Renaming For Everyone

Language-Parametric Renaming in Spoofax

Philippe D. Misteli

Renaming For Everyone

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Philippe D. Misteli
born in Vienna, Austria

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: Name tag.

Renaming For Everyone

Author: Philippe D. Misteli
Student id: 4932129

Abstract

A refactoring is a program transformation that improves the design of the source code, while preserving its behavior. Most modern IDEs offer a number of automated refactorings as editor services. The Rename refactoring is the most-commonly applied refactoring and is used to change the identifier of a program entity such as a variable, a function, or a type.

Correctly implementing refactorings is notoriously complex and these state-of-the-art implementations are known to be faulty and too restrictive. When developing a new programming language, it is both difficult and time-consuming to implement sound and complete automated refactoring transformations. Language-parametric definitions of refactorings that can be reused by instantiation with the syntax and semantics of a language, allow the development effort of refactorings to be amortized across language implementations.

In this thesis, we developed a language-parametric Rename refactoring algorithm that works on an abstract model of a program's name binding structure. We implemented the algorithm in the Spoofax language workbench, building on the language-parametric representation of name binding with scope graphs and using generic traversals in the Stratego transformation language. We evaluated the algorithm with five different languages implemented in Spoofax, which uses both NaBL2 and Statix to declare their static semantics and name binding rules. As a result, Spoofax now provides an automated Rename refactoring that works for any language developed with the language workbench using NaBL2/Statix.

Thesis Committee:

Chair:	Prof. Dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Dr. M. Finavaro Aniche, Faculty EEMCS, TU Delft
Committee Member:	A. S. Zwaan, Faculty EEMCS, TU Delft
Daily Supervisor:	D. A. A. Pelsmaeker, Faculty EEMCS, TU Delft

Preface

I would like to thank my supervisor Daniel for the professional collaboration and his meticulous feedback on my work. I would like to thank Eelco for his patronage and guidance on my way to complete my master's degree. In the same manner, I would like to thank Hendrik, Aron, Gabriel and Jeff for lending me their expertise to support my research.

I would like to thank my parents for their unconditional support of my academic pursuits. I would also like to give thanks to my best friend Joel for giving me the motivation to finish this thesis in time. Lastly, I would like to thank my amazing wife Brittany for always believing in me and pushing me to keep going, even if I could not see the way forward.

Philippe D. Misteli
Delft, the Netherlands
May 17, 2021

For Melia, my true master piece

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A refactoring is a set of transformations on the source code of a program that changes its structure without affecting its observable behavior. The main goal of refactoring a program is to improve its code quality and thus, its maintainability. The term was coined by Martin Fowler [20] in his well-known book "Refactoring - Improving the Design of Existing Code". Performing refactorings is an essential skill for every software developer.

Numerous refactorings of various complexity exist for all kinds of programming languages. For example, moving a field from a class to its parent class in Java, inlining a local variable in C, or extracting a function from an expression in Haskell. The Rename refactoring, the most used refactoring across programming languages [47], changes all name occurrences of a name identifying a program entity, such as a variable or a type.

Executing the steps necessary for a refactoring by hand is laborious and prone to introducing errors into the code. Manual refactorings also require the developer to re-test the program after completing the code modification, to ensure that the behaviour is still the same. Clearly, it would be advantageous to have tool support for refactoring, or as Erich Gamma puts it: "To avoid digging your own grave, refactoring must be done systematically." [20].

Modern IDEs, like IntelliJ IDEA or Visual Studio, offer automated refactorings as editor services. Automated refactorings should give a programmer the guarantee that the transformation is behaviour preserving and therefore, should remove the need to test the program after the change. It is also significantly faster than doing it by hand. This allows the refactorings to be integrated more dynamically into the overall software development workflow.

While providing refactorings as an editor feature makes applying them convenient and easy, the transformations themselves are notoriously complex and time-consuming to develop. When not implemented correctly, they can introduce bugs into the code that are difficult to detect. For example, through inadvertent name capture or by accidentally introducing multiple evaluation of expressions with side effects. In fact, the refactoring engines of many popular IDEs have been shown to be both flawed and too restrictive [14, 38, 37].

In this paper we focus on the Rename refactoring, which allows a programmer to rename an identifier and all its related occurrences to be renamed automatically. This may seem to be a trivial operation, which might be performed using a simple textual search-and-replace edit. However, not all occurrences of a name necessarily correspond to the entity to be renamed. Furthermore, renaming a program entity can alter the behaviour of a program if it leads to name capture. Name capture occurs when a reference inadvertently resolves to a different declaration after the renaming which can unforeseeably change the output of a program. Thus, a sound implementation of Rename refactoring must preserve a program's name binding structure.

How names are bound in a program differs substantially across programming languages. Since name binding patterns such as, lexical scoping or qualified names, are part of a language's static semantics, most renaming algorithms only work on programs of one specific

programming language. There exists sound renaming algorithm for almost all established languages such as Java [44], Scheme [24] or Smalltalk [43]. However, their implementations are not reusable across programming languages.

In this thesis we develop a language-parametric Rename refactoring that preserves a program's static semantics. Language-parametric in this context means that the implementation of the rename refactoring takes the name binding rules of the programming language as a parameter. Given a uniform representation of abstract syntax and of name binding across languages, the algorithm generically transforms a program and verifies the absence of name capture using the detection method of the name-fix algorithm [18].

We develop our language parametric renaming algorithm in the context of the Spoofax Language Workbench [28], a platform for developing (domain-specific) programming languages. The syntax and semantics of a language are defined through declarative specifications in meta-languages. Based on these specifications Spoofax generates a parser, type checker, and interpreter/compiler for a language.

The platform also offers features to generate common editor services such as, syntax highlighting and code completion. In this work, we extend Spoofax with a generic Rename refactoring that can be applied in any language developed with Spoofax using its generic name binding approach. Having this editor feature available out of the box, when developing a new DSL would arguably be a welcome commodity for language engineers. This new feature should turn Spoofax into an improved and more complete tool to research and develop programming languages.

In Spoofax, a programming language's name binding rules can be specified through the meta-languages NaBL2 [1] and its successor Statix [2]. Based on such a specification, the automatically generated type checker constructs a scope graph [39], which is a language-independent representation of a program's name binding structure. Both NaBL2 and Statix come with a deterministic name resolution algorithm that allows to find the declaration(s) to which a reference points that works on these graph representations.

We implement the Rename transformation using generic traversals and term rewrite rules in the Stratego transformation language [9, 10], another meta-language provided by Spoofax. From these rewrite rules, we can query the scope graph to gather the name binding information of a term.

To evaluate the completeness of our Rename refactoring, we tested the transformations on several languages, featuring a variety of name binding patterns including let-bindings in ML, qualified names in Java, and overloaded functions in C#. We implemented regression tests in the SPT [27] testing meta-language which executes the renaming algorithm on code fragments of the test languages in order to support the integrity of our evaluation. The refactoring is packaged as an interactive editor service that is integrated into the IDEs generated by Spoofax, providing an adequate user experience.

1.1 Contributions

The technical contributions of this thesis are the following:

- We develop a renaming algorithm that is parameterized with a name resolution algorithm and that is safe for name capture.
- We implement the renaming algorithm as an editor service in the Spoofax language workbench for languages defined with the Statix or NaBL2 static semantics specification languages. We implement the renaming transformation using a transformation that is generic in the abstract syntax structure and takes binding information as a parameter. The implementation supports multi-file transformations and is layout-preserving.
- We evaluate the renaming algorithm with respect to correctness, performance, and scalability on five programming languages which supports a variety of name binding patterns.
- We extend the SPT language with the capability to run the renaming implementation.

1.2 Outline

We structured this master thesis as follows. In Chapter 2, we describe the common problems and challenges when implementing the Rename refactoring. In Chapter 3, we present our language-parametric renaming algorithm and describe it step by step. Chapter 4 contains the details of how we implemented the Rename refactoring for the Spoofax language workbench. We tested our implementation on five different languages and report on those tests in Chapter 5. We evaluate our renaming solution in Chapter 6. The related work we discuss in Chapter 7. We outline future work and conclude this thesis in Chapter 8.

Chapter 2

On the Difficulty of Renaming

2.1 The Rename Refactoring

Choosing the right name for a program entity such as, a variable or a function can be tricky to get right on the first try. A proper name should accurately abstract the entity it represents whilst being both concise and comprehensive. Long enough to describe what it stands for but not too long to be easily remembered. It needs to follow language, organization and project conventions. The name needs to describe the same thing consistently across source code files but avoid introducing ambiguity [35].

The difficulty in this seemingly simple matter is reflected by the popularity of Phil Karlton's quote "There are only two difficult problems in computer science: cache invalidation, naming things, and the one-off error" [21]. Even if the name was expressive at first, a change in its context can make it ambiguous or misleading. And of course, there are always typos to consider. Given the complexity of the naming problem, changing a name in a computer program is a rather common operation.

Consider applying the refactoring Rename Field in Java: we want to rename the field `ctr` in Listing 2.1 to `counter`, resulting in the program in Listing 2.2. In order to do so, we need to change the name of the field's declaration on line 2 and both references that point to it on lines 4 and 7.

The process of renaming program entities is captured by the Rename refactoring [20]. In practice, there is a whole family of Rename refactorings that concern themselves with renaming different program entities, such as Rename Typeclass in Haskell or Rename Method in C#. The Rename refactoring is arguably the most basic refactoring and also the most-used one by far [47]. It can be applied to almost all programming languages.

```
1 class Counter {
2     int ctr = 0;
3     void inc() {
4         ctr++;
5     }
6     int read() {
7         return ctr;
8     }
9 }
```

Listing 2.1: Before Renaming the Field `ctr` in Java

```
1 class Counter {
2     int counter = 0;
3     void inc() {
4         counter++;
5     }
6     int read() {
7         return counter;
8     }
9 }
```

Listing 2.2: After Renaming the Field `ctr` to `counter` in Java

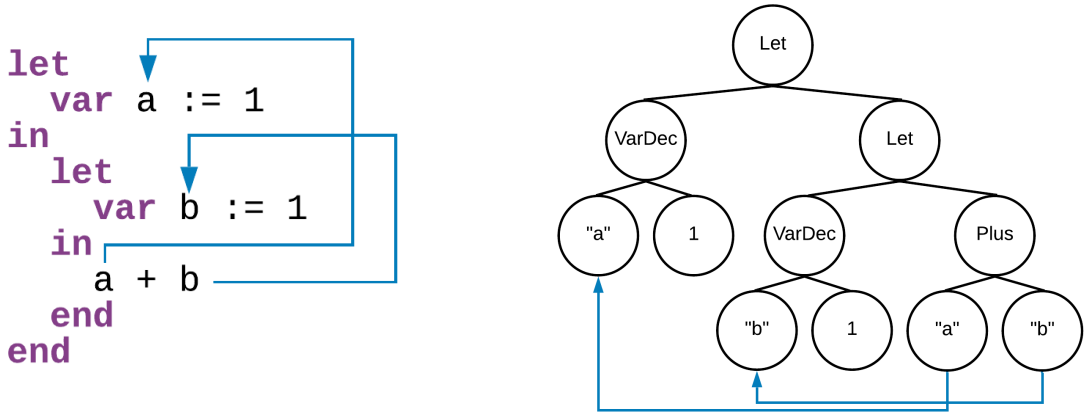
The main challenge in performing a Rename refactoring is to find all the name occurrences that belong to the same program entity. An occurrence in this context can either be

the declaration of an entity or a reference to that entity. This is rather trivial in the simple example above and could easily be done by hand. In this chapter we consider the problems we encounter when renaming more complex programs, which prompt the automation of the renaming process.

2.2 Basics of Name Binding

Naming is a powerful mnemonic tool for designing and implementing computer programs. It allows a developer to give a declaration of a *program entity* (such as a variable, a function, or a class) an alphanumeric identifier that can be referenced from another locations within the program. The procedure of finding the declaration to which a reference points to is called, *name resolution*. How names are bound to declarations and from where in a program references can access these declarations is an aspect of a programming language's static semantics, referred to as *name binding*.

Names are part of almost all computer programs but, there exists a large variety of *name binding patterns* across programming languages. The most prevalent binding pattern is lexical scoping, which makes a name accessible within the program region it was defined. An example program showing this pattern can be seen in Figure 2.1a, where the blue arrows point from reference to declaration. Non-lexical bindings relate names to non-hierarchically related occurrences, such as declarations in other modules.



(a) Source Code

(b) AST with Name Binding Edges

Figure 2.1: Example Tiger Program with Lexical Scoping

We define *scope* as a program region that behaves uniformly with respect to name resolution [39]. In the example program, the variable `a` is defined in the scope of the outer let binding and the variable `b` is defined in the scope of the inner let binding. Variable `b`'s declaration is accessible from the inner let binding because nested program regions have a parent relationship to the scope of the surrounding region in lexical scoping.

While names are such a pervasive feature in programming languages that are just taken for granted, they add a remarkable complexity to a language's static semantics. The first step in the compilation process of a program is usually to parse the textual interpretation into an abstract syntax tree (AST) according to the grammar of the language. Trees offer a well-suited interface for program transformations that happen in subsequent compilation steps, such as optimization or code generation. Adding edges from reference to declaration nodes to the AST effectively turn the data structure into a graph, complicating those transformations. The AST for the program shown in Figure 2.1 can be seen in Figure 2.1b, where the name binding edges are depicted as blue arrows.

2.3 Categories of Name Binding Patterns

What makes developing a language-parametric renaming algorithm so difficult is the variety of name binding patterns that exist across programming languages. In this section, we attempt to identify name binding pattern categories that our renaming algorithm needs to handle correctly.

2.3.1 Single-Declaration Binding

Most name binding patterns allow for a reference to point to a single declaration. We have encountered such a pattern already in Figure 2.1a, where a reference to a local variable always points to exactly one variable declaration.

The name binding pattern of fields in Java also falls into this category. In Listing 2.1, the two field references on line 4 and 7 both resolve to the single field declaration on line 2. In essence, name binding patterns in this category impose a many-to-one cardinality on the relationship between references and declarations.

2.3.2 Multi-Declaration Binding

A more complex category of name binding patterns allow to define a many-to-many relationship between references and declarations. The dynamic polymorphism supported in C# employs such a pattern. It allows child classes to declare a method that has the same signature as a method declared in a parent class, a pattern known as method overriding.

```
1  class Parent {
2    virtual String getName() {
3      return "Parent";
4    }
5  }
6
7  class Child : Parent {
8    override String getName() {
9      return "Child";
10   }
11 }
```

Listing 2.3: Class Hierarchy in C#

```
1 class Main {
2   static void logName(Parent p) {
3     Console.WriteLine(p.getName();
4   }
5 }
```

Listing 2.4: Dynamic Dispatch in C#

In Listing 2.3, the `getName` method declaration inside the `Child` class overrides the identical method declaration from the `Parent` class. Now on line 3 in Listing 2.4, there is a call to a `getName` method on an object of type `Parent`. Due to polymorphism, it is not possible to resolve the call to a single declaration at development time, as the type of object `p` can only be determined at runtime.

Since renaming happens at compile time, we can only rely on the result of the static name binding analysis. Therefore when renaming a call to an overridden method, all declarations that the call might resolve to needs to be changed.

2.4 Informal Name Binding

So far we have concerned ourselves with *formal name binding*, which is encoded into a language's static semantics and is enforced by the type checker. However, there are forms of *informal name binding* which we need to consider when developing our solution.

```
1  class Employee {
2    private:
3      int salary;
4    public:
5      void setSalary(int s) {
6        salary = s;
7      }
8      int getSalary() {
9        return salary;
10     }
11 };
```

Listing 2.5: Getter & Setter in C++

```
1  public class Calculator {
2    /**
3     * This method adds integers.
4     * @param a first number
5     * @param b second number
6     * @return int sum of a and b.
7     */
8    public int addNum(int a, int b) {
9      return a + b;
10   }
11 }
```

Listing 2.6: Javadoc

2.4.1 Binding between Entities

Formal name binding relates any name occurrence to a specific program entity. Relations between different kinds of entities are usually also somehow reflected in the name of the entities, but are usually not checked statically. A prominent example in object-oriented programs is the use of getters and setters to achieve encapsulation.

In the C++ program in Listing 2.5, we see that the name of the getter and setter method are directly dependent on the name of the field they encapsulate. Therefore when renaming the field `salary`, the methods `getSalary` and `setSalary` should also be renamed, in order not to break the naming convention.

2.4.2 Names in Comments

Most programming languages allow for developers to include comments to further explain or document their code. It is common for these comments to contain the names of the program entities they describe. The renaming transformation would ideally also change the occurrences in the comment. The JavaDoc comment in Listing 2.6 contains references to the function arguments `numA` and `numB`.

The main difficulty with identifying names in comments is that parsers usually completely ignore comments and thus, they are not represented in the AST. Therefore, the transformation either needs to work on the textual representation of the program or the comments need to become part of the AST.

Another issue with finding occurrences in comments is the fact that there are little syntactic restrictions on their content and no context available apart from its location in the source code. This makes it hard to link an occurrence inside a comment to an entity with certainty, especially in the presence of ambiguous names.

2.5 Ambiguity

The principal problem with renaming is the existence of ambiguous names, i.e. scenarios in which the same name is used to identify multiple entities in a program. For example, in Java it is allowed to give a field and a method the same name inside the same class. Similarly, a local variable declared in a method body may shadow a method parameter with the same name. Such lexical ambiguity precludes the application of textual find-and-replace to renaming, since *all* occurrences in the text would be replaced.

```
1  let
2    type foo = {
3      foo : string
4    }
5    function foo (foo: foo) = (
6      let
7        var foo := foo.foo
8      in
9        print(foo)
10     end
11   )
12 in
13   foo(foo{foo = "foo"})
14 end
```

Listing 2.7: The Foo Challenge in Tiger

```
1  let
2    type bar = {
3      foo : string
4    }
5    function foo (foo: bar) = (
6      let
7        var foo := foo.foo
8      in
9        print(foo)
10     end
11   )
12 in
13   foo(bar{foo = "foo"})
14 end
```

Listing 2.8: Renaming Type foo to bar

A more complex instance of this problem can be seen in Listing 2.7. While this is arguably an extreme and artificial case, we constructed this program to serve as a baseline benchmark and dubbed it the *Foo Challenge*. *Given an occurrence, a sound renaming algorithm should only change the occurrences that belong to the same entity*. There are five entities in the Foo Challenge program:

1. The type foo declared on line 2
2. The field foo inside the type foo declared on line 3
3. The function foo declared on line 5
4. The function argument foo of the function foo declared on line 5
5. The local variable foo inside the function body declared on line 7

A correct renaming of type `foo` would change the declaration on line 2, the reference occurrence that gives the function argument its type on line 5, and the reference occurrence on line 13 that is used to construct an instance of the type.

The Foo Challenge is written in the Tiger programming language. Tiger is a functional, statically-typed language developed to educate students about compiler construction [3]. We have used Tiger as our primary target language during the development and implementation of our language-parametric refactoring.

2.6 Name Capture

Name capture occurs, when a renaming unintentionally changes the name binding structure of a program. It happens when a renaming introduces a declaration with an ambiguous name that *captures* a reference, which used to point to another declaration with the same name as the new name of the renamed entity.

```
1 let
2   var bar := 10
3 in
4   let
5     var foo := 100
6   in
7     bar + foo
8   end
9 end
```

Listing 2.9: Capture-Prone Tiger Program

```
1 let
2   var bar := 10
3 in
4   let
5     var bar := 100
6   in
7     bar + bar
8   end
9 end
```

Listing 2.10: Tiger Program with Capture

This is easier to understand when looking at an example. In the program shown in Listing 2.9, we rename the local variable `foo` to `bar`. The output of the program before this renaming is 110. However, after the renaming the output is 200. This happens because the reference to `bar` on the left side of the plus operator is captured by the declaration of the local variable formerly named `foo`, see Listing 2.10.

Since refactorings are defined as *behaviour-preserving* transformations, capture should be avoided. Ergo *a sound renaming transformation should prevent name capture.* What makes capture so hard to detect is that the new version of the program will still pass all static name binding checks. Fortunately, we can detect capture statically [18], a technique we will employ in the next chapter.

2.7 Multi-file Programs

The examples we have looked at so far were all confined to a single *module*. However most meaningful programs consists of multiple modules, such as classes in Java, and occurrences are spread across these modules. Usually, each module is persisted in its own *file* and the renaming transformations need to correctly deal with programs that are partitioned in such a way that leave all the files intact.

As an added difficulty, some of the modules are read-only and references to declarations, that are part of these modules, cannot be renamed. An example of this can be seen in Listing 2.7 on line 3, where there is a reference to the builtin type `string`, which as a core part of the Tiger language clearly cannot be renamed.

Complex programs often have external dependencies to third-party libraries. A popular example is the Google Guava collections library that is part of many Java applications. Usually these libraries are tied into a program in a compiled format and cannot be changed either. A sound renaming algorithm needs to correctly distinguish between modules that are open for modification and modules that cannot be changed.

2.8 Performance & Scalability

In order to build a truly useful tool, our renaming transformation needs to work on industry-size applications that have thousands of lines of codes spread across hundreds of files. It is therefore not enough to just provide a sound algorithm. That algorithm also needs to execute within the time frame of a few seconds on large programs, in order to deliver an acceptable user experience.

