provide refactoring operations in Python.
Using Racket's syntax-objects to represent Python as a meta-language [14], it is possible to use the same structure used for the refactoring operations in Racket to parse and analyze the code in Python.

However, there are some limitations regarding the refactoring operations in Python. Since Python is a statement-based language instead of expression base, it raises some problems regarding what the refactoring operation has to do or even the possibility of some refactoring operations. For example, when extracting a function in a expression-based language it is not necessary to compute the location of the `return` expression, whereas in statement-based languages like Python it is necessary to compute if it is necessary to have the keyword `return` and where.

The following example shows on refactoring operation we can perform in Python.

Example of removing an If expression:

```
1 True if (alpha < beta) else False
```

```
1 (alpha < beta)
```

6.2 Processing

Processing [15] is a programming language based on the Java programming language, but with a simplified syntax and graphics programming model.

Even though Processing is a static programming language similar to Java, using P2R [16], an implementation of Processing for Racket, makes it possible to provide some refactoring operations in Processing.

The refactoring operations for Processing are highly limited since they do not take into account the type information of the language present in the program's AST, therefore, making them less powerful. It is important to note that this framework was developed for creating refactoring tools for dynamic languages. Thus, the refactoring tool for Processing is only a proof of concept to show the flexibility of the created framework. To fully support static languages, it is necessary to gather and manipulate type information.

The following example shows some of the refactoring operations we can perform in Processing.

```
1 boolean example = !(alpha < beta);
```

It removes unnecessary Nots of the expression.

```
1 boolean example = (alpha >= beta);
```

6.3 Meta-Language Refactorings

The meta-language is what allows the framework to provide language-independent refactoring operations. Since we did not want to add additional conditions to the refactoring operations in the meta-language, we only provide some basic refactoring operations. Nevertheless, in addition to these operations being simple, they can be useful as well.

We take advantage of the similarities between the implementations of Python and Processing to Racket syntax-objects and we abstract them to our meta-language. For example, the following code represents `!(a>b)` in processing:

```
1 (p-not (p-gt a b))
```

While the following code represents `not(a>b)` in python:

```
1 (py-not (py-gt a b))
```

With such similarities in the language implementation to Racket syntax-objects, it is straightforward to abstract those languages to a meta-language.

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 use of some of the refactoring operations previously presented and here is explained the motivation for their existence.

The first example is a very typical error made by beginner programmers.

```
(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-movement? 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 makes the code cleaner and simpler to understand.

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

However, this code can 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 in the initial learning phases, it is common to forget whether or not a sequence of expressions need to be wrapped in a `begin` form. The `when`, `cond` and `let` expressions have a implicit `begin` and as a result it is not necessary to add the `begin` expression. Nevertheless, sometimes students still keep the `begin` keyword because they often use a trial and error approach in writing code. Our refactoring tool checks for those mistakes and corrects them.

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

This is a simple refactoring operation and does not have a big impact. However, it makes the code clearer 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))))
      (and (integer? internal-column)
           internal-column))
    (let ((internal-column (/ (sub1 column) 2)))
      (and (integer? internal-column)
           internal-column)))
```

The next example shows a nested `if`. Nested `if`s are difficult to understand and error prone.

```
(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)))))))))
```

It is much simpler to have a `cond` expression instead of the nested `if`. In addition, every true branch of this nested if contains `if` expressions that are or expressions and by refactoring those `if` expressions to `or`s it makes the code simpler to understand.

```
(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]))
```

However, this code could still be further improved by refactoring it into a `case`.

The examples presented above appear repeatedly in almost every code submission of this final project supports the need to provide a better support to beginner programmers.

Table 2 shows he average reduction in lines of code (LOC) is 10.63%, which shows how useful these refactoring operations are. It also shows how many refactoring operations

Table 2: Refactoring Operations

Code #	1	2	3	4	5	6	7	8	Total
Initial LOC	582	424	332	1328	810	569	798	614	4045
Final LOC	545	373	300	1259	701	527	733	457	3705
Difference	37	51	32	69	109	42	65	140	340
Percentage	-6.36	-12.03	-9.64	-5.19	-13.46	-7.38	-8.14	-22.80	-10.63
Remove Begin	11	4	6	9	5	2	0	7	44
If to When	4	0	0	0	0	7	0	0	11
If to And	3	1	0	0	0	2	0	0	6
If to Or	6	6	1	13	20	3	2	0	51
Remove If	0	3	7	6	3	5	0	2	26
Remove And	0	2	0	4	0	0	0	0	6
Remove Eq	0	4	0	0	0	0	0	0	4
Extract function	0	3	0	0	4	0	1	5	13
If to Cond	0	0	0	2	3	0	1	0	6

were applied. This tool is not only for beginners: during the development of the tool we already used some of the refactoring operations, namely the extract function, in order to improve the structure of the code.

8. FRAMEWORK EVALUATION

In this section we evaluate the framework, the simplicity of creating a refactoring tool with the reusability of features already existents tools provided to help the maintainability of the refactoring tools developed.

8.1 Simplicity

The following piece of code is the function that implements the refactoring operations available for Processing.

```
1  (define (processing-parser1 arg)
2    (syntax-parse arg
3      #:datum-literals (p-not p-and p-or p-lt p-gt p-le p-ge)
4      [(p-not (p-gt a b)) #'(p-le a b)]
5      [(p-not (p-le a b)) #'(p-gt a b)]
6      [(p-not (p-lt a b)) #'(p-ge a b)]
7      [(p-not (p-ge a b)) #'(p-lt a b)]
8      [_ (void)]))
```

It is quite simple to create refactoring operations in the framework. With simple rules, like the ones presented above, it is possible to create simple, but useful refactoring operations.