For this we largely depend on a deterministic and performant name resolution algorithm, which we use to find all occurrences we have to rename. We also need to consider how to effectively detect name capture, as it will require to re-analyze the name binding structure of a program after the renaming transformation was completed in order to find any differences.

2.9 User Interface Integration

In addition to developing a sound renaming algorithm, we need to wrap the transformation in an editor service that delivers a satisfying user experience. The renaming needs to be applicable at the push of a button in order to seamlessly tie into a developer's programming workflow. In case the refactoring leads to an error (for example, if name capture occurs) we need to display a descriptive error message that provides all the information the user requires to understand and fix the problem.

It is also important that the renaming leaves the formatting of the code intact. The transformation should really only change the name occurrences of the selected program entity and leave the layout, namely line breaks, white spaces and comments, unchanged. In essence, the user should not have to perform any changes by hand after executing the renaming in order to make the code look the same.

There exists a plethora of code editors and integrated development environments (IDEs), which support an array of general-purpose and domain-specific programming languages. Since we aim to develop a language-parametric transformation, we need to identify the program model requirements that an editor needs to fulfil in order to successfully integrate the renaming, making our solution editor-agnostic.

2.10 Renaming Requirements

To summarize, we have identified the following requirements for a renaming transformation. A sound and complete Rename refactoring should:

- Given an occurrence, only change the occurrences that correspond to the entity identified by that occurrence
- Prevent name capture
- Rename all occurrences across all files in a project
- Identify program entities that cannot be renamed
- Scale to large programs/projects
- Provide an effective interactive user interface for applying renaming
- Preserves the layout and comments of a program

Chapter 3

A Language-Parametric Solution

3.1 A Language-Parametric Foundation

As name binding and name resolution lay at the center of the difficulty in renaming, it clearly must be an important part of its solution. In order to develop a language-parametric renaming algorithm, we need both a way to express name binding rules in a language-parametric manner as well as a language-parametric name resolution algorithm. Mature solutions to both these problems already exist and we will discuss some of this related work in Chapter 7. Spoofax offers two solution to these concerns, Statix [2] and NaBL2[1], which we will investigate in detail in Chapter 4. Many of the observations described in this chapter are based on the research of Neron et al.[39] on name-binding and α-equivalence.

As the focus of this work is the renaming algorithm, we expect that a language-parametric name resolution algorithm, which encapsulates the language-specific name binding rules, exists as the foundation of our solution. This algorithm needs to be deterministic and we presume it takes an arbitrary name occurrence as an input and returns the declaration(s) this occurrence resolves to. As our solution is designed to work on the tree representation of a program, we also require a parser, which abstracts away the language-specific syntax. Lastly, we expect the static name binding analysis to annotate the tree nodes with name binding information. Thus, we can determine which nodes are reference occurrences and which nodes are declaration occurrences.

3.2 A Generic Representation for Name Bindings

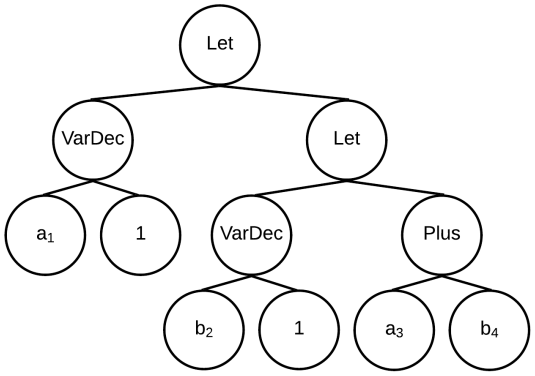
The versatility of name bindings across different programming languages is the main reason for the complexity of developing a language-parametric solution. To tackle this, we need a generic representation of a program's name binding structure that is independent of the language it is written in.

As a first step, the algorithm needs to identify all the name occurrences in a program and enumerate them, so it can refer to them by an index. A program's textual representation is clearly too verbose, as we are not interested in all the indentation that makes it easier to read. Parsing the source code to an AST gets rid of all the white space and gives us a nice tree data structure to work with. Traversing the AST in a deterministic way allows us to give each occurrence a unique *name index*. In Figure 3.1, the name indices are shown as subscripts of the identifiers. It is important to mention that a name index is not attached to the identifier itself but rather to the AST node that represents the identifier. This means that the name index is preserved if the name changes.

We pair up the name index of a reference occurrence x_i^R with the name index of the declaration occurrence x_i^D to create a *resolution pair* (x_i^R, x_i^D). The set of all resolution pairs found in a program gives us the *resolution relation* R. The resolution relation is a binary

```
let
  var a1 := 1
in
  let
    var b2 := 1
  in
    a3 + b4
  end
end
```

(a) Source Code



(b) AST

Figure 3.1: Example Tiger Program with Name Indices

relation on the set of all occurrences Ω and represents the complete name binding structure of a program in a simple mathematical format. We denote Ω_R as the set containing all reference occurrences and Ω_D as the set containing all declaration occurrences, both being subsets of Ω. The program shown in Figure 3.1 yields the resolution relation $R = ((3, 1), (4, 2))$.

It is noteworthy that we only consider occurrences that are tied to identifiers and leave out ones that are tied to keywords. A prominent example for such a keyword is `this`, which appears in object-oriented programming languages and represents a reference to the object on which a method was called. Such a reference occurrence cannot be changed by renaming and thus our transformation does not need to handle it.

3.3 Finding What Belongs Together

After extracting the name binding structure of a program in the form of a resolution relation, the next challenge is to find out which occurrences (declarations and references) that identify the same entity. For example, consider again the Foo Challenge from Section 2.5.

```
1  let
2    type foo1 = {
3      foo2 : string
4    }
5    function foo3 (foo4: foo5) = (
6      let
7        var foo6 := foo7.foo8
8      in
9        print(foo9)
10     end
11   )
12 in
13   foo10(foo11{foo12 = "foo"})
14 end
```

Listing 3.1: The Foo Challenge with Name Indices in Tiger

Listing 3.1 shows the program with all the occurrences carrying their name indices as subscripts. The programs name binding structure in the form of the sets and relations described in the previous section are shown in Figure 3.2.

$$\Omega = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

$$\Omega_D = \{1, 2, 3, 4, 6\}$$

$$\Omega_R = \{5, 7, 8, 9, 10, 11, 12\}$$

$$R = \{(5, 1), (7, 4), (8, 2), (9, 6), (10, 3), (11, 1), (12, 2)\}$$

Figure 3.2: Name Binding of the Foo Challenge (Listing 3.1)

To visualize the name binding structure of a program, we represent the resolution relation as a *name graph* [18]. In this graph the nodes represent occurrences labeled with their name index and the relations from references to declarations are represented as directed edges. The name graph of the Foo Challenge is shown in Figure 3.3.

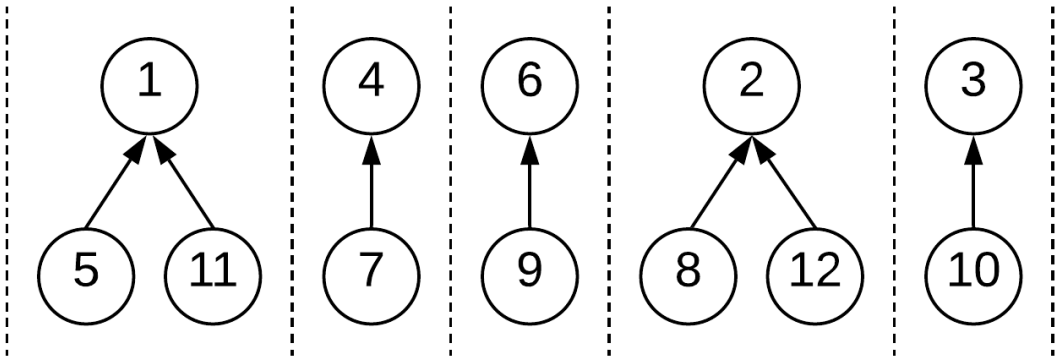


Figure 3.3: Name Graph of the Foo Challenge

Through this visualisation, we can observe that the name graph is disconnected. It consists of multiple sub-graphs, separated in the figure by dotted lines, where each sub-graph contains all the occurrences of one specific program entity. As an example, the leftmost sub-graph in the figure contains the node labeled 1, representing the declaration of the type `foo`, as well as the two nodes 5 and 11, representing the references to that type. If we wanted to rename the type `foo`, we need to change the identifiers at the name indices 1, 5 and 11.

We generalize this observation to the assumption that the name graphs of all non-trivial programs are disconnected and that the sub graphs it contains represent one named program entity each. From the perspective of set theory, each sub-graph corresponds to an equivalence class e_i, which is a partition of Ω. This view is in line with the observation of Neron et. al [39], that the reflexive-transitive closure of the resolution relation R forms an equivalence relation E over Ω. Figure 3.4 shows the equivalence classes of the Foo Challenge including the entity type of each class.

For a correct renaming of a program entity, we need to change all the identifiers in the program that carry the name indices present in the corresponding equivalence class.

While we do not provide a formal proof for this statement, we can evaluate it by examining the effect on renaming if it was reversed. If a program with multiple entities would result

type: $e_1 = \{1, 5, 11\}$

function argument: $e_2 = \{4, 7\}$

variable: $e_3 = \{6, 9\}$

field: $e_4 = \{2, 8, 12\}$

function: $e_5 = \{3, 10\}$

Figure 3.4: Equivalence classes of the Foo Challenge

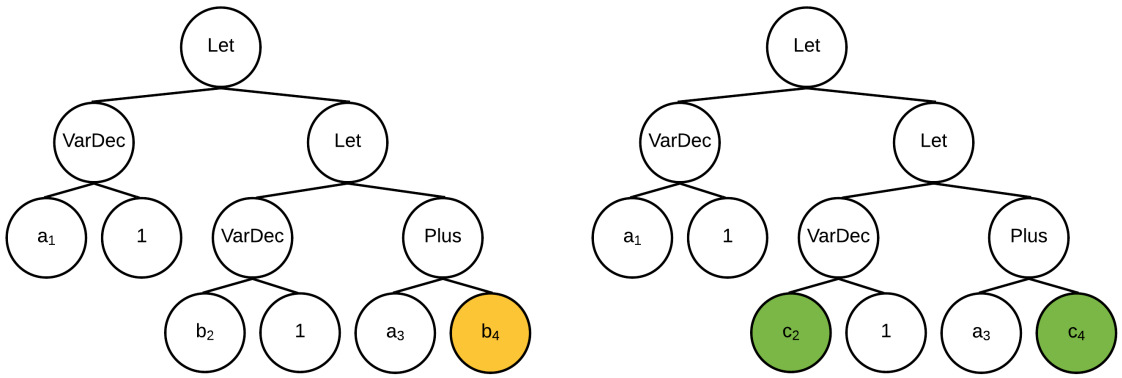
in a fully connected name graph, there would be no way of deciding which occurrences belong to one specific entity. Changing all the identifiers carrying the name indices in the same equivalence class would lead to changing all names in the program, which would only be correct in two trivial cases.

Programs without any names result in an empty name graph and are not interesting, as they do not allow for any renaming. If a program only contains one named entity, the name graph will be fully connected, however changing all the identifiers will result in a correct renaming, as all the names in the program belong to the same entity anyway.

To rename an entity in a program, it suffices to select one occurrence of the name of the entity, regardless whether the selected element represents a declaration or a reference. By identifying the equivalence class in the name graph of the identifier, we can find all other occurrences to rename (and not more). We can rely on the union-find algorithm [23] to solve this problem for us. The time complexity of this algorithm is $\mathcal{O}(m\alpha(n))$ [46, 45] for m operations and n equivalence classes, which was proven to be optimal [22]. With its near-constant performance it is definitely suitable to be the backbone of a fast renaming algorithm.

3.4 Changing the Names

With the ability to find all the name occurrences of a specific program entity we can now progress to perform the actual renaming by changing the identifiers in the program.



(a) Before (b) After

Figure 3.5: Example Renaming on AST

We do this by traversing the AST and inspecting each node. If the node represents an identifier in the source code, we check if its name index belongs to the equivalence class of the entity to be renamed and if so, change the identifier to the new one. We call this the *renaming transformation* T.

While certain name binding patterns can be characterized by the proximity of the identifier nodes in the AST, we are not exploiting this fact and instead require a complete traversal of the AST in an arbitrary order. We leave the optimization of the tree traversal based on the name binding pattern to future work.

As an example, assume we want to rename the local variable `b` in the program shown in Figure 3.6a to `c` by selecting the occurrence $\texttt{b}_4$. The selected name index $n_s = 4$ is in the equivalence class e_2, as can be seen in Figure 3.6b. To rename `b` to `c`, we can traverse the AST and change the identifiers in the nodes that carry the name indices contained in e_2, which is displayed in Figure 3.5. Note that the name indices of the renamed local variables stayed the same, even though the identifiers changed.

```
let
  var a1 := 1
in
  let
    var b2 := 1
  in
    a3 + b4
  end
end
```

(a) Source Code

$$\Omega = \{1, 2, 3, 4\}$$

$$R = \{(3, 1), (4, 2)\}$$

$$e_1 = \{1, 3\}$$

$$e_2 = \{2, 4\}$$

$$n_s = 4$$

(b) Sets

Figure 3.6: Example Tiger Program for Renaming

3.5 Ensuring Preservation of Behavior

We expect that the renaming is semantics preserving, as all refactorings are by definition [20]. Thus the changing of the name of a program entity should not alter a program's observable behaviour. We take a static approximation to semantics preservation.

As a baseline, we define two programs P and P' to be *equivalent* if their ASTs are identical. We expect two equivalent programs to exhibit the same behavior. If we apply the renaming transformation T to a program P it will result in a new non-equivalent program P'. The only trivial case where $T(P) = P$ arises if the new name of the entity to be renamed is the same as the old one.

If we examine our renaming transformation T with focus on its effect on program equivalence, we observe that it does not change the structure of the AST. It does not create, delete or move any of the nodes. It only changes the content of certain leaf nodes that represent the identifiers of the entity to be renamed. We define two programs P and P' to be *structurally-equivalent* if their ASTs are identical with the exception of leaf nodes which represent identifiers. The ASTs of two structurally-identical programs can be seen in Figure 3.5. Since the rename transformation only replaces names with new names, it produces a structurally equivalent program by definition.

Our intuition might tell us that two structurally-equivalent programs would also exhibit the same behavior, thus making our renaming transformation behavior-preserving. For a large class of programs this assumption actually holds true. If each program entity's name is unique and the new name being introduced also doesn't introduce a duplicate, T is indeed behavior-preserving. The example program in Figure 3.5 belongs to this class of programs with entirely singular names.

Even if a program has duplicate names, T is behavior preserving for most renamings. However there are certain cases where a renaming changes the name binding structure of a program by introducing a duplicate name, a problem referred to as *name capture* (see Section 2.6). Changing the name binding structure of a program might also change its semantics and therefore making T a behavior-altering transformation.

Two structurally-equivalent programs that also exhibit the same name binding structure are defined to be α-*equivalent*. This concept was introduced by Alonzo Church's foundational work on the lambda calculus [13]. Expressed using this well-known term, our goal is for $T(P)$ to produce an alpha-equivalent program P'. That is, each reference in P' points to the same declaration (as identified by its name index) as in P and therefore the resolution relation of P and P' are equal. Unfortunately, this is not the case for each application of the rename transformation. A rename transformation that produces a program with a different resolution relation suffers from *name capture*.

However, that two α-equivalent programs exhibit the same runtime behavior can not be guaranteed, when working with languages more complex than the lambda calculus. As an example, Java's Reflection API allows for runtime instrumentation of programs running in the Java Virtual Machine. The API allows to access a member of an object through its name given as a string literal. If we were to rename the method `getOne` in Listing 3.2, the literal on line 5 would remain unchanged, resulting in an error at runtime.

```
1  class A {
2      public int getOneReflection() {
3          B b = new B();
4          Class cls = b.getClass();
5          Method getOne = cls.getDeclaredMethod("getOne");
6          getOne.invoke(b);
7      }
8  }
9
10 class B {
11     public int getOne() {
12         return 1;
13     }
14 }
```

Listing 3.2: Java Program using Reflection

This is a manifestation of the problem of informal name binding, which we described in Section 2.4. The string literal is not recognized as a reference to the method `getOne` and is therefore missing in the resolution relation. As a conclusion of this observation, we need to declare a limitation to our solution: T is only behavior preserving if R contains all occurrences appearing in P.

3.6 Detecting Name Capture

To avoid changing a program's semantics by our refactoring transformation T, we need a way to detect if a renaming leads to name capture. Our approach to this is simple but effective: we just compare the name binding structure of the program before and after the renaming. If the structure changed, name capture occurred and the transformation needs to be aborted. Expressing a program's name binding structure as a resolution relation R basically reduces the problem of capture detection to checking two binary relations for equivalence.

```
1 let
2   var bar1 := 10
3 in
4   let
5     var foo2 := 100
6   in
7     bar3 + foo4
8   end
9 end
```

Listing 3.3: Capture-Prone Tiger Program

```
1 let
2   var bar1 := 10
3 in
4   let
5     var bar2 := 100
6   in
7     bar3 + bar4
8   end
9 end
```

Listing 3.4: Tiger Program with Capture

Revisiting the example from Section 2.6, we can demonstrate that name capture leads to an unintended change of a program's name structure. Listing 3.3 shows a program that is at risk of name capture if a duplicate name `bar` is introduced through renaming. If we were to rename the local variable `foo`, the reference $\texttt{bar}_3$ would be captured by the renamed declaration $\texttt{bar}_2$.

$$\Omega = \{1, 2, 3, 4\}$$
$$R = \{(3, 1), (4, 2)\}$$
$$N_1 = \{1, 3\}$$
$$N_2 = \{2, 4\}$$

(a) Before

$$\Omega = \{1, 2, 3, 4\}$$
$$R = \{(3, 2), (4, 2)\}$$
$$N_1 = \{1\}$$
$$N_2 = \{2, 3, 4\}$$

(b) After

Figure 3.7: Name Binding Structure

The resulting program, shown in Listing 3.4, would output 200 as a result instead of 110, clearly violating the behavior-preserving property of T. By comparing the resolution relations of two programs P and $P' = T(P)$, we can however detect this violation statically, without the need to execute them. Figure 3.7a and Figure 3.7b show the name binding structures of the programs next to them and we can see that the resolution pair $(3, 1)$ was altered to $(3, 2)$ by the transformation, resulting in different equivalence classes.

There are cases where name capture does not alter a program's behavior. Going back to Listing 3.4, if both variables were assigned the value 10, the output of the program would stay unchanged. However, this is generally undecidable at compile time and our algorithm therefore always aborts with an error if name capture is detected. By definition (see Section 2.6) name capture introduces an ambiguous name, which makes it harder to understand a program and is considered to be bad programming style [35]. Since refactoring is a tool meant to improve code quality, it makes little sense to allow name capture anyway, even if it were behavior-preserving.

Simply aborting the transformation with an error might seem like a blunt solution to the name capture problem. Clearly it would be much more elegant if the transformation somehow would repair the name binding structure of the program and thus avoid the change in behavior. For example, the renaming transformation Schaefer[44] developed for Java can automatically rectify certain cases of name capture by using qualified names. Since this solution relies on a specific language features, it does not fit our goal to create a solution that is as language-agnostic as possible. Therefore we did not consider it further in our research.

3.7 The Algorithm

We now have all the pieces in place to assemble our language-parametric, behavior-preserving renaming algorithm. From a top-level view, the algorithm consists of four steps:

1. Extracting the resolution relation from the AST
2. Calculate the equivalence class the selected occurrence belongs to
3. Changing the identifiers in the AST
4. Checking for capture

The first step extracts the resolution relation from the AST through a name resolution algorithm which encapsulates the target language's name binding rules. From the resolution relation we can then find which occurrences describe the same program entity by calculating the equivalence classes. Finding the class to which the occurrence selected by the user belongs to then allows us to find all the occurrences that need to be changed by the renaming.

The third step actually changes the identifiers in the AST, targeting all the nodes with name indices belonging to the selected equivalence class. Lastly, the algorithm checks for capture to ensure the binding structure hasn't changed. This is done by extracting the resolution relation of the renamed AST and comparing it to the resolution relation of the initial one. If they are equal, the algorithm succeeds and returns the renamed AST. It aborts with an error otherwise. A pseudo-code implementation of the algorithm is displayed in Algorithm 1 on the following page. We are going to discuss its implementation, both the Statix and NaBL2 version, in the next chapter.

Algorithm 1: Language-parametric renaming

```
def rename(ast, name-resolution-algorithm, selected-occurrence, new-name):
    resolution-relation := calc-resolution-relation(ast, name-resolution-algorithm)
    target-occurrences := find-equivalence-class(resolution-relation, selected-occurrence)
    renamed-ast := rename-ast(ast, target-occurrences, new-name)
    check-capture(resolution-relation, renamed-ast, name-resolution-algorithm)
    return renamed-ast
def calc-resolution-relation(ast, name-resolution-algorithm):
    var resolution-relation
    for reference in ast do
        (declaration, reference) := name-resolution-algorithm(reference)
        resolution-relation += (declaration, reference)
    end
    return resolution-relation
def find-equivalence-class(resolution-relation, selected-occurrence):
    return union-find(resolution-relation, selected-occurrence)
def rename-ast(ast, target-occurrences, new-name):
    for term in ast do
        if term.nameindex in target-occurrences then
            term = new-name
        end
    end
    return ast
def check-capture(resolution-relation, renamed-ast, name-resolution-algorithm):
    var new-resolution-relation := calc-resolution-relation(renamed-ast,
     name-resolution-algorithm)
    assertEquals(resolution-relation, new-resolution-relation)
```

Chapter 4

Implementation in Spoofax

4.1 Introducing Spoofax

We put the theoretical solution we presented in the previous chapter to the test and create a working implementation. For that we need an implementation of the language-parametric foundation we established in Section 3.1. Specifically, we need to be able to extract the name binding structure of a program written in an arbitrary language along with a name-resolution algorithm that works on top of that structure. Additionally, we need the means to perform the renaming transformation T on an actual program.

The Spoofax Language Workbench [28] is a platform to develop domain-specific and general-purpose programming languages. The aspects of a language, such as its syntax or type system, are implemented as declarative specifications written in meta-languages. From these specifications, Spoofax can generate parsers, type checkers, and other tools commonly found in language SDKs.

Spoofax stands out through its interactivity, as it allows to develop and test a language within one tool. Changes made to a language's definition are hot-loaded into the running Spoofax instance and have immediate effect on programs written in the language under development. As an example, we assume a language engineer would change the syntax definition to use the keyword `let` instead of `var` to declare local variables. Spoofax will generate a new parser from the syntax definition, which would then promptly mark all the loaded programs as faulty which still contain the old keyword. This short feedback loop greatly increases productivity when developing a new programming language.

The Eclipse IDE is the basis on top of which Spoofax runs, packaged as a plugin. As a state-of-the-art development tool, Eclipse provides a user-friendly GUI, integration points for collaboration tools like Git or build tools like Maven, and a host of other useful features. This makes developing a new programming language with Spoofax as convenient as developing a Java program in Eclipse. An experimental plugin for IntelliJ IDEA is also available [42].

The renaming algorithm will be implemented in Stratego [9], a DSL in which program transformations are expressed as term rewrite rules. These rules can be applied when traversing an AST through programmable rewrite strategies and allow us to express the refactoring in a concise way. To remodel a program to an AST, we can rely on the SGLR [48] parser Spoofax generates from a language's SDF3 [29] syntax definition. The parser produces a tree in the language-agnostic ATerm [7] format, on which Stratego can execute a language-agnostic transformation. Spoofax also produces a pretty printer based on the syntax definition, which turns the AST back into source code.

Name binding and typing information of a program are represented as a scope graph [39] in Spoofax. This graph is built based on the results of the static analysis, which is defined in the constraint language NaBL2 [1] or its successor Statix [2]. Spoofax includes a language-agnostic name resolution algorithm which works on top of scope graphs and can be accessed

through an API from a Stratego transformation. In other words, Spoofax provides us with the language-parametric foundation we need in order to build our renaming tool.

While it already offers certain editor services, such as syntax highlighting and syntactic code completion, Spoofax is still missing a renaming refactoring. The implementation of this refactoring, which we describe in this chapter, is one of the main contributions of this Master thesis.

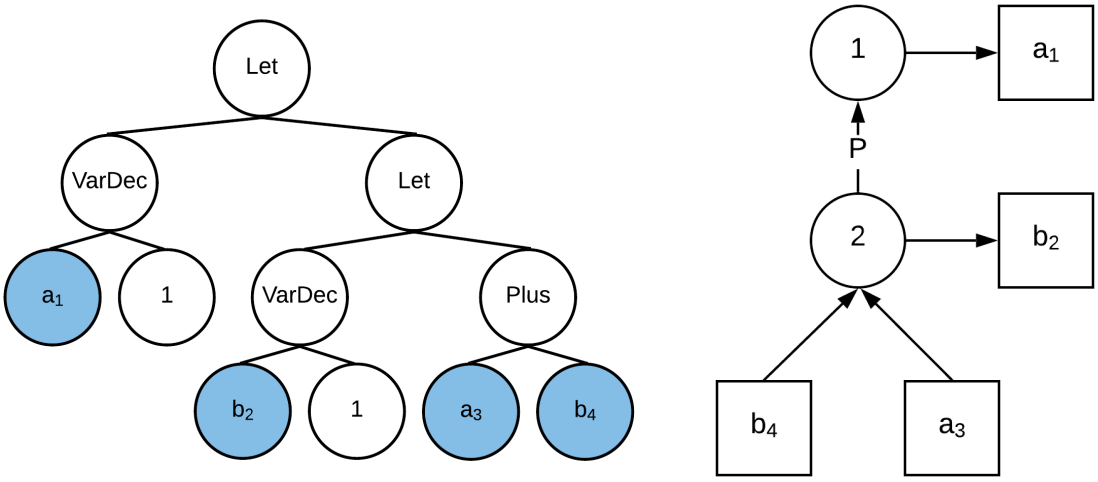
4.2 Name Binding with Scope Graphs

As name binding and resolution are of principal concern when developing a renaming algorithm, we describe first how Spoofax handles these matters. *Scope graphs* are a formalism to represent a program's name binding structure [39]. The scope graph abstracts away all unnecessary details from the AST and aggregates all the information needed for name resolution into a concise model. This model is language independent an can represent an array of different name binding patterns. Scope graphs consist of the following components:

- A *scope* represents a program region that behaves uniformly with respect to name resolution. Graphically, they are depicted as circles that are labeled with a number that identifies the scope.
- A *declaration* represents the introduction of a name into a scope. In text we represent them as $a_i^{\mathbf{D}}$, where a is the identifier and i refers to the location in the AST. In diagrams we represent them as rectangles labeled with their names. A declaration is associated with a scope through a directed edge pointing from the scope to the declaration.
- A *reference* represents the use of a name that refers to a declaration with the same name. Formally we write $a_i^{\mathbf{R}}$ for a reference to name a at position i. Graphically, they are represented the same as declarations, except the the arrow points from the the reference to the scope.
- A *labeled edge* is a directed connection between scopes. The labels can be used to distinguish different connections, e.g. I for imports or P for parent relations. Graphically, they are represented as pointed arrows with their label on them. In text we write s -I-> s'.

Figure 4.1a shows the AST of a simple Tiger program, which we use to showcase a scope graph. The nodes representing names are highlighted blue and the identifiers carry their name index as a subscript. Figure 4.1b shows the scope graph diagram of the example program. The name index relates nodes in the AST to the corresponding nodes in the scope graph.

The outer let binding introduces scope 1, in which the variable $a_1^{\mathbf{D}}$ is declared. The inner let binding introduces scope 2, which connects to the scope of the outer let binding through a parent(P) edge. This has the effect that declarations from the parent scope are accessible from the child scope. Scope 2 introduces the variable $b_2^{\mathbf{D}}$ and contains the two variable references $a_3^{\mathbf{R}}$ and $b_4^{\mathbf{R}}$.



(a) AST

(b) Scope Graph Diagram

Figure 4.1: Scope Graph Example

Consider how we can perform name resolution within this scope graph. Specifically we look at how we can find $a_1^{\mathbf{D}}$ from $a_3^{\mathbf{R}}$. To resolve a name, the scope graph is traversed following the directed edges starting from the reference node. Looking at our example in Figure 4.2, we first take an R step from $a_3^{\mathbf{R}}$ to scope 2. Next we look for declarations in that scope, taking step D to $b_2^{\mathbf{D}}$. Since the names don't match, this is clearly not the right declaration. As there are now other declarations in that scope, we take a step along the P-edge to scope 1. Her we look again for declarations, reaching $a_1^{\mathbf{D}}$ through another D step to find the correct declaration.

While traversing the scope graph, the situation can arise where there are multiple directed edges that can be followed. In the example after taking the R step, we can either take a D step to $b_2^{\mathbf{D}}$ or an P step to scope 1. To resolve such ambiguous situations, we impose a strict partial *specificity ordering* on resolution paths, which defines which step to take. In our case this ordering is $R < D < P$, meaning reference steps have priority over declaration steps, which in turn have priority over a parent relation step. This explains why in our example we first check the declaration $b_2^{\mathbf{D}}$ before taking the parent step. Note however that this specificity ordering is language specific and might range over more edge labels. For example, many languages also include I steps to model imports.

Besides the specificity ordering, path's also need to adhere to a path well-formedness predicate *WFP*. This predicate is a regex expression that disallows certain resolution paths. $WFP = R.P^*.D$ determines that a valid path always has to start with a reference step, may take 0-n parent steps after that and ends in a single declaration step.

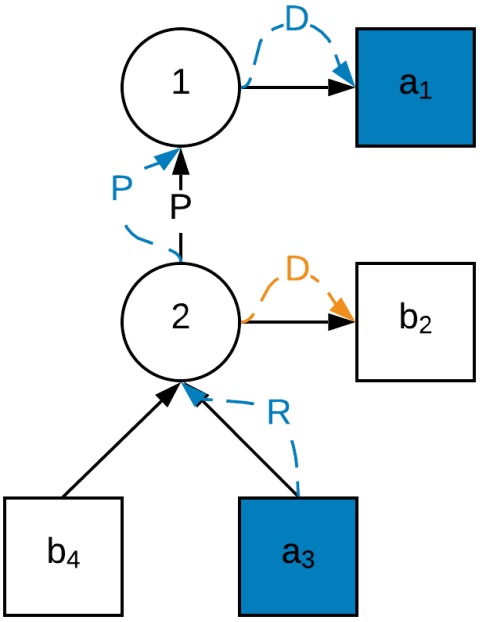


Figure 4.2: Name Resolution Example

4.3 Defining the Name Binding Rules

We've seen how scope graphs can represent a program's name binding structure in an abstract way. Now, let's discuss how we actually construct a graph from an input program.

As mentioned before, manipulation programs in their textual form can be cumbersome and therefore we first parse the program and represent it as an AST. Given a language's grammar as an input, Spoofax can generate a parser for that language [30], that we can use for this first step. The grammar must be declared using the meta-language SDF [29], an example syntax definition is shown in Listing 4.1. On line 1 of the example program we define the *sorts* Dec, Exp and $Name$. Sorts are abstract syntactic categories that can be instantiated through the application of *constructors*, which are defined below the keyword `context-free-syntax`.

The constructor on line 4 is called `VarDec` and it creates instances of the `Dec` sort. The textual pattern to define a variable can be seen on the right side of the equals sign in angle brackets. It starts with the keyword `var`, followed by a name, the assignment operator `:=`, and an expression that initializes the variable. The constructor `Var` on line 5 represents a variable reference as a name. When revisiting the AST in Figure 4.1a, we can see that the names of the constructors appear as labels on the AST nodes, where the parser applied them.

```
1 sorts Dec Exp Name
2
3 context-free syntax
4   Dec.VarDec = <var <Name> := <Exp>>
5   Exp.Var = Name
```

Listing 4.1: Example SDF Specification

When developing a language in Spoofax, we use the NaBL2 meta-language [1] to declaratively specify name binding and typing rules, i.e. the static semantics. We do this by declaring

rules that match on nodes in the AST, based on the constructors from the syntax definition, and mapping these nodes to constraints. A *constraint* is a logical assertion that needs to hold in order for the static analysis to pass.

In Listing 4.2 we define rules to statically check variable declarations and references. The rule on line 2 matches the `VarDec` constructors, binding its two children `name` and `init-exp`. The matching clause also contains a scope, in which the variable declaration will be introduced. On the lines 3-5 are the constraints that need to hold in order for the type checker to not complain.

The first constraint invokes a rule that checks the expression that initializes the variable and yields the expressions type as a return value. The second constraint assigns the type of the initializer expression to the declaration. The last constraint adds a declaration node to the scope graph in the current scope. This declaration is part of the `Var` namespace.

NaBL2 allows for the definition of multiple *namespaces* within a scope graph, allowing the same name to be declared in the same scope without a name clash, given the declarations belong to different namespaces. In C# for example, it is not possible to define multiple fields with the same name in the same class. It is however possible to have a method and a field in the same class that have an identical name. This can be modeled through separate namespaces for fields and methods.

```
1  rules
2      [[ VarDec(name, init-exp) ^ (scope) ]] :=
3           [[ init-exp ^ (scope) : type ]],
4           Var{name} : type !,
5           Var{name} <- scope.
6
7      [[ Var(name) ^ (scope) : type ]] :=
8           Var{name} -> scope,
9           Var{name} |-> d,
10          d : type.
```

Listing 4.2: Example NaBL2 Specification

The rule for evaluating variable references starts on line 7. The match-clause for this rules determines that the evaluation yields a type as the result. The first constraint adds the reference to the scope graph and the second constraint says that this reference must resolve to some declaration `d`. The third constraint asserts that the type of the declaration equals the type of the reference.

Given a NaBL2 specification and an AST, Spoofax can now check the static semantics of the program. It does so in two phases: *constraint* generation and constraint solving. First the AST is traversed and every node is evaluated according to the rules defined in the NaBL2 specification, yielding a set of constraints as a result. In the second phase, a generic solver algorithm tries to find a solution for that set of constraints by means of *unification* [5]. If a solution is found this generally means the program type checks and does not contain any unbound references.

The results of the static analysis are linked to the nodes in the AST through *annotations*. These annotations can be queried by later compilation steps, such as desugaring or code generation. For example when generating byte code for the JVM, we can query a variable reference for its type, from which we derive the corresponding load instruction. In case of our renaming transformation, he name indexes are added to the AST nodes as annotations, which we can use to query the scope graph and perform name resolution.

4.4 Introducing Stratego

Spoofax includes the Stratego [9] transformation language which we used to implement our renaming transformation T. In Stratego, program transformations are expressed as *term rewrite rules* which match on certain terms in a program's AST and then add, delete or modify terms in order to manifest the transformation. In this section we give a brief overview of Stratego that should help the reader understand the code in the rest of this chapter. If some part of the code remains unclear, a more extensive introduction of the language can be found in its official documentation [15].

An example of a program transformation expressed in Stratego is shown in Listing 4.3. There we implemented the law of De Morgan [36] in the form of the rewrite rule `de-morgan`. The rule matches on the boolean And and Or operators, which are nested inside a Not operator, expressed as terms on the left-hand side of the arrow. If the match succeeds, the term is then transformed into the form given on the right-hand side of the arrow, thus applying De Morgan's law in our example.

```
1 rules
2   de-morgan : Not(And(x, y)) -> Or(Not(x), Not(y))
3   de-morgan : Not(Or(x, y))  -> And(Not(x), Not(y))
```

Listing 4.3: Applying De Morgan's law in Stratego

The terms Stratego operates on are formulated in the *Annotated Term Format* [7], or *ATerm* for short. The ASTs that are produced by Spoofax's generated JSGLR parser, are also expressed in that format, which facilitates a seamless workflow when parsing, transforming and then pretty printing a program. A term t in the ATerm format can have one of the following forms:

- An integer constant
- A string literal
- A constructor application, where the constructor c is an arbitrary identifier which has 0-n terms t_n as arguments, expressed as $c(t_0, ..., t_n)$
- A list of terms, expressed as $[t_0, ..., t_n]$
- A n-tuple of terms, expressed as $(t_0, ..., t_n)$

Optionally, each term t in the ATerm format can be annotated with a list of other term $tt_a, ...t_n$ to provide additional semantic information. This can be used, for example, to provide type information on an expression such as $Or(x, y)\{Type("bool")\}$. We use these annotations to implement the name indexes described in Section 3.2.

To express more complex program transformation, rewrite rules can be combined to produce *rewrite strategies*, which can selectively apply rules during an AST traversal. The example in Listing 4.4 was taken from the official Spoofax documentation [15] and shows the strategy `cnf`, which rewrites a boolean expression into its conjunctive normal form.

It does so by traversing the AST bottom up, using the builtin `innermost` traversal strategy. During the traversal it tries to apply the 7 rules defined on lines 2-8 on each node. The rules are applied using the *deterministic choice* combinator $s_1 < +s_2$. This combinator first tries to apply s_1 to the current term, but backtracks if s_1 fails and applies s_2 to the original term.

```
rules
  DefI : Impl(x, y) -> Or(Not(x), y)
  DefE : Eq(x, y)   -> And(Impl(x, y), Impl(y, x))
  DN   : Not(Not(x)) -> x
  DMA  : Not(And(x, y)) -> Or(Not(x), Not(y))
  DMO  : Not(Or(x, y))  -> And(Not(x), Not(y))
  DOAL : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  DOAR : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))

strategies
  cnf = innermost(DefI <+ DefE <+ DOAL <+ DOAR <+ DN <+ DMA <+ DMO)
```

Listing 4.4: Rewrite Boolean Expression to Conjunctive Normal Form

4.5 The Implementation in Detail

We implemented the algorithm we described in Section 3.7 as the Stratego strategy in Listing 4.5. We will explain each of the steps in detail in the following subsections by reference to the NaBL2 version of the implementation. The Statix implementation works mostly the same, but there are some differences between the respective Stratego APIs which we detail in Section 4.6. The implementation discussed in this section only works on single-file programs, the version for multi-file programs is detailed in Section 4.7.

In the rest of this chapter, we present the code as we developed it going from NaBL2 to Statix and from single-file to multi-file. In some cases we took the liberty to leave out certain uninteresting implementation details for the sake of brevity. The final implementation of the refactoring exists in two versions, one using NaBL2 and one using Statix. Both versions can deal with single- and multi-file programs. We opted to describe the implementation in a step-by-step manner to make it easier to understand and follow along. In Appendix A.1 we list all the repositories where our source code can be found as well as all our code changes in the form of pull request.

One thing we need to do before we can start the algorithm is to access the result of the static analysis of the program. In the strategy `get-analysis`, we extract the analysis object, which contains all the name binding information our algorithm needs, through the API call `nabl2-get-ast-analysis`.

```
rename(|selected-term, new-name, path): ast -> renamed-ast
  where
    analysis := <get-analysis> ast
    ; resolution-relation := <calc-resolution-relation> analysis
    ; selected-occ := <find-occurrence(|analysis)> selected-term
    ; target-indices := <find-all-related-occs> (selected-occ, res-rel)
    ; renamed-ast := <rename-ast(|target-indices, new-name)> ast
    ; check-capture(|renamed-ast, resolution-relation, path)

get-analysis: ast -> analysis
  where
    analysis := <nabl2-get-ast-analysis> ast
```

Listing 4.5: Rename Strategy

4.5.1 Calculating the Resolution Relation

The first step of the algorithm is to build the resolution relation as an abstract representation of a program's name binding structure. The strategies for this step are shown in Listing 4.6. We start by extracting all the reference and declaration occurrences in the program from the analysis object. We can do that by using the strategies `nabl2-get-all-refs` and `nabl2-get-all-decs` which are part of NaBL2's Stratego API.

Having all the occurrences we can start building the resolution relation. Using the `make-resolution-pair` strategy, we loop through all references and pair them with their corresponding declarations. To make these resolution pairs we use NaBL2's name resolution algorithm to by calling the strategy `nabl2-get-resolved-name` on the reference occurrence.

References and declarations are represented through ATerms in the form `Occurrence(namespace, name, term-index)`. The namespace and name are not relevant for us, as we form the resolution relation out of pairs of term indices. Therefore we use the strategy `get-term-index-from-occ` to extract the `TermIndex` subterm from the `Occurrence` term.

The constructor `TermIndex(path, index)` is an implementation of the name index concept we described in Section 3.2, which we use to identify occurrences in the AST. The first subterm `path` refers to the file path of the program and the second subterm `index` is used to enumerate AST nodes within a file.

```
1  calc-resolution-relation: analysis -> user-defined-relation
2    where
3      refs  := <nabl2-get-all-refs(|analysis)>
4      ; decs := <nabl2-get-all-decls(|analysis)>
5      ; ref-dec-pairs := <map(make-resolution-pair(|analysis))> refs
6      ; decs-reflexive-pairs := <map(make-reflexive-pair)> decs
7      ; relation := <conc> (decs-reflexive-pairs, ref-dec-pairs)
8      ; user-defined-relation := <filter(is-user-defined)> relation
9
10 make-resolution-pair(|analysis) : ref -> (ref-index, dec-index)
11   where
12     (dec, _) := <nabl2-get-resolved-name(|analysis)> ref
13     ; ref-index := <get-term-index-from-occ> ref
14     ; dec-index := <get-term-index-from-occ> dec
15
16 make-reflexive-pair: dec -> (term-index, term-index)
17   where
18     term-index := <get-term-index-from-occ> dec
19
20 is-user-defined: (ref, dec@TermIndex(path, num-index)) -> <id>
21   where
22     <not(eq)> (num-index, 0)
23
24 get-term-index-from-occ: Occurrence(_, name, term-index) -> term-index
```

Listing 4.6: Calculating the Resolution Relation

Pairing all references to their declaration however doesn't necessarily collect all occurrences into the resolution relation. If a program contains declarations which are never referred to, it would not be gathered that way and thus the name binding structure would be incomplete. Therefore we simply add a reflective resolution pair for each declaration to the relation through the strategy `make-reflexive-pair`.