8.2 Reusability

We can share features such as the automatic suggestion of refactoring operations and the preview of the outcome of such refactorings, like already created for the Racket refactoring tool, for Processing and Python.

8.2.1 Highlight

In this section we show and explain what was needed to improve in order to re-use the highlight feature initial developed for the Racket refactoring tool.

Figure 4 shows, the automatic suggesting of the detected refactoring opportunities working for the Python refactoring tool. The main difference for the original is that the highlighting now has to use the pretty printing to compute the size of the region where the refactoring opportunities occurs. This difference, a cosmetic one, happens because the syntax-objects for the Racket refactoring tool have the same length to be highlighted as the program's code, whereas the Python's representation syntax-objects do not. Figure 4 highlights the refactoring operation that removes unnecessary if conditions.

```
1  #lang python
2
3  True if (a < b) else False
4
5
6
7
```

Figure 4: Highlight in Python

8.2.2 Preview

In this section we show and explain what was needed to improve in order to re-use the preview feature initial developed for the Racket refactoring tool.

Figure **??** shows, the preview of the outcome of the refactoring operation works for the Processing refactoring tool. The main difference for the original is that the preview now, like the highlight feature, has to use the pretty printing to compute the size of the region where the refactoring opportunities occurs.Figure **??** shows the preview of the outcome of the refactoring operation that removes unnecessary not conditions.

```
1  #lang processing
2
3
4  boolean example = !(a < b);
5
6                    Refactoring Operations  (a >= b)
7                    Undo
```

Figure 5: Preview in Processing

8.3 Maintainability

The provided automatic testing tool can help the developer maintaining the developed refactoring tools. The tests run at the startup of the IDE, providing feedback to the developer check the correct running of the refactoring tool and the test case that is failing.

Figure 6 shows an image of an error in the Python refactoring tool, the error is from the test case: removeIf.

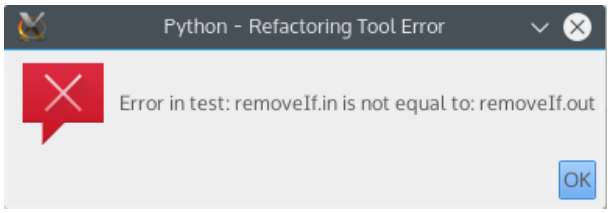

Figure 6: Error in Python refactoring tool

9. CONCLUSION

A refactoring tool designed for beginner programmers would benefit them by providing a tool to restructure the programs and improve what more knowledgeable programmers call "poor style" or "bad smells". In order to help those users the refactoring tool must be usable from a pedagogical IDE, to inform the programmer of the presence of the typical mistakes made by beginners, and to correctly apply refactoring operations preserving semantics.

Our solution tries to help those users to improve their programs by using the AST of the program and the def-use-relations to create refactoring operations that do not change the program's semantics. This structure is then used to analyze the code to detect typical mistakes using automatic suggestions and correct them using the refactoring operations provided.

There are still some improvements that we consider important for the user. Firstly, the detection of duplicated code is still very naive and improving the detection in order 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. Then, it is possible to further improve the automatic suggestion of refactoring operations by having different colors for different types of refactoring operations, with a low intensity for low "priority" refactoring operations and a higher 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. Lastly it would be interesting to detect when a developer is refactoring in order to help the developer finish the refactoring by doing it automatically [17].

A framework for creating refactoring tools for dynamic languages would benefit the refactoring tool developers by simplifying their work and would indirectly benefit its users since the developer would have more time to spend improving the refactoring tool. In order to be helpful, the framework must be simple to use by the developers, must facilitate the reuse of the already implemented features, and must provide maintainability support to the refactoring tools.

Our framework tries to help developers to create refactoring tools. We achieve that by providing modules to access program information such as the AST or the def-use-relations. Combined with the provided tools to expand and process the AST it drastically simplifies the refactoring tool creation. We also created a prototype architecture to have simple language-independent refactoring operations in the framework. These simple refactoring operations can be helpful for the user for those languages that the user does not have refactoring tools and wants to continue use the same IDE.

The framework would be improved with a better program's information support, improved feedback, and a better decoupling in the framework architecture.

Firstly, having PDG support will allow the framework user to develop more useful and complex refactoring operations for statement-based languages such as Python. In addition, it would also be interesting to add type information support to the framework since Racket has already a typed racket [10] which would allow a refactoring tool for typed racket.

10. REFERENCES

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

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

[4] 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.

[5] 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.

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

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

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

[9] Christopher Brown, Kevin Hammond, Marco Danelutto, and Peter Kilpatrick. A language-independent parallel refactoring framework. In *Proceedings of the Fifth Workshop on Refactoring Tools*, pages 54–58. ACM, 2012.

[10] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.

[11] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *Software Engineering, IEEE Transactions on*, 33(9):577–591, 2007.

[12] Pedro Palma Ramos and António Menezes Leitão. An implementation of python for racket. In *7 th European Lisp Symposium*, 2014.

[13] Pedro Palma Ramos and António Menezes Leitão. Reaching python from racket. In *Proceedings of ILC 2014 on 8th International Lisp Conference*, page 32. ACM, 2014.

[14] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 154–164. IEEE, 2000.

[15] Casey Reas and Ben Fry. *Processing: a programming handbook for visual designers and artists*, volume 6812. Mit Press, 2007.

[16] Hugo Correia and António Menezes Leitão. Combining processing with racket. In *Languages, Applications and Technologies*, pages 101–112. Springer, 2015.

[17] 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.