As we described in Subsection 2.3.2, a program might contain references to language entities that are declared outside the program itself. For example in Listing 4.6 we call multiple strategies from NaBL2's Stratego API. Clearly we cannot rename an entity declared in an external assembly, therefore we remove all the resolution pairs that contain foreign declarations from the resolution relation.

When resolving a reference that leads to an external declaration the NaBL2 name resolution algorithm outputs a `TermIndex` with the same path as the reference and the index 0. Therefore we can filter all resolution pairs where the declaration's index is 0 out of the resolution relation, using the strategy `is-user-defined`. Ergo the return value of the `calc-resolution-relation` strategy contains the name binding information of all the user-defined language entities in the program.

4.5.2 Checking The Selected Name

In the second step of the algorithm we need to find a name occurrence in the users selection and check if it is possible to rename the program entity it belongs to. The strategies we use for this are shown in Listing 4.7. We traverse the selected term in the strategy `find-name-index` and check if any of the nodes have a `TermIndex` that occurs in the resolution relation.

```
1  find-name-index(|res-rel): term -> occ-index
2    where
3      occ-index := <collect-one(get-name-index(|res-rel))> term
4        <+ add-error(|"The selected entity cannot be renamed.")
5
6  get-name-index(|res-rel): term -> occurrence
7    where
8      if (<is-list> term) then
9        <map(get-occurrence(|res-rel))> term
10     else
11       <is-string> term
12       ; term-index := <nabl2-get-ast-index> term
13       ; occurrence := <fetch-elem(res-pair-contains(|term-index))> res-rel
14     end
15
16 res-pair-contains(|term-index): (ref-index, dec-index) -> term-index
17   where
18     <eq> (term-index, ref-index) <+ <eq> (term-index, dec-index)
```

Listing 4.7: Find Selected Name Index

We use the strategy `get-name-index` to check if a term is part of the user-defined name binding structure of a program. If the strategy is called on a list, we simply recursively apply the strategy to every element of the list. If it is not a list, we first check if the term is a string, as we expect all names to be alphanumeric identifiers. If it is a string, we extract its `TermIndex` and then check if that `TermIndex` appears in the resolution relation.

For this we use the `fetch-elem(s)` strategy, which calls the argument strategy `s` on each element of a list until the strategy succeeds. In our case, we loop through all resolution pairs and check if the `TermIndex` of either the declaration or the reference is equal current term's `TermIndex`. If that is the case on any resolution pair, we conclude that the user selected a term which we can rename.

4.5.3 Finding All Related Occurrences

At this point, we aggregated the program's name binding structure and identified the name occurrence that the user has selected. The next step in our renaming algorithm is to find all the other occurrence that belong to the same program entity which the user wishes to change. We do that by building a name graph from the resolution relation and partitioning it in a way that all the occurrences which belong together are part of the same sub-graph. We describe the theory behind this process in Section 3.3.

We use a basic version of the union-find [23] algorithm to store the name graph in a disjoint-set data structure. It is important to note that our variant of union-find is a lot simpler than any reference implementation and does not share their algorithmic complexities. However, adapting the concept of the algorithm to our specific problem domain made it easier to understand and test.

Since Stratego's collections library is not very extensive and because the name graph lends itself to be modeled in an object-oriented fashion, we opted to implement this step of the algorithm in Java. Also the JUnit testing framework made it convenient to test our union-find implementation.

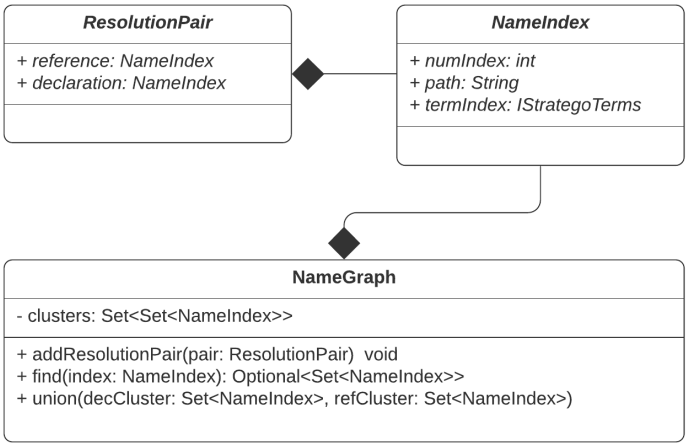


Figure 4.3: Class Diagram of Union-Find Implementation

Figure 4.3 shows the UML class diagram of our implementation. The class `NameIndex` represents a node in the name graph and the class `ResolutionPair` represents an element of the resolution relation. Both are immutable value classes which have their `equals` and `hashCode` methods overridden in order for them to behave correctly when stored inside a Java collection. The algorithm itself is implemented in the `NameGraph` class which is displayed in Listing 4.8. The graph state is represented as a set of sets of `NameIndex` objects using the `HashSet` implementation of the Java Collections API.

We build the name graph by calling the `addResolutionPair` method with every pair from the resolution relation. In that method we first check if either the declaration or the reference are already in the name graph by calling `find` on both `NameIndex` objects. The `find` method just loops through all the graph's partitions and returns the one the node is in, given it exists.

Knowing whether the declaration and the reference node are already in the graph allows us to select one of four possible branches through which the algorithm continues. If neither of them are part of the name graph yet, we create a new partition containing both nodes and add the partition to the graph. In the case one node is already in the graph but the other one is not, we add the missing node to the partition of the included one. If both nodes are already in the graph, we call the `union` method to merge the two partitions.

```
1  public void addResolutionPair(ResolutionPair pair) {
2    Optional<Set<NameIndex>> decCluster = find(pair.getDeclaration());
3    Optional<Set<NameIndex>> refCluster = find(pair.getReference());
4
5    if(!(decCluster.isPresent() || refCluster.isPresent())) {
6      Set<NameIndex> cluster = new HashSet<>();
7      cluster.add(pair.getDeclaration());
8      cluster.add(pair.getReference());
9      clusters.add(cluster);
10   } else if (decCluster.isPresent() && !refCluster.isPresent()) {
11     decCluster.get().add(pair.getReference());
12   } else if (!decCluster.isPresent() && refCluster.isPresent()) {
13     refCluster.get().add(pair.getDeclaration());
14   } else if (decCluster.isPresent() && refCluster.isPresent()) {
15     union(decCluster.get(), refCluster.get());
16   }
17 }
18
19 public Optional<Set<NameIndex>> find(NameIndex index) {
20   return clusters.stream()
21     .filter(cluster -> cluster.contains(index))
22     .findFirst();
23 }
24
25 public void union(Set<NameIndex> decCluster, Set<NameIndex> refCluster) {
26   if (decCluster != refCluster) {
27     decCluster.addAll(refCluster);
28     clusters.remove(refCluster);
29   }
30 }
```

Listing 4.8: Union-Find Implementation

To call this Java code from a Stratego strategy, we need to wrap in a *primitive* which is a delegate object that relays the strategy call and transforms the Stratego terms into Java objects if necessary. We call the union-find algorithm through the primitive shown in Listing 4.10 which is a subtype of `AbstractPrimtive`. It overrides the `call` method, which is executed when the primitive strategy shown in Listing 4.9 is called.

```
1  find-all-related-occs = prim("FindAllRelatedOccurrences")
```

Listing 4.9: Union-Find Primitive Strategy

The way our Rename refactoring uses the union-find algorithm. to find all name occurrences of a specific program entity, is implemented in the private method `findCluster`. There we grab the resolution relation and the selected occurrence from the term the primitive strategy is called with. We then build the name graph by passing the resolution relation to the constructor of the `NameGraph` class.

Once we have the name graph we can simply call the `find` method on it to retrieve all the occurrences related to the one the user selected. To return the target term indices from the primitive strategy we need to convert them from domain objects back to Stratego terms. This is why each `NameIndex` object retains a reference of type `IStrategoTerm` to the `TermIndex` term, from which it was created.

In the `call` method of the primitive, we check if we found the equivalence class of the selected occurrence. If so we set the list term returned by `findCluster` as the contexts current term and return true. If not so we return false which lets the strategy fail.

```
1  public class FindAllRelatedOccurrencesPrimitive extends AbstractPrimitive {
2    @Override
3    public boolean call(IContext env, Strategy[] svars, IStrategoTerm[] tvars) {
4      IStrategoTerm targetIndices = findCluster(env.current(), env.getFactory());
5      if (targetIndices != null) {
6        env.setCurrent(targetIndices.get());
7        return true;
8      }
9      return false;
10   }
11
12   private IStrategoTerm findCluster(IStrategoTerm current, ITermFactory fac) {
13     NameIndex selectedOccurrence = new NameIndex(current.getSubterm(0));
14     List<IStrategoTerm> resRel = TermUtils.toJavaList(current.getSubterm(1));
15     NameGraph nameGraph = new NameGraph(resRel);
16
17     Set<NameIndex> cluster = nameGraph.find(selectedOccurrence).get();
18     List<IStrategoTerm> targetIndices = cluster.stream()
19       .map(term -> term.getTermIndex())
20       .collect(Collectors.toList());
21     IStrategoList targetIndicesTerm = fac.makeList(targetIndices);
22     return targetIndicesTerm;
23   }
24 }
```

Listing 4.10: Union-Find Primitive Class

4.5.4 Changing The Names

Now that we know which name occurrences we need to change, we can actually modify the program to rename the selected program entity. We do that by traversing the whole AST and replacing the terms representing the old name with terms representing the new name chosen by the user. The source code of this step is shown in Listing 4.11. We implemented this in the strategy `rename-ast`, where we traverse the AST from the bottom up and try to apply `rename-term` on each node.

```
1  rename-ast(|target-indices, new-name): ast -> renamed-ast
2    where
3      renamed-ast := <bottomup(try(rename-term(|target-indices, new-name)))> ast
4
5  rename-term(|target-indices, new-name): t -> new-name
6    where
7      <is-string> t
8      ; term-index := <nabl2-get-ast-index> t
9      ; <elem> (term-index, target-indices)
```

Listing 4.11: Changing The Name Terms

The `rename-term` strategy basically checks if a term is a name occurrence that is targeted by the Rename refactoring. It first checks if the term is a string, because we expect a name to be a textual identifier. It then checks if the `TermIndex` of the node is part of the target indices that are passed into the strategy as a term argument. If that is the case, then the term represents a name that needs to be changed and it is replaced with the new name.

4.5.5 Checking for Capture

After performing the renaming transformation on the AST, what remains is to check for name capture. As described in Section 3.6, we do this by comparing the name binding structure of the program before and after the renaming. We implemented this as the strategy `check-capture` shown in Listing 4.12.

To extract the name binding structure of the new program, we need to rerun the static analysis. We do this by calling the `nabl2-analyze-ast` strategy from the NaBL2 API, which returns a new analysis term that contains the name binding information of the renamed program. From this analysis we can calculate the resolution relation in the same way we did before we executed the renaming. For that we call the strategy `calc-resolution-relation` again, which we described in subsection 4.5.1.

Before we can compare the old and the new resolution relation, we need to make sure that the list terms, which represent the relations, are ordered the same. We use the `qsort(s)` strategy from the Stratego standard library, which is an implementation of Quicksort [25], to sort the lists. The strategy that `qsort` expects as an argument is used to compare the elements in the list that is to be sorted.

Since we are trying to sort a list of resolution pairs, we implemented the strategy `res-pair-gt` which passes judgment which one is "greater" when comparing two pairs. We can do this by only comparing the `TermIndex` terms of the reference occurrence, as each reference only appears once in the resolution relation.

We compare `TermIndex` terms with the strategy `term-index-gt`. The strategy orders the resolution relation first lexicographically by the path and second by the numerical index. We compare the subterms using the built-in `string-gt` and `gt` strategies.

```
1  check-capture(|renamed-ast, res-rel, path) =
2    (_, new-analysis, _, _, _) := <nabl2-analyze-ast(|path)> renamed-ast
3    ; new-res-rel := <calc-resolution-relation; qsort(res-pair-gt)> new-analysis
4    ; old-res-rel := <qsort(res-pair-gt)>  res-rel
5    ; <eq>(old-res-rel, new-res-rel)
6      <+ add-error(|"This renaming leads to name capture")
7
8  res-pair-gt: ((ref-1, dec-1), (ref-2, dec-2)) -> <id>
9    where
10     <term-index-gt> (ref-1, ref-2)
11
12 term-index-gt: (TermIndex(path-1, num-1), TermIndex(path-2, num-2)) -> <id>
13   where
14     if <eq> (path-1, path-2) then
15       <gt> (num-index-1, num-index-2)
16     else
17       <string-gt> (path-1, path-2)
18     end
```

Listing 4.12: Detecting Capture

Now that we have the name binding structure of the program before and after the renaming represented as two sorted list terms, we can simply compare the terms using the `eq` strategy. If they differ, we detected a case of name capture and abort the transformation with an error. If they are the same, the renaming was successfully and the algorithm terminates.

4.5.6 User Interface

Since a refactoring is triggered by the developer, we implemented a user interface to make our renaming solution accessible from the Eclipse IDE. We added a new entry to the Spoofax menu to allow programmers to execute the renaming. We implemented that menu using the Editor SerVice (ESV) meta-language [28], which is part of Spoofax. ESV allows the declarative configuration of various editor services in an editor-agnostic way. Besides a way to define menus, ESV also supports editor feature such as syntax highlighting, hover tooltips, code completion and file outlines.

We added the menu for renaming using the ESV module shown in 4.13. Here we created a whole new menu for refactorings, hoping that renaming might soon be joined by other refactorings (see the future work section 8.1). It would also be possible to just add the menu item for renaming (line 4) to a menu already existing within the language project. Figure 4.4 shows how this menu looks like in the Eclipse IDE.

```
1 module Refactoring
2 menus
3   menu: "Refactoring"
4     action: "Rename" = rename-menu
```

Listing 4.13: Renaming Menu in ESV

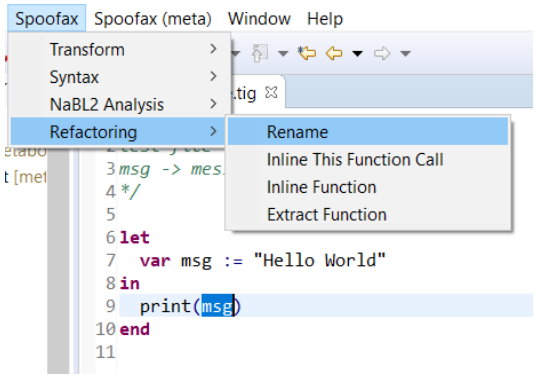


Figure 4.4: Menu to Trigger Renaming

When the user clicks on the Rename menu, the language-specific renaming strategy `rename-menu` is executed which is shown in Listing 4.14. Both the menu and the strategy it calls need to be part of the Spoofax language project. The `rename-menu` strategy simply calls the builtin language-parametric `rename-action` strategy, passing it a language-specific strategy for pretty-printing the result of the renaming transformation. We describe how the RENAME refactoring can be added to a language in detail in subsection 4.8.2.

Spoofax generates a strategy for pretty printing from the syntax definition of a language, which we could use to turn the renamed AST back into source code. However, this strategy does not preserve the layout of the original program and inline comments are lost as well. Since renaming only should result in a few local changes to the source code by replacing some specific identifiers, having the whole program formatted after each refactoring is clearly a major inconvenience.

Jonge and Visser[26] developed an algorithm for "automatic source code reconstruction for source-to-source transformations" which we use to solve that problem. Their solution tracks the textual origin of each AST node which they use to compute the change to the source code by comparing the AST before and after the transformations. They implemented their research in Spoofax and their algorithm generates the layout-preserving pretty-printing strategy `construct-textual-change` for each language project.

```
1 rename-menu: menu-terms ->
2   <rename-action(construct-textual-change|)> menu-terms
```

Listing 4.14: Language-specific Renaming Strategy

The entry point for our language-parametric renaming algorithm is the `rename-action` strategy shown in Listing 4.15. It wraps the algorithm strategy and deals with all the user interface aspects of our solution. That way we achieve a separation of concern between the transformation and the user interaction.

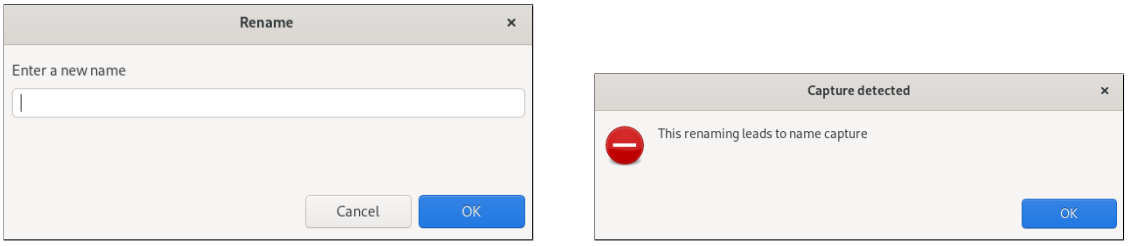
If a strategy is called from a menu, the editor composes a 5-tuple that gets passed to the strategy. This tuple contains the term that corresponds to the users selection, the analyzed AST of the currently open file and the relative path of that file. The menu strategy is expected to return a pair consisting of a file path and an arbitrary term. When the strategy returns, the editor then persists that term at the location of the given path.

The construct enveloping the body of the strategy (line 5 - 7) is part of our algorithm's error handling which we discuss in detail in the following subsection 4.5.7 and thus will ignore for now. Before we can execute the renaming algorithm, we need to ask the user for the new name of the selected program entity. The strategy `read-new-name` returns the name the user types in the modal form shown in Figure 4.5a. It uses the the builtin `show-input-dialog` strategy to display the modal.

```
1  rename-action(construct-textual-change|) :
2    (selected-term, _, ast, path, _) -> (path, result)
3      where
4        {| ErrorMessage:
5          new-name := <read-new-name>
6          ; renamed-ast := <rename(|selected-term, new-name, path)> ast
7          ; (_, _, result) := <construct-textual-change> (ast, renamed-ast)
8        <+ show-errors |}
9
10 read-new-name: _ -> new-name
11   where
12     new-name := <show-input-dialog(|"Rename")> "Enter a new name"
```

Listing 4.15: Rename-Action Strategy

After the user chooses the new name, we then execute the renaming algorithm, as we described it in the previous subsections 4.5.1 - 4.5.5. The algorithm returns the AST with the changed identifiers, which we feed into the `construct-textual-change` strategy together with the original AST. This call returns the refactored source code of the program which we pass back to the editor to be persisted.



(a) Enter New Name (b) Error Message

Figure 4.5: User Interface Modals

4.5.7 Error Handling

There are multiple cases in which our renaming transformation needs to abort. For example, if we detect name capture. To handle such error cases gracefully, we integrated an explicit error handling in our transformation. This allows us to notify the developer in a user friendly way in case something went wrong. An example error message is shown in Figure 4.5b.

We've already come across the `add-error` strategy in previous subsections, for example in Listing 4.12 on line 6. There we use the deterministic choice operator to call `add-error`, in case the comparison of the old and the new resolution relation fails. We pass the strategy an error message as a term argument and then `add-error` stops the execution of the strategy. This has a similar effect to throwing an exception in a object oriented language like C++.

The `add-error` strategy, shown in Listing 4.16, is implemented as a *scoped dynamic rule* [8]. Rewrite strategies are context-free in general, but these dynamic rules allow us to track context-sensitive information during term rewriting without explicitly passing it. We use this to add error messages to the `ErrorMessage` context. This context is valid within a certain scope that is defined in the `rename-action` strategy shown in Listing 4.15. The delimiters for the scope are on line 4 and 8, and any strategies called within that scope have access to the named context.

If any of the strategies inside the scope fail, which they do if `add-error` is called, the `show-errors` strategy is called. So this basically catches the "exception" thrown by `add-error`. Then `show-errors` reads the first error message from the `ErrorMessage` context and displays it as an error modal using the builtin `show-dialog` strategy.

```
1  add-error(|message) =
2    rules(
3      ErrorMessage :+ () -> (message)
4    ); fail
5
6  show-errors =
7    [(message) | _] := <bagof-ErrorMessage> ()
8    ; <show-dialog(|"ERROR")> message
```

Listing 4.16: Error Handling Strategies

4.6 Using Statix

In the preceding section, we described the implementation of our renaming solution which works for projects that use NaBL2 to define their static semantics. However NaBL2 is becoming deprecated and is being replaced by its successor Statix. Therefore we implemented a version of our renaming transformations that works with languages that use Statix.

Since NaBL2 and Statix are directly related, they are very similar. However, there are certain differences that have an effect on the renaming algorithm which we will outline throughout this section. The Statix API is completely separate from the NaBL2 API. So, the main difference between the two implementations is that all the API calls are different, even though they often do the same.

4.6.1 The Renaming Algorithm with Statix

The basic structure of the algorithm is the same for the Statix implementation as it is for the NaBL2 implementation. Therefore, the `rename` strategy shown in Listing 4.17 is almost the same as its NaBL2 counterpart from Listing 4.5.

However, the Statix version expects an additional strategy argument `analyze`. This strategy is used to re-analyze the program in the capture detection step. The strategy contains some language specific parameters and thus has to be passed into our language-parametric algorithm, similar to how we pass the pretty-printing strategy into the `rename-action` in Listing 4.15. In the NaBL2 version these language parameters are passed implicitly.

```
1 rename(analyze | selected-term, new-name, path): ast -> renamed-ast
2   where
3     analysis := <get-analysis> ast
4     ; res-rel :=  <calc-resolution-relation(|ast)> analysis
5     ; selected-index := <find-name-index(|ast, res-rel)> selected-term
6     ; target-indices := <find-all-related-occs> (selected-index, res-rel)
7     ; renamed-ast := <rename-ast(|target-indices, new-name)> ast
8     ; check-capture(analyze|renamed-ast, res-rel, path)
```

Listing 4.17: Rename Strategy in Statix

The first step of the algorithm is to aggregate the program's name binding structure into a resolution relation. Since we extract the name binding information from the result of the static analysis, this step heavily depends on whether the Statix or NaBL2 solver was used to perform the analysis. Therefore the Statix version of this algorithm step substantially differs from the NaBL2 version. We explain the Statix implementation of this step in detail in the following subsection 4.6.2.

The resolution relation that results from the first step is represented as a list of pairs of `TermIndex` terms. Ergo the name binding structure of the program is represented in the same form in both implementations of the algorithm and therefore is agnostic to whether a language employs Statix or NaBL2. Since the following steps of the algorithm work on the resolution relation they are implemented in the same way as their NaBL2 counterparts.

In step two we find a name occurrence in the users selection. The only difference there is that we use `stx--get-ast-index` instead of `nabl2-get-ast-index` to read the `TermIndex` from a terms annotations. Step three calls the exact same primitive in the exact same way as the NaBL2 implementation to find all related name occurrence. Changing the identifier terms in step 4 also works like its NaBL2 counterpart, but like in step two we use the strategy from the Statix API to read the term indices.

To check for capture in the last algorithm step we need to reanalyze the changed AST. Executing the static analysis from Stratego is done different in Statix than it is in NaBL2, so we explain this step in detail in subsection 4.6.3. The error handling as well as the user interface integration are equal in both the Statix and NaBL2 implementation.

4.6.2 Calculating the Resolution Relation with Statix

One of the main differences between NaBL2 and Statix is that the scope graph built from a Statix specification doesn't contain references and declarations. This is somewhat inconvenient for us and makes it more difficult to gather the name binding information of a program and assemble it into a resolution relation. Since we cannot just get all occurrences from the static analysis result through a simple API call like we did in the NaBL2 implementation, we need to traverse the AST and find the occurrences ourselves. Listing 4.18 shows the strategies we implemented for this algorithm step.

We traverse the AST using the `collect(s)` strategy which applies its strategy argument to every node an collects the result in a list if the passed strategy succeeds. We are passing `collect` the two strategies `get-dec-ref-pair` and `get-dec-reflexive-pair` linked with the deterministic choice combinator. Ergo we first check if a term is a reference and resolve it to a

declaration to create a resolution pair. If the term is not a reference we check if its a declaration to create a reflexive resolution pair.

```
1  calc-resolution-relation(|ast): analysis -> relation
2    where
3       relation := <collect(get-dec-ref-pair(|analysis)
4                            <+ get-dec-reflexive-pair(|analysis))> ast
5
6  get-dec-ref-pair(|analysis): t -> (ref-index, dec-index)
7    where
8      <is-string> t
9      ; dec := <stx--get-ast-property(|analysis, Ref())> t
10     ; ref-index := <stx--get-ast-index> t
11     ; dec-index := <stx--get-ast-index> dec
12
13 get-dec-reflexive-pair(|analysis): t -> (dec-index, dec-index)
14   where
15     <is-string> t
16     ; dec := <stx--get-ast-property(|analysis, Prop("decl"))>
17     ; dec-index := <stx--get-ast-index> dec
```

Listing 4.18: Calculating the Resolution Relation with Statix

Both strategies first check if the current term is a string, as we expect any name to be an alphanumerical identifier. Statix adds *properties* to AST nodes which contain information from the result of the static analysis and can be accessed through the `stx--get-ast-property` API strategy. In `get-dec-ref-pair` we extract the builtin `Ref` property, which contains the `TermIndex` of the corresponding declaration.

For declarations there exists no builtin property, therefore we extract the custom property represented by `Props("decl")` from a term to check if it is a declaration. In order for the Statix solver to add the `decl` property to declaration nodes, we need to extend the Statix specification. In Listing 4.19 we display the `declareType` predicate of the Tiger language where we added the `decl` property on line 3.

```
1  declareType(scope, name, T) :-
2    scope -> Type{name} with typeOfDecl T,
3    @name.decl := name,
4    typeOfDecl of Type{name} in scope |-> [(_, (_, T))].
```

Listing 4.19: Setting the Declaration Property

4.6.3 Checking for Capture with Statix

The basic process of capture detection is the same for NaBL2 and Statix: we compare the program's name binding structure before and after the renaming transformation. For that, we need to trigger the static analysis on the changed AST and calculate the resolution relation again. Executing the analysis from Stratego is slightly more complex using the Statix API than when using the NaBL2 API, so we encapsulated the interaction with the API in the strategy `rerun-analysis`, shown in Listing 4.20.

Unlike when using the NaBL2 API, we need to explicitly provide some language-specific parameters when executing the static analysis through the Statix API. That is why we need to pass that API call into our language-parametric algorithm as the strategy argument `analyze`. Stratego automatically generates the strategy `editor-analyze` for every language project that

uses Statix which calls the API with the language-specific parameters. The `editor-analysis` for the Tiger language can be seen on line 18 as an example.

To start the analysis, we need to create an `AnalyzeSingle` term which contains the file path to the program and a `Changed` term which in turn contains the new AST and the old analysis object. This term then serves as an input for the language-specific `analyze` strategy. From the term returned by the analysis we then gather the newly analyzed term and the new analysis object which we use to recalculate the resolution relation. The rest of the `check-capture` strategy works the same as its NaBL2 equivalent. We sort both the old and the new resolution relation and compare them to check for name capture.

```
1  check-capture(analyze|renamed-ast, res-rel, path) =
2    (new-ast, new-analysis) := <rerun-analysis(analyze)> (path, renamed-ast)
3    ; new-res-rel := <calc-res-rel(|new-ast); qsort(res-pair-gt)> new-analysis
4    ; old-res-rel := <qsort(res-pair-gt)> res-rel
5    ; <eq>(old-resolution-relation, new-resolution-relation)
6      <+ add-error(|"Capture detected", "This renaming leads to name capture")
7
8  rerun-analysis(analyze): (path, renamed-ast) -> (new-ast, new-analysis)
9    where
10     input := <make-analysis-input> (path, renamed-ast)
11     ; Full(new-ast, FileAnalysis(_, new-analysis), _, _,_) := <analyze> input
12
13 make-analysis-input: (path, renamed-ast) -> AnalyzeSingle([(path, change)])
14   where
15     old-analysis := <stx--get-ast-analysis> renamed-ast
16     ; change := Changed(renamed-ast, old-analysis)
17
18 editor-analyze = stx-editor-analyze(pre, post|"statics", "programOk")
```

Listing 4.20: Capture Detection in Statix

4.7 Renaming over Multiple Files

The two Rename implementations we've presented in the previous two sections only work on programs that are confined to a single file. However, most computer programs are distributed over multiple modules which usually are persisted in separate files on disk. To make our solution really practical we need to extend it to so it can deal with multi-file programs. In this section we describe how we further developed our Statix implementation so it can perform renaming across multiple files. We also extended our NaBL2 version with the capability to deal with multi-file program but omit a detailed description as it works similarly.

While the multi-file capability adds some complexity to our solution, it does not change the fundamental algorithm flow. Therefore the `rename` strategy shown in Listing 4.21 looks similar to the single-file version shown in Listing 4.17. The main difference is that we now work with multiple ASTs instead of just one, as the source code in each file is represented as a separate AST.

Since we only get passed the AST of the file currently open we need to first load all the ASTs present in the project from the solvers's context using the `get-project-analyzed-asts` API call. This returns a list of pairs where the left subterm is the path of the file and the right subterm the AST of the file.

The solver creates an analysis object for each file and one analysis object which contains the analysis result for the whole project. The project analysis object specifically contains in-

formation of name bindings that span multiple files and therefore we query that object when building the resolution relation. To build the resolution relation for a multi-file program we execute the `calc-resolution-relation` on each AST in the project and concatenate the results.

Finding a name occurrence in the user's selections works exactly the same as in the single file versions of the algorithm. Finding all the occurrences to rename also works nearly identical since the `TermIndex` terms contain the path of the file where they occur. However, we additionally assemble all the paths of the files where we need to change identifiers in the strategy `find-rename-target`.

```
1  rename(analyze|selected-term, new-name): selected-ast -> renamed-asts
2    where
3      asts := <get-analysed-asts>
4      ; analysis := <get-project-analyzed-asts>
5      ; res-rel :=  <calc-resolution-relation-project(|analysis)> asts
6      ; selected-index := <find-name-index(|selected-ast, res-rel)> selected-term
7      ; (paths, target-indices) := <find-rename-target> (selected-occ, res-rel)
8      ; renamed-asts := <rename-asts(|target-indices, paths, new-name)> asts
9      ; check-capture(analyze|renamed-asts, resolution-relation)
10
11 calc-resolution-relation-project(|analysis): asts -> res-rel
12   where
13     res-rels := <mapconcat(calc-resolution-relation(|analysis))> asts
14
15 find-rename-target: (selected-occ, res-rel) -> (paths, target-indices)
16   where
17     target-indices := <find-all-related-occs> (selected-occ, res-rel)
18     ; paths := <get-paths-to-rename> target-indices
19
20 get-paths-to-rename: target-indices -> paths
21   where
22     paths := <map(get-paths);make-set> target-indices
```

Listing 4.21: Multi-File Renaming

Step four of our algorithm, where we actually perform the program transformation, is marginally more complex when dealing with multi-file programs. The implementation of this step is shown in Listing 4.22. Like when building the resolution relation we process each AST separate by mapping the `rename-ast` strategy over the list of all ASTs in the project.

```
1  rename-asts(|target-indices, paths, new-name): asts -> renamed-asts
2    where
3      renamed-asts := <map(rename-ast(|target-indices, paths, new-name))> asts
4  rename-ast(|target-indices, target-paths, new-name):
5    (path, ast) -> (path, ast, renamed-ast)
6    where
7      if (<path-in(|target-paths)> path) then
8        renamed-ast := <bottomup(try(rename-term(|target-indices, new-name)))> ast
9      else
10       renamed-ast := ()
11     end
12 path-in(|paths): path -> <elem> (path, paths)
```

Listing 4.22: Multi-File Renaming Transformation

In order to not needlessly traverse every AST in the project we first check for each AST if its path is in the list of paths to rename which we gathered in the previous step. If that is the case, we change the identifiers through the same traversal that we use in the single-file version of the algorithm. The `rename-ast` strategy returns a triple of the file path, the original AST and the renamed AST. If the AST of a file was unaffected by the renaming we return an empty term in place of the renamed AST.

4.7.1 Checking for Capture in Multi-File Programs

As in the single file version, we reanalyze the program after the renaming transformation and check for capture by detecting changes in the name binding structure. However, performing a complete static analysis of the whole project would be quite wasteful, since the renaming likely only affected a few files. The Statix solver supports a minimal incremental analysis which caches and reuses a small part of the previous analysis result. This does add some complexity to this algorithm step which we show in Listing 4.23.

```
1  check-capture(analyze|renamed-asts, res-rel) =
2      (new-asts, new-analysis) := <rerun-analysis(analyze)> renamed-asts
3      ; new-res-rel := <calc-resolution-relation-project(|new-analysis)
4                        ; qsort(res-pair-gt)> new-asts
5      ; old-res-rel := <qsort(res-pair-gt)> res-rel
6      ; <eq>(old-res-rel, new-res-rel)
7        <+ add-error(|"This renaming leads to name capture")
8
9  rerun-analysis(analyze): renamed-asts -> (new-asts, new-analysis)
10   where
11     input := <make-analysis-input> renamed-asts
12     ; analysis-result := <analyze> input
13     ; new-asts := <get-new-asts(|renamed-asts)> analysis-result
14     ; new-analysis := <get-new-analysis> analysis-result
15
16 make-analysis-input: renamed-asts -> AnalyzeMulti(project-changes, file-changes)
17   where
18     project-analyses := <get-project-constraint-analyses>
19     ; analysis := <get-project-entry> project-analyses
20     ; project-changes := (".", Cached(analysis))
21     ; file-changes := <map(make-file-change(|project-analyses))> renamed-asts
22
23 make-file-change(|analyses): (path, ast, renamed-ast) -> (path, file-change)
24   where
25     (_, analysis) := <fetch-elem(get-entry(|path))> project-analyses;
26     if (<eq> (renamed-ast, ())) then
27       file-change := Cached(analysis)
28     else
29       file-change := Changed(renamed-ast, analysis)
30     end
```

Listing 4.23: Capture Detecion for Multi-File Programs

Like in the single-file version, we need to create a term to call the language-specific `analyze` strategy with. Instead of an `AnalyzeSingle` term we built an `AnalyzeMulti` term in the strategy `make-analysis-input`. The API strategy `get-project-constraint-analysis` returns a list that contains the analysis object of all files as well as one object for the whole project.

We create a `Cached` term for the project analysis to signal the solver we want to perform an incremental static analysis. We then loop trough all the ASTs and create a `Cached` term for the files that have not changed or a `Changed` term if a file has been affect by the renaming and thus needs to be reanalyzed.

From the new result of the solver we then extract the project analysis object and the reanalyzed ASTs inside the `rerun-analysis` strategy. We then pass these to the `calc-resolution-relation-project` strategy the same way did in step one of the algorithm. After that we compare the old and the new resolution relation to detect name capture like we did in the single-file implementation.

4.7.2 Reconstructing the Source Code

We need to pretty-print the renamed ASTs back into source code before we persist the files to disk. The `rename` strategy from Listing 4.21 returns a list of all the ASTs in the project. More specifically, each entry in the list is a triple consisting of the file path, the AST before the renaming and the AST after the renaming. However, we only want to change the content of a file if its corresponding AST was actually affected by the renaming.

We filter out all the unchanged ASTs in the strategy `construct-renamed-program` shown in Listing 4.24. To distinguish unchanged ASTs we simple check if the renamed AST is equal to an empty term. On the ones that have changed we construct the new source code using the language-specific `construct-textual-change` strategy which takes the old and the new AST as arguments.

```
1  construct-renamed-program(construct-textual-change): asts -> program
2    where
3      changed-asts := <filter(has-changed)> asts
4      ; program := <map(construct-file(construct-textual-change))> changed-asts
5
6  construct-file(construct-textual-change): (path, ast, renamed-ast)->(path, text)
7    where
8      (_, _, text) := <construct-textual-change> (ast, renamed-ast)
9
10 has-changed: (_, _, renamed-ast) -> <id>
11   where
12     <not(eq)> (renamed-ast, ())
```

Listing 4.24: Reconstruct Source Code

4.8 Integrating Our Solution

During development, we kept the source code of our algorithm in the language project. This allowed us to just rebuild the project after any change and tests its effect immediately on a program in the target language. Although this was very convenient while the implementation was still under construction, it also made it necessary to copy the source code into every language project that we used to test our solution.

Since it is our goal to provide a comfortable user experience to language engineers using Spoofax, we aim to make the integration of our renaming refactoring into a language project as easy as possible. We integrated the source code, both the Java and Stratego part, directly into Spoofax. Thus, our solution can be rolled out as part of the Eclipse plugin.

4.8.1 Making It Part of Spoofax

Spoofax is a very big and complex software project and describing its architecture in detail would go beyond the scope of this thesis, we refer the interested reader to the official Spoofax documentation [15] and the paper by Kats and Visser[28]. Instead, we give a brief overview of its composition and focus on the parts that we had to extend in order to add renaming capability to Spoofax.

Looking from the top, Spoofax consists of 21 modules. We had to change the following three modules in order to integrate our solution. The `nabl` module contains the implementation of Statix and NaBL2, as well as everything else related to name binding and type checking. Naturally this is the module where we placed the body or our solution. The `spoofax` module contains the core functionality of Spoofax and manages the dependencies between the other modules. The `spoofax-eclipse` module packages all relevant modules and rolls them out as an Eclipse plugin.

Within the `nabl` module are two projects that contain the Stratego API used to interact with the name binding and type analysis: `nabl2.runtime` and `statix.runtime`. We added the Stratego code of our Statix and NaBL2 implementation into the respective project.

The primitives for NaBL2 and Statix are contained in the projects `nabl2.solver` and `statix.solver` respectively. However, these primitives libraries are only usable within their respective languages. This meaning it is not possible to call a primitive defined in `statix.solver` from an NaBL2 language project and vice-versa. Therefore, we had to create a new library that would be accessible from both languages.

To achieve this, we created a new Maven module `renaming.java` and placed the implementation of the union-find algorithm and the accompanying unit tests in there. To make it accessible from both languages, we registered this module as a dependency in both `nabl2.build` and `statix.build`.

```
1 public class RenamingLibrary extends GenericPrimitiveLibrary {
2   public static final String name = "RenamingLibrary";
3   public static final String REGISTRY_NAME = "RENAMING";
4
5   @Inject
6   public RenamingLibrary(@Named(name) Set<AbstractPrimitive> primitives) {
7     super(primitives, RenamingLibrary.REGISTRY_NAME);
8   }
9 }
```

Listing 4.25: RenamingLibrary

We also registered the dependency in the `org.metaborg.spoofax.core` module, in which all the primitives libraries are bundled. Within this module, each library is represented as a sub-type of `IOperatorRegistry`, so we created the class `RenamingLibrary` shown in Listing 4.25. Further, we bound the particular primitive we added to the library in the method `SpoofaxModule.bindAnalysis` with the statements shown in Listing 4.26.

```
1 Multibinder<AbstractPrimitive> renamingLibrary = Multibinder.newSetBinder(
2   binder(), AbstractPrimitive.class, Names.named(RenamingLibrary.name));
3
4 bindPrimitive(renamingLibrary, FindAllRelatedOccurrencesPrimitive.class);
```

Listing 4.26: Bind Primitive to Library

Lastly, we included the new library in the Eclipse plugin through which Spoofax gets integrated into the IDE. While there exists an IntelliJ plugin for Spoofax as well, it is not

actively maintained. Therefore, we omitted to integrate our solution into it. How the plugin is built is defined in the module `org.metaborg.spoofax.eclipse.feature` this is an Eclipse Feature Project. To have the library loaded into eclipse, we extended this projects `feature.xml` as shown in Listing 4.27.

```
1   <plugin id="renaming.java" version="2.6.0.qualifier" unpack="false"/>
```

Listing 4.27: Registering Library in Eclipse Plugin

4.8.2 Adding Renaming to a Language

With our language-parametric refactoring now integrated into Spoofax, it can be added to any language project in a few simple steps. The language project needs to implement a language-specific renaming strategy that calls the the integrated rename strategy. Listing 4.28 shows such a strategy for a language that uses NaBL2. The language-parametric strategy `nabl2-rename-action` is imported from the NaBL2 runtime. It gets passed the language-specific strategies `construct-textual-change` and `editor-analyze` for pretty-printing and static analysis. The third strategy parameter needs to be set to `id` if the language supports multi-file analysis or `fail` if it does not.

```
1  module renaming
2  imports
3    nabl2/runtime
4    analysis
5    pp
6  rules
7    my-rename = nabl2-rename-action(construct-textual-change, editor-analyze, id)
```

Listing 4.28: Language-specific Renaming Strategy NaBL2

Adding the Rename refactoring to a language using Statix works almost identical and an exmaple is shown in Listing 4.29. The only difference is that the integrated renaming strategy is called `rename-action` and is imported from the module `statix/runtime/renaming`. To integrate the refactoring into the UI, the language-parametric strategy needs to assigned to a menu item as we have shown in subsection 4.5.6.

```
1  module renaming
2  imports
3    nabl2/runtime
4    analysis
5    pp
6  rules
7    my-rename = rename-action(construct-textual-change, editor-analyze, id)
```

Listing 4.29: Language-specific Renaming Strategy Statix

Chapter 5

Testing Renaming

5.1 Test Approach

Since our solutions is designed to be language-parametric, it is only logical to test our implementation on an array of different programming languages. However, we are limited to languages for which Spoofax implementations already exist, although they do not need to be complete. We need at least a syntax definition in SDF3 to parse test programs to an AST. Then, we run the static analysis according to either an NaBL2 or Statix specifications.

The languages we chose to test are shown in Table 5.1. This set of languages covers a sizeable spectrum of paradigms, purposes, and features. Therefore, they can evaluate the completeness of our language-parametric solution to a satisfying degree.

Name	**Description**	**Static Semantics Meta-Language**
Tiger	Functional Toy-Language	NaBL2 & Statix
Chicago	Functional Multi-Module Toy-Language	Statix
MiniJava	General-Purpose Object-Oriented Language	NaBL2
WebDSL	Domain-Specific Language to Develop Web Applications	Statix
Statix	Declarative Meta-Language to Define Static Semantics	NaBL2

Table 5.1: Test Languages

Since our renaming transformation is self-contained, it is perfectly suitable to evaluate its soundness through unit testing. The fact that unit tests can be automated contributes to the integrity of our solution as it makes our results easily reproducible. The simple structure of unit tests allows them to be developed quickly. This requires little effort to add more test cases in case we wish to test more languages or name binding patterns in future work. Each renaming unit test is composed of the following four elements:

1. A *source program* in which we want to rename an entity
2. A *selected occurrence* of the name of the entity we want to rename
3. A *target name* to which we want to change the entities current name to
4. A *target program* in which the target entity was correctly renamed

A unit test case passes if the source program and the target program are α-equivalent. An exception to this definition are test cases that target the name capture detection. In this instance, the test passes if the transformation aborts with an error.

5.2 Test Case Implementation

5.2.1 Introducing SPT

We implemented our unit tests using the Spoofax Testing Language (SPT) [27]. SPT is a declarative meta-language specifically designed to test certain language aspects, such as parsing or type checking, and allows for the test-driven development [6] of programming languages. An SPT *test case* consists of an input *program fragment* and an output *expectation*. Test cases that target similar features of a language can be grouped into a *test suite*.

```
1  module name-resolution
2  language Java
3
4  test param name resolution [[
5    public class Foo {
6      int id(int [[param]]) {
7        return [[param]];
8      }
9    }
10 ]] resolve #2 to #1
```

Listing 5.1: Name Resolution Test Case

```
1  module optimization
2  language Assembler
3
4  test optimize inc [[
5    mov 12(fp), r1
6    add r1, 1, r1
7    mov r1, 12(fp)
8  ]] run optimize to [[
9    inc 12(fp)
10 ]]
```

Listing 5.2: Optimization Test Case

An example test case `param name resolution` is shown in Listing 5.1. The input program fragment which is under test is delimited by double square brackets on line 4 and 10 respectively. Inside that code fragment, we see that both occurrences of the `param` name are also enclosed by double square brackets, which represents a selection. The expectation `resolve #2 to #1` invokes the name binding analysis on the code fragment and then checks if the reference at selection 2 resolves to the declaration at selection 1, determining the outcome of the test case.

The test case `optimize to inc` in Listing 5.2 shows how SPT can trigger a transformation, in the form of a Stratego strategy, and then compare the result to another given program fragment, using the `run` expectation. In this particular case, we run the `optimize` strategy on the code fragment consisting of the three assembler instructions on the lines 5 - 7 and then check if the resulting program is equal to the instruction on line 9.

5.2.2 Extending SPT

To evaluate our renaming solution, we need two kinds of test cases. The first case calls the renaming strategy and then compares the renamed program to a target program, as shown in Listing 5.3. The second case is used to test capture detection, which manifests as the renaming strategy resulting in an error, as shown in Listing 5.4.

Unfortunately, there was no test expectation available yet that was suitable for evaluating our renaming transformation. Luckily, the modular architecture of the SPT language definition and runtime environment made it easy to extend the language to suit our needs.

Specifically, we extended the `run` expectation that is shown in 5.2. It was not possible to pass term arguments to a strategy. Since the `rename` strategy takes a selected identifier occurrence and a new name as arguments, we developed an extension to allow this.

It was also not possible, that a strategy returns an error as the expected result of a test case. With our extension, it is possible to allow strategies to be able to fail, while still letting the test case pass, using the output-part `fails` of the expectation. The implementation of the extension is explained in detail in Section 5.3.

```
module renaming
language Haskell

test rename function [[
  [[plus]] :: Num -> Num -> Num
  plus x y =  x + y
]] run rename(|#1, "add") to [[
  add :: Num -> Num -> Num
  add x y =  x + y
]]
```

Listing 5.3: Renaming Test Case

```
module capture
language Tiger

test name capture [[
  let
    var bar := 10
  in
    let
      var foo := 100
        in
      bar + [[foo]]
    end
  end
]] run rename(|#1, "bar") fails
```

Listing 5.4: Capture Test Case

5.3 Developing the SPT Extension

In this subsection, we outline the changes we had to make to the SPT implementation in order to use it for the evaluation of our renaming implementation. However for the sake of brevity, we only present a simplified view of the modifications we made. The source code of SPT is open source and we refer the interested reader to look at the implementation details on GitHub[1].

5.3.1 Expanding the Syntax

As a first step, we extend the syntax in order for the parser to allow strategy calls with arguments via the `run` expectation. The parser also needs to recognize the `fails` part of an expectation, so we can express that a test is designed to check whether a strategy fails in the context of a specific test case. SPT is implemented in Spoofax and therefore its syntax is declared in SDF3 [29]. For an introduction to SDF3 see Section 4.3.

```
TestDecl.TestDecl = <test <Description> [[
                        <Fragment>
                    ]] <Expectation>>

Expectation.Run = <run <STRAT><TermArgs?> <OnPart?> <SResult?>>

SResult = ToPart
SResult.Fails = <fails>

TermArgs.TermArgs = <(|<{Arg ","}*>)>
Arg.Int = INT
Arg.String = STRING
Arg.Ref = SelectionRef

SelectionRef.SelectionRef = <#<INT>>
```

Listing 5.5: SPT Syntax Extension

[1] https://github.com/metaborg/spt

Listing 5.5 shows an excerpt of SPT's syntax definition which is relevant to our modification. On line 1 we see the definition of a test case, starting with the `test` keyword followed by an arbitrary description. The fragment enclosed by the double square brackets holds the program under test. At the end of the test case is the expectation, which the test is declared to fulfill. This is the part extended to make it possible for the renaming algorithm to be called.

On line 5 we can see the modified `Run` constructor that allows calling strategies with term arguments. The question mark `?` next to the sort names indicate that they are optional. After the `run` keyword and the name of the strategy the parsers accepts now an `TermArgs` sort. The `OnPart` sort was already part of the definition and allows the strategy to be executed on a specific snippet of code instead of the whole program inside the test case. As this feature is not relevant to testing our refactoring we just leave it as is.

Lastly, the parser expects a term that determines what result we expect after running the strategy. This can be term of sort `ToPart`, which allows us to compare the test program after the application of the strategy with a specific target program. This feature was already present as well and fulfills our requirement without any modification. Instead of `ToPart`, the parser also accepts the string literal `fails`, to express that the strategy is expected to abort with an error.

In place of the `TermArgs` sort, the parsers accepts a comma-separated list of term arguments that is prefixed by the pipe operator `|` and enclosed in brackets. As arguments, it excepts integer numbers, string literals or selection references. A selection reference is used to point to a segment of the test case program fragment by enclosing it in double square brackets and referring to it with the pound sign and a one-based index. An example can be found in Listing 5.3 on line 5. We use this feature to simulate how a programmer would select an identifier in the text editor before triggering the renaming.

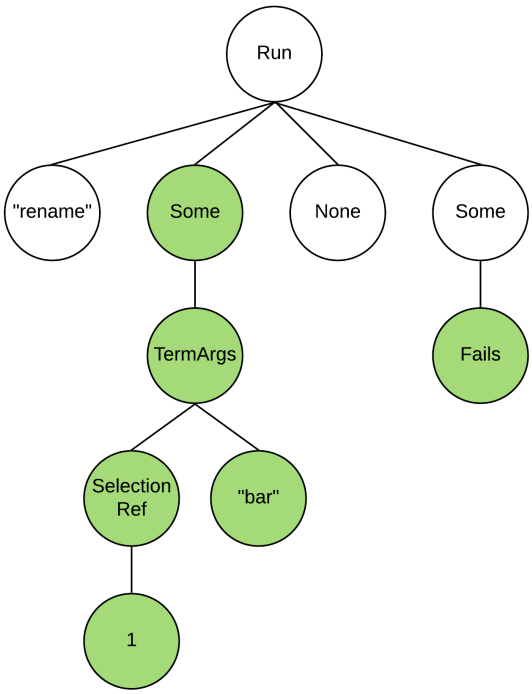


Figure 5.2: Run Expectation Term

Figure 5.2 shows the term that the parser produces for the expectation in Listing 5.4. The `Some` and `None` terms are generate for all the child nodes which are defined to be optional. The subterm that stem from our syntax expansion are highlighted in green.

5.3.2 Adapting the Runtime

From the syntax definition we wrote in SDF, Spoofax generates a SGLR [48] parser, which transforms SPT source code into an AST. This AST is then fed into the SPT runtime, which executes the test cases and determines whether they failed or passed. Since we expanded the syntax, the parser is going to produce different terms for the `run` expectations and we adapted the runtime so it can handle them.

The SPT runtime is implemented in Java. Its entry point is in the `SPTRunner` class in the method `test`. Here the .spt text files are read from disk and passed to the generated parser, which is also implemented in Java. The AST generated by the parser, where each node is of a sub-type of `IStrategoTerm`, is then fed into the `SpoofaxTestCaseExtractor`, where it is transformed into a list of `TestCase` objects. This extraction had to be adapted in order to consider the changed `run` expectation term.

Each test case is then passed on to the method `TestCaseRunner.run`, where they are evaluated isolated from each other, as unit tests should be. The outcome of the test cases are then collected in a `TestResult` object and passed back to the `SPTRunner`. This process is visualized in Figure 5.3 as a sequence diagram.

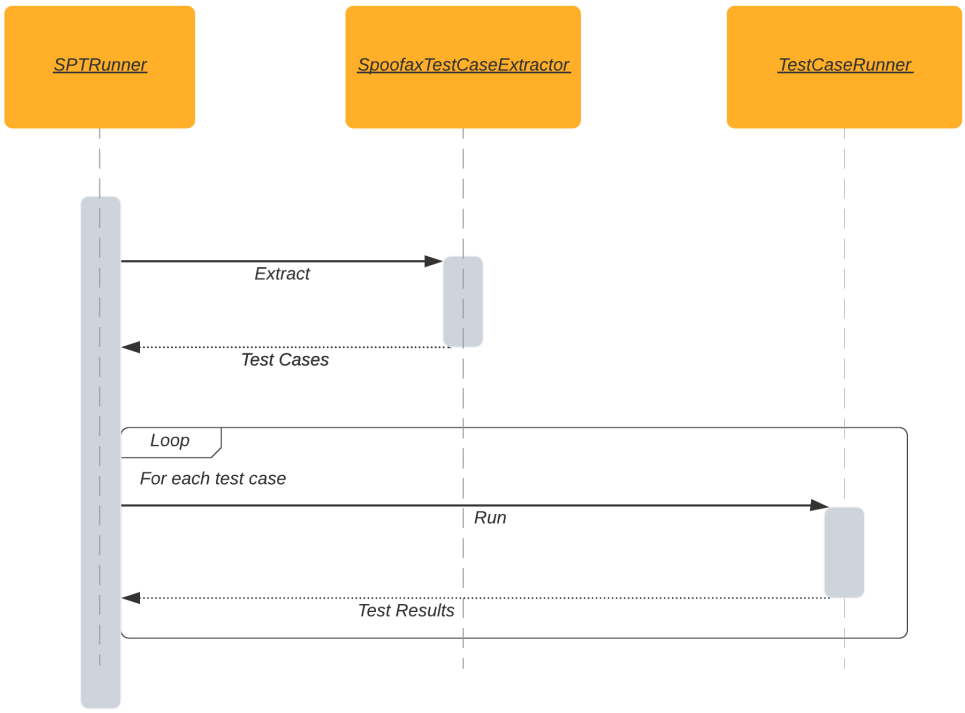


Figure 5.3: Sequence Diagram of Test Suite Execution

Inside the method `TestCase.evaluateExpectations` the actual execution of the test case happens, where a value object representing the expectation is handed to an evaluator service, which assesses whether the expectation is met or not. In case of the `run` expectation, a `RunStrategoExpectation` object is handed to the `RunStrategoExpectationEvaluator`. We modified both these classes, so that they accept a list of term arguments to call the strategy with.

Inside of `RunStrategoExpectationEvaluator.evaluate`, the test case fragment and the strategy call are then passed to the method `StrategoCommon.invoke`. `StrategoCommon` is an adapter class which relays the strategy calls to the Java implementation of the Stratego interpreter. We modified the adapter as well so it would be able to deal with strategy calls with term arguments. The output of the interpreter is then returned to the `RunStrategoExpectationEvaluator`, where it is inspected and the expectation is determined passed or failed. We adjusted this part, so that the test would still pass, even if the strategy failed, in case the test case was expecting it to fail. The sequence diagram for the evaluation is shown in Figure 5.4.

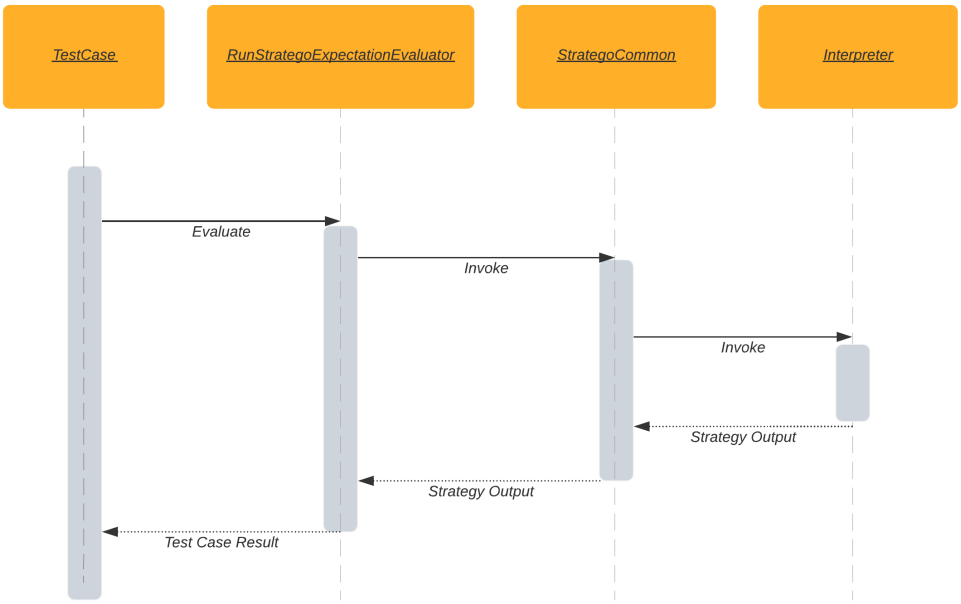


Figure 5.4: Sequence Diagram of Test Case Evaluation

5.4 Test Cases in Detail

In this section, we will look at some interesting test cases in detail and explain how they contribute to the evaluation of our refactoring. The listings of all test cases used in the assessment of our solution can be found in Appendix A.3.

5.4.1 Tiger

We will start by looking at some test cases in Tiger, a functional, statically-typed toy language developed to educate students about compiler construction [3]. Its simplicity makes it a prime candidate to test our algorithm against very basic name binding patterns. The implementation of Tiger's name binding rules are available both in NaBL2 as well as Statix. This allows us to test both implementations of our renaming tool.

Lexical Scoping

Tiger is lexically scoped and offers five different named language entities, that our tests need to cover: variables, functions, function arguments, types and fields. The let bindings in Tiger are sequential, which means that entities can only be used after they are declared.

To check whether our renaming algorithm can deal with lexical scoping, we devised the test shown in Listing 5.6. In this test case, we rename the function `plus` to `add`, where the scopes of the function declaration and the function call are adjacent. A correct renaming should change both the function declaration and the function call.

```
1  test rename function from declaration [[
2  let
3      function [[plus]](a : int, b:int) : int = (
4          a + b
5      )
6  in
7      plus(1,1)
8  end
9  ]] run rename(|#1, "add") to [[
10 let
11     function add(a : int, b:int) : int = (
12     a + b
13     )
14 in
15     add(1,1)
16 end
17 ]]
```

Listing 5.6: Renaming Function from Declaration

The test case in Listing 5.7 also targets the Lexical Scoping name binding pattern, this time for the language entity Variable. Another aspect of our solutions this test covers is that the selected occurrence can be either a declaration or a reference. No matter which one is selected, both the variable declaration and reference should be changed.

```
1  test rename variable from reference [[
2  let
3      var msg := "Hello World"
4  in
5      print([[msg]])
6  end
7  ]] run rename(|#1, "message") to [[
8  let
9      var message := "Hello World"
10 in
11     print(message)
12 end
13 ]]
```

Listing 5.7: Renaming Variable from Reference

To see if our solution also works on more complex programs we revisit the Foo Challenge, see Listing 5.8. In this example we have multiple program entities with the same name and a correct renaming would only change the identifiers of the selected entity. In this test case

we attempt to rename the type `foo`, which would require a change of its declaration on line 18, as well as the two references on lines 21 and 29.

```
1  test Foo Challenge [[
2  let
3      type foo = {
4          foo : string
5      }
6      function foo (foo: [[foo]]) = (
7          let
8              var foo := foo.foo
9          in
10             print(foo)
11         end
12     )
13 in
14     foo(foo{foo = "foo"})
15 end
16 ]] run rename(|#1, "bar") to [[
17 let
18     type bar = {
19         foo : string
20     }
21     function foo (foo: bar) = (
22         let
23             var foo := foo.foo
24         in
25             print(foo)
26         end
27     )
28 in
29     foo(bar{foo = "foo"})
30 end
31 ]]
```

Listing 5.8: Renaming Variable from Reference

Shadowing

As we have discussed at length previously, the presence of ambiguous names is one of the principal challenges when developing a sound refactoring algorithm. In case a reference could resolve to two nearby declarations, it is usually bound to the declaration which is ”closer” to that reference. This name binding pattern is referred to as *Shadowing*.

An example of this pattern can be seen in Listing 5.9. There we have two declarations of a variable `scope` which are both reachable from the lone reference. A correct renaming would only change the variable that is declared closer to the selected reference and leave the outer declaration, the one that is being shadowed, as is.

```
test rename variable with shadowing [[
  let
    var scope := "Outer"
  in
    let
      var scope := "Inner"
    in
      print([[scope]])
    end
  end
]] run rename(|#1, "innerScope") to [[
  let
    var scope := "Outer"
  in
    let
      var innerScope := "Inner"
    in
      print(innerScope)
    end
  end
]]
```

Listing 5.9: Shadowing

Instance Scoping

Many programming languages offer the feature of declaring user-defined types, which encapsulate a set of fields. Prominent examples would be classes in Java or structures in C. Usually each instance of such a user-defined type has its own scope, from which the populated values of the fields are accessible.

```
test rename field [[
  let
    type student = {
      [[name]] : string, age : int
    }
    var student1: student := student{name = "Alan Turing", age = 109 }
  in
    student1.name
  end
]] run rename(|#1, "fullName") to [[
  let
    type student = {
      fullName : string, age : int
    }
    var student1: student := student{fullName = "Alan Turing", age = 109 }
  in
    student1.fullName
  end
]]
```

Listing 5.10: Renaming Field

In Tiger we have the entity type Type which exhibits such a name binding structure. In Listing 5.10, we are renaming the field `name` of the type `student` to `fullName`. For this to result in an α-equivalent program, the declaration and both the references need to be changed. One is part of the constructor on line 7 and the other is a the field access on line 9.

Inadvertent Name Capture Detection

A sound renaming algorithm needs to make sure that no name capture is introduced through the program transformation, as this might alter the program's behaviour. We described this problem in detail in Section 2.6 and are going to revisit the example from Listing 2.9 to check if our renaming algorithm handles this correctly.

For this we use the extension for SPT we developed that allows us to check if a strategy fails, which in the case of capture would be the correct behavior. In Listing 5.11 the renaming of the variable `bar` should fail, as it would lead to capture when the reference on line 8 would change.

```
1  test rename variable with capture [[
2    let
3      var [[bar]] := 10
4    in
5      let
6        var foo := 100
7      in
8        bar + foo
9      end
10   end
11 ]] run rename(|#1, "foo") fails
```

Listing 5.11: Capture Detection

Invalid User Input

We aim to build a robust solution, that can deal with invalid user input. To test this, we can also use the `fails` extension we developed for `run` expectation. In Listing 5.12, the user attempts to rename the function `print`, which is built into the Stratego runtime library. Since this library is tied into the language project as a compiled dependency, that is clearly not possible and needs to result in an error.

```
1  test rename built-in function [[
2    let
3      var msg := "Hello World"
4    in
5      [[print]](msg)
6    end
7  ]] run rename(|#1, "println") fails
```

Listing 5.12: Renaming Built-in Function

Another example of a term that cannot be renamed is a string literal, see Listing 5.13. Literals are not part of a program's name binding structure and thus our algorithm has no way of dealing with them, leading it to abort with an error.

```
1 test select literal  [[
2   let
3     var msg := [["Hello World"]]
4   in
5     print(msg)
6   end
7 ]] run rename(|#1, "Hi World") fails
```

Listing 5.13: Renaming Literal

There are some cases of an invalid user input however, which we cannot cover through an SPT test case. In Listing 5.14, the user selected the keyword `var`, which as a builtin part of the language's syntax clearly cannot be changed. We cannot detect this in the `run` expectation, because it works with the AST representation of the test fragment, in which the keywords are not present anymore. This leads to this test case being marked as failed by with the message "Could not resolve this selection to an AST node.".

```
1 test rename keyword [[
2 let
3   [[var]] msg := "Hello World"
4 in
5   print(msg)
6 end
7 ]] run rename(|#1, "println") fails
```

Listing 5.14: Renaming Keyword

5.4.2 Chicago

Chicago is a simple functional programming language which was developed to test and showcase the capabilities of Statix. The language supports multi-module programs and therefore we chose it to test if our renaming algorithm behaves correctly if the name occurrences of a language entity are spread out over multiple files. Since it is not possible to spread a SPT test case over multiple files, we had to execute these tests by hand.

Chicago is very similar to Tiger in terms of language features and name binding patterns. Therefore, we wrote test cases very resembling to the ones we wrote for Tiger to evaluate if the multi-file version of our renaming algorithm works properly. Since these test cases are very similar, we are not going to discuss them again here. The module-spanning name binding is the only pattern that Chicago uses and Tiger does not.

Module-Spanning Name Binding

While SPT doesn't work with multi-file programs, we can define multiple modules in the same file to simulate a multi-file program. This allows us to at least test our solution on a module-spanning name binding pattern, just without the multi-file persistence aspect.

In Listing 5.15, we show a test case targeting a module-spanning name binding. Module `Alpha` defines a variable `a` which we want to rename to `x`. Module `Beta` imports module `Alpha` and thus can access all its declarations. Therefore, when renaming `a` we need to change the references in all other modules that import the variable's enveloping module.

```
1  test rename variable across modules [[
2    module Alpha {
3      def [[a]] = 1
4    }
5    module Beta {
6      import Alpha
7      def b = a + 1
8    }
9  ]] run rename(|#1, "x") to [[
10   module Alpha {
11     def x = 1
12   }
13   module Beta {
14     import Alpha
15     def b = x + 1
16   }
17 ]]
```

Listing 5.15: Renaming Variable Across Modules

Having a renaming algorithm with name capture detection is even more valuable when working with multi-file programs. In a small single-file program, an attentive developer might spot name capture by himself. However in a big multi-file project, it becomes virtually impossible to manually detect the problem.

In Listing 5.16, we show an example of name capture that happens in another module than the one from which the renaming is triggered. Renaming the variable `a` to `b` would cause the reference to `a` on line 8 to be captured by the declaration on line 7.

```
1  test capture detection across modules [[
2    module A {
3      def [[a]] = 1
4    }
5    module B {
6      import A
7      def b = 2
8      def c = a + 1
9    }
10 ]] run rename(|#1, "b") fails
```

Listing 5.16: Name Capture Detection Across Modules

5.4.3 MiniJava

Java is an object-oriented general-purpose programming language developed by Oracle. It is one of the most popular programming languages, ranking second in both the TIOBE [11] and PYPL [12] index at the time of writing. Its wide user base alone would be reason enough to consider it in our tests, but Java also exhibits some interesting name binding rules which are rooted in its underlying object-oriented paradigm. Specifically, the inheritance relationship between classes and the use of qualified names led us to some compelling test cases. We used a subset of Java called MiniJava [4] for our tests.

Classes

One of the landmark features of object-oriented programming languages is the use of classes. A *class* is a user-defined type that encapsulates state as *fields* and provides *methods* that operate on those fields. Classes can be organized into *class hierarchies* and are *polymorphic* types, which means any child class provides the same interface as its parent class. As a result, any object of the child type can be used in place of an object of its parent type.

In Listing 5.17, the parent class `Foo` is renamed to `Bar`. A correct renaming needs to find all type references in the program and change them to the new identifier. The *access modifier* `public` declares that the class can be used from anywhere in the program. This is an example of a name binding pattern that has no restrictions on the locality of its occurrences and thus makes a complete search of the program necessary. As unit tests are limited to small single-module programs, admittedly this test is of limited significance in showing that.

```
1  test rename class from declaration [[
2    public class [[Foo]] {
3      public Foo create() {
4        Foo foo;
5        foo = new Foo();
6        return foo;
7      }
8    }
9
10   public class KidFoo extends Foo {
11     private Foo parent;
12
13     public Foo setParent(Foo foo) {
14       parent = foo;
15       return parent;
16     }
17   }
18 ]] run rename(|#1, "Bar") to [[
19   public class Bar {
20     public Bar create() {
21       Bar foo;
22       foo = new Bar();
23       return foo;
24     }
25   }
26
27   public class KidFoo extends Bar {
28     private Bar parent;
29
30     public Bar setParent(Bar foo) {
31       parent = foo;
32       return parent;
33     }
34   }
35 ]]
```

Listing 5.17: Renaming Class

Method Overriding

A child class can *override* a method of its parent class by providing a different implementation of a method with the same method signature, i.e. the same name, return type, and parameter list. So if a method is called on a reference of the parent type, the overridden implementation of one of its child types might actually be executed. The exact method implementation is chosen at runtime.

This feature is called *dynamic dispatch*. On line 16 of Listing 5.18 the method `getX` is called on a reference of type `Foo`. Both the class `Foo` and its child class `Bar` provide an implementation for this method. If this reference points to an object of type `Foo` or `Bar` is generally undecidable at compile time, hence the name *dynamic* dispatch.

This case is very interesting, as it is allows us to test if our solution works with multi-declaration binding pattern (see Section 2.3.2). The static name binding analysis will find two possible declarations for the reference represented by the call of `getX`. However even in the case of a multi-declaration binding, the name binding structure must remain intact to ensure behavior preservation. Therefore, both declarations need to be renamed.

```
1  test rename overridden method from ref[[
2    public class Foo {
3      public int getX() {
4        return 1;
5      }
6    }
7    public class Bar extends Foo {
8      public int getX() {
9        return 2;
10     }
11   }
12   public class Main {
13     public static int getNumber(Foo foo) {
14       return foo.[[getX()]];
15     }
16   }
17 ]] run rename(|#1, "getY") to [[
18   public class Foo {
19     public int getY() {
20       return 1;
21     }
22   }
23   public class Bar extends Foo {
24     public int getY() {
25       return 2;
26     }
27   }
28   public class Main {
29     public static int getNumber(Foo foo) {
30       return foo.getY;
31     }
32   }
33 ]]
```

Listing 5.18: Renaming Overridden Method

Qualified Names

Some programming languages allow names to be *qualified*. This means that additional information is provided at a reference site in order to make the name resolution unambiguous. In Java, the qualifier `this` points to the scope of the current object, forcing the name resolution to start from that scope instead of the scope the qualified reference occurs in.

In Listing 5.19, we can see an example of the common Setter pattern where both the function argument and the field have the same name `value`. The field reference is qualified with `this` to make it clear that it should resolve to the field declared on line 4.

As we can see on line 15, the renaming left the qualifier in place and only changed the identifier. In this example, keeping the qualifier would not be necessary, as the function argument and the field now have different names, making the name resolution unambiguous anyway. However, this is not trivial to detect and it is unclear how this could be done language-parametric. Thus, we leave it to future research. By just keeping the qualifier, we can at least ensure the name binding structure stays the same, even if it leaves some unnecessary code behind.

```
1  test rename qualified name [[
2    public class Foo {
3      private int value;
4
5      public Foo setValue(int value) {
6        return [[this.value()]] = value;
7      }
8    }
9  ]] run rename(|#1, "number") to [[
10   public class Foo {
11     private int number;
12
13     public Foo setValue(int value) {
14       return this.number = value;
15     }
16   }
17 ]]
```

Listing 5.19: Renaming Qualified Name

5.4.4 WebDSL

General-purpose programming languages, such as Python or PHP, benefit from a sizable user base which results in good tool support. Offering a language-parametric refactoring for such a language would yield limited benefit, as there already exist a number of language-specific solutions. However, for less popular or domain-specific languages which are not backed by big communities or corporations, our solution provides a plug-and-play feature to automate one of the most common tasks when developing software.

WebDSL [49] is a domain-specific language to develop web-applications with rich data models. It provides abstractions for common components of web-application such as relational queries, web pages and e-mail templates. WebDSL is an interesting target language for our tests as it is a fairly big DSL that has been used to develop multiple web-application which are used productively by thousands of users.

In the tests we have described in this chapter, the renamings always targeted general-purpose program entities, such as variables or functions, which occur in most programming

languages. DSLs usual provide program entities that are specific to the target problem domain and it is interesting to test our renaming algorithm on such domain-specific program entities.

```
1  test rename item [[
2    entity [[Item]] {
3      text : String
4    }
5
6    page grocery-list {
7      var newitem := Item{}
8      form {
9        input( newitem.text )
10       submit action{ newitem.save(); }{ "Add text" }
11     }
12     for( i: Item ){
13       div {
14         output( i.text )
15         submit action{ i.delete(); }{ "Remove" }
16       }
17     }
18   }
19 ]] run rename(|#1, "Grocery") to [[
20   entity Grocery {
21     text : String
22   }
23
24   page grocery-list {
25     var newitem := Grocery{}
26     form {
27       input( newitem.text )
28       submit action{ newitem.save(); }{ "Add text" }
29     }
30     for( i: Grocery ){
31       div {
32         output( i.text )
33         submit action{ i.delete(); }{ "Remove" }
34       }
35     }
36   }
37 ]]
```

Listing 5.20: Renaming Entity

Listing 5.20 shows a simple web-application written in WebDSL that allows the user to add items to a grocery list. It contains the entity `Item`, which represents a domain object, that can be persisted in a relation database in the backend and displayed on a webpage in the frontend. This is a good example of an abstraction that a DSL offers to make programs more expressive. Renaming the entity to `Grocery` should also change all references in the page `grocery-list`. This test shows that our renaming solution can deal with program entities both domain-specific and general-purpose.

Function Overloading

Function overloading is a name binding pattern that uses the function's parameter list, additionally to the name, for name resolution. Specifically, the types of the function arguments are needed to resolve a call to the correct declaration.

In Listin 5.21, we test our renaming algorithm on the overloaded function `ret`. One declaration of `ret` expects a string as an argument and the other declaration expects an integer. When renaming the function from a call, the name resolution algorithm needs to evaluate the type of the parameters passed to the function in order to find the correct declaration. Similarly, the refactoring should only change the name of one of the function declarations.

```
1  test rename overloaded func from ref [[
2    built-in
3      page root {}
4      function ret(s: String) { }
5      function ret(i : Int) { }
6
7      function testRet() {
8        [[ret]](1);
9      }
10 ]] run rename(|#1, "retInt") to [[
11   built-in
12     page root {}
13     function ret(s: String) { }
14     function retInt(i: Int) { }
15
16     function testRet() {
17       retInt(1);
18     }
19 ]]
```

Listing 5.21: Renaming Entity

5.4.5 Statix

We chose Statix [2] as another DSL to evaluate our solution against. Testing a renaming algorithm on a name binding language exhibits an interesting quasi-recursive quality. Since Statix's static semantics are implemented in its predecessor NaBL2 [1], it was an obvious candidate for testing our solution.

The test in Listing 5.22 shows a simple Statix specification with one predicate `typeOfExp` that, as the name suggests, checks the type of an expression. A predicate with that purpose can likely be found in every Statix specification, as expressions are a concept used in virtually every programming language.

While our test specification only has one rule for the predicate, a complete implementation will have many more and thus have many name occurrences of `typeOfExp`. For example, Chicago's specification contains 75 occurrences of the predicates name distributed over 12 modules/files. Changing the name of the predicate to `typeOfExpr` would require quiet a bit of manual editing.

Our automated Rename refactoring gets this change done in a few seconds. This case perfectly exemplifies how our language-parametric solution can increase the usability of DSLs like Statix that are developed with limited resources. Our refactoring can be added to any Spoofax language with little effort and provides a helpful tool for developers using the DSL.

However, this test case also shows one of the limitations of our solution. Our refactoring ignores the implied name binding between the sort `Exp` and the predicate `typeOfExp`. It would make sense to also rename the sort to `Expr` in order to consistently use the same abbreviation for the word "expression" within the specification. Since this kind of cross-entity name binding is not specified in Statix's name binding rules, our renaming ignores it.

```
1  test rename typeOfExp [[
2    module m
3    signature
4     sorts Exp constructors
5       Add: Exp * Exp -> Exp
6     sorts Type constructors
7       INT : Type
8
9    rules
10    typeOfExp : scope * Exp -> Type
11
12    [[typeOfExp]](s, Add(e1, e2)) = INT() :-
13    typeOfExp(s, e1) == INT(),
14    typeOfExp(s, e2) == INT().
15 ]] run rename-test(|#1, "typeOfExpr") to [[
16   module m
17   signature
18    sorts Exp constructors
19      Add: Exp * Exp -> Exp
20    sorts Type constructors
21      INT : Type
22
23   rules
24    typeOfExpr : scope * Exp -> Type
25
26    typeOfExpr(s, Add(e1, e2)) = INT() :-
27    typeOfExpr(s, e1) == INT(),
28    typeOfExpr(s, e2) == INT().
29 ]]
```

Listing 5.22: Renaming Predicate

Chapter 6

Evaluation

To evaluate our solution, we will revisit the common problems of implementing the Rename refactoring and check if and to what degree we were able to solve them. We described those problems at length in Chapter 2.

Our solution relies on a language-parametric name resolution algorithm to solve some of the problems for us. For example, a text search-and-replace approach to renaming struggles when encountering duplicate names. The name resolution algorithm can clear up such an ambiguity and allows us to change the correct identifiers.

However, more important is the name resolution algorithm completely encapsulates the complexity of language-specific name binding. Therefore, we achieve a separation of concern between name resolution and renaming, resulting in a rather simple renaming algorithm.

6.1 Coverage of Name Binding Patterns

One of the main challenges of developing a language-parametric renaming algorithm is there exists a theoretically infinite amount of different name binding patterns across programming languages. We solved this problem by abstracting a program's name binding structure to a *resolution relation* which contains all reference occurrences paired with the declaration occurrence they resolve to. Our implementation uses the solvers of NaBL2 and Statix to perform name binding analysis.

Language	**# Test Cases**	**Program entities**	**Name binding patterns**
Tiger	13	Variables, Functions, Types, Fields, Arguments	Lexical Scoping, Shadowing, Instance Scoping
Chicago	20	Variables, Functions, Types, Fields, Arguments, Modules	Lexical Scoping, Shadowing, Instance Scoping, Module-Spanning Scoping
MiniJava	22	Variables, Methods, Classes, Fields, Arguments	Lexical Scoping, Shadowing, Instance Scoping, Type Hierarchies, Method Overriding, Qualified Names
WebDSL	16	Entities, Pages, Variables, Functions	Lexical Scoping, Module-Spanning Scoping, Function Overloading, Shadowing
Statix	14	Sorts, Constructors, Namespaces, Labels, Predicates, Variables	Lexical Scoping, Module-Spanning Scoping

Table 6.1: Test Coverage Summary

To evaluate the coverage of various name binding patterns in practice we tested it on five different languages. For each language, we wrote multiple unit tests in SPT to check if our implementation can deal with the language's name binding patterns. These tests are described in detail in Chapter 5. We summarize the coverage of our test cases in Table 6.1.

We adapted the approach of the name-fix [18] algorithm to detect capture. Specifically, we recalculate the resolution relation after the renaming transformation is done and compare it to the original one. If we detect a difference, the name binding structure of the program was inadvertently altered which may lead to a different runtime behavior. The test suites include test cases that would lead to name capture when applied in order to test detection of name capture.

The experiments confirm that renaming applies out of the box to a wide range of name binding patterns and correctly addresses name capture.

6.2 Tool Integration & Usability

We integrated our renaming algorithm into the Spoofax Language Workbench [28] which is deployed as a plugin for the Eclipse IDE. Every language engineer using Spoofax can add the Rename refactoring to their project in a few simple steps(see subsection 4.8.2). The main requirement for using our renaming feature is that the target language needs a Statix or NaBL2 specification that declares its name binding structure.

In a way, developing the refactoring was also a test for Spoofax's maturity and capability. Many of Spoofax's pieces had to work together properly to build such a complex editor service and we were able to fix a handful of bugs in the process. Starting with the parser which turns source code into ASTs and is also used to recognized what term a user has selected to be renamed. As we mentioned before, we heavily relied on the capabilities and interfaces of the Statix and NaBL2 solver.

Further, we used Stratego to perform the transformation and depended on its correct origin tracking in order to use the layout-preserving pretty printing. As we described in Section 5.2, we extended SPT so that we could unit test our renaming algorithm. Lastly, we used ESV to package our Rename refactoring as an editor service with multi-file capability.

The user interface of our solution is certainly a lot cruder than the refactoring engines of commercial products like IntelliJ IDEA. While our UI only offers the basic functions of entering a new name and displaying error messages, using our refactoring is still superior to manual renaming as it guarantees absence of name capture. It is also faster than renaming by hand, works on multi-file programs, and is layout-preserving.

6.3 Performance Analysis

One of the main advantages of automating refactorings is that it saves the developer time. To see if our implementation of the Rename refactoring provides this advantage, we conducted a basic performance analysis. For that, we renamed arbitrary program entities in programs of various sizes and measured the execution time of the refactoring. We tested the NaBL2 implementation on Statix programs and the Statix implementation on Chicago programs. The averages of the execution time measurements are summarized in Table 6.2. The measurements were performed on a laptop with 16 GB of RAM and an Intel Core i7-7700HQ CPU clocked at 2.8 GHz.

For programs smaller than 100 lines of code, the refactoring finishes without any notable lag. Renaming something in programs with a couple hundred lines of code also finishes within an acceptable time frame. Although, the lag becomes noticeable on the NaBL2 version. When dealing with programs greater than 1000 lines of code, the renaming takes well over a few seconds and becomes unacceptable in regards to usability. Based on these measurements,

we conclude that the performance results do not warrant application to large programs with the current implementation.

Program Size	Execution Time NaBL2	Execution Time Statix
< 100 LOC	0.19 s	0.19 s
< 1'000 LOC	2.48 s	0.25 s
< 10'000 LOC	104.80 s	21.26 s

Table 6.2: Execution Time Analysis

6.3.1 Execution Time Breakdown

To investigate why our refactoring performed so slowly on large programs, we recorded more detailed measurements of the execution time. Specifically, we measured the time each of the five steps of the algorithm took and put it into relation to the overall execution time of the algorithm. The results of these in-depth measurements are summarized in Table 6.3. Breaking down the execution time of the refactoring in that matter allowed us to identify the bottleneck of the implementations.

	Execution Time Share	
Step	**NaBL2**	**Statix**
1. Build Resolution Relation	18.7 %	2.1 %
2. Find Selected Occurrence	0.6 %	0.6 %
3. Union-Find	1.5 %	2.9 %
4. Change ASTs	0.7 %	0.5 %
5. Detect Capture	78.3 %	93.7 %

Table 6.3: Execution Time Breakdown per Algorithm Step

In the first step of the algorithm, we extract the resolution relation from the result of the static analysis. The implementation of this step is very dependent on the format of the analysis result and the API available to read the necessary name binding data from that result. That explains why there is a significant difference between the execution time share of NaBL2 version and the Statix version. It's important to mention that Spoofax pre-analyzes a project. Therefore, the initial static analysis on which the resolution relation is built is not included in these measurements.

In the NaBL2 version, we read all the references from the scope graph and performed name resolution for each of these occurrences to find the associated declarations. The time complexity of the name resolution algorithm is not trivial and executing it for every reference in a program comes at a significant cost, even if the solver uses a local cache for look-ups.

In Statix on the other hand, the declaration a reference resolves to is added to the reference AST node as a property. While this requires us to traverse all the ASTs in the project in order to build the resolution relation, we don't have to call the name resolution algorithm on every reference. Having the results of the name resolution cached in the form of AST properties clearly saves us a lot of time. Extending the NaBL2 solver so it would also somehow cache the resolution pairs would likely eliminate the aforementioned bottleneck.

In the second algorithm step, we check the users selection traversing the term he selected. Since this term only consists of a few nodes in most cases, this takes almost no time. In the next step, we find all name occurrences related to the users selection using the union-find [23] algorithm. While our implementation of union-find is rather simple, it could be optimized to run in $\mathcal{O}(m\alpha(n))$ [46, 45] for m operations and n equivalence classes. However,

since that step only takes up 1.5 % and 2.9 % respectively of the entire execution time that optimization would have little effect.

In step four we perform the actual renaming through transforming the AST. From the previous step, we know which name indexes we have to change. Since the name indexes contain the file path where they occur, it is easy to determine which files will be affected by the renaming and we can avoid a traversal of all ASTs in the project. Most renamings only affect a small number of files, regardless of the project's size. Therefore, this algorithm step scales rather well.

We check for capture in the last step of the algorithm. To do this, we rerun the static analysis and recompute the resolution relation. We've already established that calculating the resolution relation is slow in the NaBL2 version. Since we repeat that step as part of the capture detection step, we pass that bottleneck twice making it more detrimental to the performance of our refactoring.

From the execution time shares in Table 6.3, we see that both versions of the algorithm spend the majority of their time in the capture detection step. This is because we execute the static analysis again, which constitutes a bottleneck for both versions of our refactoring. Since the solvers is only minimally incremental, it performs an almost complete analysis of the program on every renaming.

We conclude that the poor scalability of our renaming implementation is mostly due to the performance of the NaBL2 and Statix solver. This is out of scope of this thesis. We see that the execution is mostly dominated by the cost of reanalysis. Improvement of the raw performance of the solver and incrementalizing the reanalysis step should scale the algorithm to larger projects. Alternatively, the algorithm we have presented in this thesis does work with any type checker that produces a resolution relation.

6.4 Limitations

One problem we were not able to solve is the consideration of informal name binding patterns when renaming. For example, when renaming a private field `foo` to `bar` in C++, our solution would not rename the associated setter to `setBar`. The reason being the name binding between the field and the setter is only implied by the convention of the setter pattern. While such conventions are helpful for developers, they do not matter at runtime and are therefore usually not encoded in a language's static semantics.

Since our renaming algorithm operates on the result of the static analysis, we have no way of adding informal name bindings to the resolution relation because those name bindings are not recognized by the name resolution algorithm. One key limitation of our solution is that it cannot deal with name binding patterns that are not explicitly part of a language's static name binding rules. In other words, our renaming algorithm is limited to the occurrences that appear in the resolution relation.

Unfortunately, this limitation can lead to a Rename refactoring not being behaviour preserving. For example, in Java a method can be called through the Reflection API, where the method's name is passed as a string literal. Since there is no explicit name binding between that literal and the method itself, our renaming would fail to change the string literal. This would result in the reflection call failing at runtime, effectively breaking the program through the refactoring.

Another limitation of our implementation is it does not change occurrences which appear in inline comments. Our solution works on the AST representation of a program, which does not contain nodes for comments as they are ignored by the parser. Although even if inline comments would be part of the AST, there would be no explicit name binding between the comment text and the associated program entity. Therefore the name binding would not be present in the resolution relation and the text in the comment would not change.

As we discussed in the previous section, the long execution time of our refactoring when dealing with big programs is another limitation. However, that limitation is a shortcoming of the implementation rather than the core renaming algorithm. Employing a faster implementation of the static analysis would remove this limitation.

Chapter 7

Related Work

7.1 Refactoring

The topic of refactoring was made popular by Martin Fowler's seminal book *Refactoring: Improving the Design of Existing Code* [20]. He proposes the concept of improving the design of existing code without altering a program's behaviour as an important tool for software developers to increase the quality and maintainability of software products. He describes how to perform 68 refactorings manually in a step-by-step fashion, including Rename Field, Rename Method and Rename Variable. While the examples given in the book are in Java, they could easily be applied in another language as well. Although, most of them are geared towards object-oriented languages.

While the book focuses on performing the refactoring transformations manually, in Chapter 14, Don Roberts and John Brant outline the benefits of automating refactorings. They state six criteria which a refactoring tool needs to meet in order to be successful, which are fulfilled by our implementation:

1. **Perform Transformations on a Parse Tree**
   The textual representation of a program is ill-fitted to execute an automated refactoring transformation upon. Formatting and white spaces obscure a programs structure and make it difficult to define a transformation in a concise way. Stratego allows us to define transformations on an abstract syntax tree which, is basically a simpler version of a parse tree. As we described in Section 3.2, we use Spoofax's generated JSGLR parser to turn the source code into an AST.

2. **Build a Program Database**
   Most refactorings require a way to find specific language entities in a program. To fulfill this, the tool needs to build a program database (or program model), that can be queried for the necessary information when executing the transformation.
   One kind of program database used in Spoofax is a scope graph, which represents name binding and typing information. The static analysis decorates the terms in the AST with annotations that associate elements in the scope graph. The Stratego language has an API for querying the program database through these annotations.

3. **Preserve Behavior**
   That an automated refactoring transformation does not break the program is arguably the most important criterion for a refactoring tool. We evaluated our solution on five different languages through unit tests using SPT (see Chapter 5).

4. **Save Time**
   Refactoring by hand is not only error-prone but also time consuming. Using a tool to refactor should save developers time and therefore money. The performance analysis of our solution can be found in Section 6.3.

5. **Undo a Renaming**
   As it is difficult to find a good name, it certainly can occur that a name needs to be changed back. The change in the source code performed by the refactoring is registered by the editor just like any other textual change. Therefore, the renaming can be reverted through Eclipse's standard Undo function.

6. **Integration with Tools**
   Because refactorings occur quiet often, they should be easy to perform and integrate seamlessly in the developer's workflow. DSLs developed in Spoofax can be deployed as plugins for the Eclipse IDE and programmers can make use of the platforms rich feature set when writing code in the implemented language. The integration into Eclipse's UI is described in Subsection 4.5.6.

Academically, the PhD thesis of Griswold[24] was one of the first major works on the topic of refactorings. In his thesis, he shows how "meaning-preserving transformations can restructure a program to improve maintainability" using various program transformations that would nowadays be considered refactorings, including one to rename a variable.

Similar to our solution, his transformations operate on the AST representation of a program. However, he uses a Program Dependence Graph (PDG) [34] as an abstract program model instead of a scope graph. To ensure the transformations are behavior-preserving, he postulates a set of allowed changes to the PDG which do not alter a program's semantics. While his model is language-agnostic, the prototype of his restructuring tool was implemented language-specifically with Scheme as the target language.

As a part of his work, he also conducted an experiment to compare manual and tool-aided restructurings, coming to the conclusion that "computer-aided restructuring is a potentially valuable approach to reducing the overall cost of software evolution."

The PhD thesis of Opdyke[41] was praised to be "the most substantial work on refactoring" by Fowler [20] and was a major influence for Fowler's famous book. His work focuses on object-oriented programming languages, which are touted to lower the complexity of programs by facilitating the reuse of software components. In that regard, he finds that "object-oriented software often needs to be restructured before it can be reused." The large number of refactorings that exist for object-oriented languages arguably confirms his statement.

Opdyke defines 26 low-level restructurings, such as Move Method or Rename Variable, from which he builds high-level refactorings, such as Extract Superclass. To ensure behaviour-preservation, each low-level transformation is associated with a set of 35 possible preconditions, which cause the whole refactoring to be aborted and rolled back in case one of them evaluates to false.

In his PhD thesis, Roberts[43] advanced the research of Opdyke, focusing on the Smalltalk programming language. He extends Opdyke's model with post-conditions that are expected to hold after the refactoring transformation is complete. The way we implemented capture detection in our solution is through such a post-condition. His work served as the basis for the development of The Refactoring Browser[1], which claims to be one of the first commercial refactoring engines.

Building on Opdyke and Roberts work, Schaefer developed the refactorings Rename, Inline Temp and Extract Method for Java [44]. His work contains a detailed description of

[1] https://refactory.com/refactoring-browser/

Java's name binding structures and the challenges they pose for a correct renaming. His research covers complex name binding patterns such as nested types, method overloading, and qualified names. He also extensively investigated the problem of name capture and provides a solution that can fix capture in certain cases through qualifying names.

Schaefer implemented his refactoring engine on top of the JastAddJ compiler [17] that extends Java with rewritable reference attribute grammars [16]. He implemented a name lookup algorithm using an attribute grammar and built his Rename refactoring on top of that algorithm, similar to how our solution relies on name resolution information from the Scope Graph.

Kniesel and Koch[31] further refined Opdyek's approach of composing complex refactoring from small and basic program transformations. They developed "a formal model for automatic, program-independent composition of conditional program transformations." This is language-parametric in general, although most of their basic *conditional transformations* are limited to object-oriented languages. They implemented the refactoring browser ConTraCT based on their model with Java being the target language. Their work served as an inspiration to break down our renaming algorithm into small steps. This made it easier to separate the generic steps from the steps specific to the name binding language in use.

A lot of the works discussed in this section present theoretical models which could be considered language-parametric, most of them being focused on object-oriented languages. However, none of them evaluated their models with multiple programming languages and none of them have the plug-and-play ability of the Spoofax implementation of our solution.

7.2 Name Binding

There exists a substantial body of research considering name binding in formal languages and multiple language-parametric solutions, such as symbol tables, attribute grammars [32] or visibility predicates [40]. We opted to build our renaming transformation on top of the solution provided by Spoofax using declarative name binding and scope graphs.

Konat et al.[33] started the work on this by developing "a declarative metalanguage for the specification of name bindings in terms of namespaces, definition sites, use sites, and scopes" that comes with a language-parametric name resolution algorithm. Neron et al.[39] further developed that idea, introducing the concept of the scope graph and its accompanying formal name resolution calculus. They outlined how a renaming could be implemented on top of their framework. This served as a foundation for this master thesis.

The scope graph formalism was extended to "a full framework for static semantic analysis" by Antwerpen et al.[1], through essentially "uniting a type checker with our existing name resolution machinery." They present a constraint-based approach to name and type resolution and their work resulted in the meta-language NaBL2. One of our implementations uses the NaBL2 Stratego API to gather the necessary name binding information.

Antwerpen et al.[2] carried on the work on the scope graph framework to allow it to deal with more complex type systems. They find that "viewing scopes as types enables us to model the internal structure of types in a range of non-simple type systems". They also presented Statix as a successor to NaBL2 as part of their work, for which we built an implementation of our Rename refactoring.

7.3 Capture Detection

The Name-Fix algorithm of Erdweg et al.[18] can repair name capture that was caused by an arbitrary program transformation as it "renames variable names to differentiate the captured variables from the capturing variables, while preserving intended variable references". Our approach to detect capture was heavily influenced by their work.

Even though their solution focuses on code generation transformations rather than refactorings, we were able to adopt their capture detection approach as a key piece of our renaming algorithm. Name-fix builds a name graph of the program as an abstract representation of its name binding structure and detects capture by comparing the graphs before and after the transformation is done.

7.4 Other Language Workbenches

Racket [19] is both a meta-programming language and framework to build domain-specific languages. It has a powerful macro system and has been used to implement renaming for local variables. Racket's Resyntax library provides macros to build more complex refactoring. However, it is still in an experimental development state.

JetBrains MPS (Meta Programming System) [50] is a commercial tool for developing domain-specific languages. While MPS does provide an automated Rename refactoring, it is less sound and complete than our solution. Complex name binding patterns, such as method overloading or qualified names, cannot be expressed in MPS. Thus, are also not covered by its renaming algorithm. Renaming in MPS also doesn't detect name capture and is therefore not behavior-preserving.

Chapter 8

Conclusion

Refactorings are a valuable tool for software developers to improve code quality. Developing automated refactorings as an editor service is notoriously complex and time-consuming. Most existing implementations only target one specific programming language and therefore need to be reimplemented when developing a new language. We built an automatic Rename refactoring that works on any programming language, achieving the goal of this thesis.

The main challenge of developing a renaming transformation that works on an arbitrary programming language is that each language has its own unique set of name binding rules. We delegate the complexity of dealing with these rules to a language-parametric name resolution algorithm [33]. This hides the details of a specific language's name binding specification from the renaming transformation, achieving a separation of concern between name resolution and refactoring.

In the first step of our renaming algorithm, we build a language-independent representation of a program's name binding structure. For this, we assign each name occurrence a name index that uniquely identifies it. We then use the name resolution algorithm to associate each reference occurrence with its respective declaration occurrence, representing each name binding as a pair of name indexes. Collecting all resolution pairs into a resolution relation [39] creates an abstraction that contains all the name binding information of a program.

Next, we calculate the equivalence classes of the resolution relation with the union-find algorithm [23], where each equivalence class contains all the name occurrences that represent one specific program entity. In order to rename a program entity, we just need to find its equivalence class and then, change all the name occurrences it contains.

To ensure the transformation is behavior-preserving, we adapted the name capture detection approach of the name-fix algorithm of Erdweg et al.[18]. In order to detect capture, we recalculate the resolution relation after the transformation and compare it to the original one, since name capture alters the name binding structure of a program.

We implemented our Rename refactoring using the Spoofax Language Workbench [28]. The algorithm was implemented as rewrite rules in the Stratego [9] term transformation meta-language and equipped with a user interface using the Editor Service Language (ESV) of Spoofax. Spoofax offers the two meta-languages NaBL2 [1] and Statix [2] for declaring a language's static semantics. We developed two versions of the refactoring; one using the NaBL2 solver and the second using the Statix solver for name binding and resolution.

To test our refactoring, we wrote a total of 85 unit tests in the testing meta-language SPT [27]. We were able to show that our algorithm behaves correctly for a wide array of name binding patterns on the basis of our tests with the languages Tiger [3], MiniJava [4], Chicago, Statix, and WebDSL [49]. To test our renaming algorithm with SPT, we also extended its runtime with the ability to deal with more complex Stratego strategies.

Our solution has two noteworthy limitations. First, the refactoring only changes names that are explicitly bound together through the target language's static semantics. For example, renaming a field in Java would not change the name of the accompanying getter or any occurrences of the field's name in a JavaDoc comment. Second, our implementation scales poorly when dealing with programs of several thousands lines of code. Executing a renaming on program of that size takes on average 21 seconds using the Statix solver and 105 seconds using the NaBL2 solver. This long execution time restricts the usability of the refactoring. We identified the solvers as the bottleneck of the algorithm and any speed-up of the solver should improve the performance of the renaming proportionally.

Finally, we integrated our automated refactoring into the Spoofax Language Workbench, providing a renaming editor service for every language out-of-the-box.

8.1 Future Work

To close out this thesis, we present a few ideas for improvements and extension to our Rename refactoring, on which we hope to work in the future.

8.1.1 Qualified Names

Some implementations of the Rename refactoring automatically repair the name binding structure of a program in the event of name capture through injecting qualified names. For example, when the renaming of a method argument would capture a reference to a field, the name binding structure could be restored by qualifying the field reference with `this`. The implementation of the Java Rename refactoring by Schaefer[44] has this capability. Our renaming solution simply aborts upon detecting name capture.

While not all programming languages support qualified names, it is a name binding pattern that appears in many object-oriented languages. It would certainly be interesting to try and extend our language-parametric solution with an option to remedy capture through qualified names, assuming the target language supports that name binding feature.

In case a reference resolves to a different declaration after renaming, it is possible to find other declarations which are reachable from that reference within the scope graph. If the original declaration which the captured reference was bound to is still reachable, we would then need to modify the AST to qualify said reference. We would need to implement a function that would map the correct resolution path in the scope graph to a transformation that inserts the name qualifier.

8.1.2 Binding-Specific Renaming

Our solutions requires a complete traversal of the AST in order to perform the renaming. However, there is evidence that this may not always be required. For example, when renaming an entity which is bound through lexical scoping, it is clear that all references are going to be located on a lower level of the same AST branch as the declaration. Therefore, only that sub-tree would have to be traversed instead of the entire AST, which we would expect to result in a performance gain.

Implementing such an enhancement would require to survey common name binding patterns and categorizing them with regards to their locality within the AST. Using the scope graph framework, such a categorization could probably be done by inspecting the steps of the resolution paths and their length. The renaming transformation would then need to adjust its traversal strategy based on the name binding pattern of the selected language entity. This in turn would require the algorithm to analyze the name binding specification to find such patterns whereas, our current solution is agnostic to that specification.

It is very difficult to estimate how much such an enhancement would actually limit the AST traversal, as this depends on a number of factors. It might vary greatly between different languages, depending on which patterns they apply to what language entities. The size and complexity of the target program is also expected to affect the extent of speed-up. Whether the performance gains would warrant the complexity of such a solution is future work.

8.1.3 More Language-Parametric Refactorings

Besides Rename, there exist dozens of other refactorings that would be interesting to implement language-parametrically. Move Method, Inline Method, and Extract Method are the next most commonly used refactorings [47]. It would be convenient to have those available out of the box as well when developing a new language. As we describe in Subsection 7.1, there exist a number of theoretical approaches to implement more complex language-parametric refactorings, as well as many language-specific implementations.

Our Rename solution could most certainly serve as a starting point to implement more complex refactorings in Spoofax. Most refactorings interface with the name binding structure of a program in some way and some of our Stratego code could surely be reused in other refactorings. Name capture is not just a concern for the Rename refactoring. Therefore, our capture detection implementation might serve as a foundation to spot this problem in more complex transformations. Our SPT extension will definitely be useful when testing them.

Bibliography

[1] Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. "A constraint language for static semantic analysis based on scope graphs". In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, pp. 49–60. DOI: 10.1145/2847538.2847543. URL: https://doi.org/10.1145/2847538.2847543.

[2] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. "Scopes as types". In: *PACMPL* 2.OOPSLA (2018), 114:1–114:30. DOI: 10.1145/3276484. URL: https://doi.org/10.1145/3276484.

[3] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. ISBN: 0-521-58274-1.

[4] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java, 2nd edition*. Cambridge University Press, 2002. ISBN: 0-521-82060-X.

[5] Franz Baader, Wayne Snyder, Paliath Narendran, Manfred Schmidt-Schauß, and Klaus U. Schulz. "Unification Theory". In: *Handbook of Automated Reasoning (in 2 volumes)*. Ed. by John Alan Robinson and Andrei Voronkov. Elsevier and MIT Press, 2001, pp. 445–532. DOI: 10.1016/b978-044450813-3/50010-2. URL: https://doi.org/10.1016/b978-044450813-3/50010-2.

[6] Kent L. Beck. *Test-driven Development - by example*. The Addison-Wesley signature series. Addison-Wesley, 2003. ISBN: 978-0-321-14653-3.

[7] Mark van den Brand, H. A. de Jong, Paul Klint, and Pieter A. Olivier. "Efficient annotated terms". In: *Softw. Pract. Exp.* 30.3 (2000), pp. 259–291. DOI: 10.1002/(SICI)1097-024X(200003)30:3\<259::AID-SPE298\>3.0.CO;2-Y. URL: https://doi.org/10.1002/(SICI)1097-024X(200003)30:3%5C%3C259::AID-SPE298%5C%3E3.0.CO;2-Y.

[8] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. "Program Transformation with Scoped Dynamic Rewrite Rules". In: *Fundam. Informaticae* 69.1-2 (2006), pp. 123–178. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi69-1-2-06.

[9] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. "Stratego/XT 0.17. A language and toolset for program transformation". In: *Science of Computer Programming* 72.1-2 (2008), pp. 52–70. DOI: 10.1016/j.scico.2007.11.003. URL: http://dx.doi.org/10.1016/j.scico.2007.11.003.

[10] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. "Stratego/XT 0.17. A language and toolset for program transformation". In: *Science of Computer Programming* 72.1-2 (2008), pp. 52–70. DOI: 10.1016/j.scico.2007.11.003. URL: http://dx.doi.org/10.1016/j.scico.2007.11.003.

[11] TIOBE Software BV. *TIOBE Index for June 2020*. https://www.tiobe.com/tiobe-index/. June 2020.

[12] Pierre Carbonnelle. *PYPL PopularitY of Programming Language*. https://pypl.github.io/PYPL.html. June 2020.

[13] Alonzo Church. "A Set of Postulates For the Foundation of Logic". In: *Annals of Mathematics* 34.4 (1933), pp. 839–864. ISSN: 0003486X. URL: http://www.jstor.org/stable/1968702.

[14] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. "Automated testing of refactoring engines". In: *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. 2007, pp. 185–194. DOI: 10.1145/1287624.1287651. URL: https://doi.org/10.1145/1287624.1287651.

[15] Programming Language Research Group TU Delft. *The Spoofax Language Workbench*. https://www.metaborg.org/en/latest/index.html. Dec. 2020.

[16] Torbjörn Ekman and Görel Hedin. "Rewritable Reference Attributed Grammars". In: *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*. Ed. by Martin Odersky. Vol. 3086. Lecture Notes in Computer Science. Springer, 2004, pp. 144–169. DOI: 10.1007/978-3-540-24851-4_7. URL: https://doi.org/10.1007/978-3-540-24851-4%5C_7.

[17] Torbjörn Ekman and Görel Hedin. "The jastadd extensible java compiler". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. ACM, 2007, pp. 1–18. DOI: 10.1145/1297027.1297029. URL: https://doi.org/10.1145/1297027.1297029.

[18] Sebastian Erdweg, Tijs van der Storm, and Yi Dai. "Capture-Avoiding and Hygienic Program Transformations". In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. Ed. by Richard E. Jones. Vol. 8586. Lecture Notes in Computer Science. Springer, 2014, pp. 489–514. DOI: 10.1007/978-3-662-44202-9_20. URL: https://doi.org/10.1007/978-3-662-44202-9%5C_20.

[19] Matthew Flatt. "Creating languages in Racket". In: *Commun. ACM* 55.1 (2012), pp. 48–56. DOI: 10.1145/2063176.2063195. URL: https://doi.org/10.1145/2063176.2063195.

[20] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999. ISBN: 978-0-201-48567-7. URL: http://martinfowler.com/books/refactoring.html.

[21] Martin Fowler. *TwoHardThings*. https://martinfowler.com/bliki/TwoHardThings.html. July 2009.

[22] Michael L. Fredman and Michael E. Saks. "The Cell Probe Complexity of Dynamic Data Structures". In: *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*. Ed. by David S. Johnson. ACM, 1989, pp. 345–354. DOI: 10.1145/73007.73040. URL: https://doi.org/10.1145/73007.73040.

[23] Bernard A. Galler and Michael J. Fischer. "An improved equivalence algorithm". In: *Commun. ACM* 7.5 (1964), pp. 301–303. DOI: 10.1145/364099.364331. URL: https://doi.org/10.1145/364099.364331.

[24] William G. Griswold. "Program Restructuring as an Aid to Software Maintenance". UMI Order No. GAX92-03258. PhD thesis. USA, 1992.

[25] C. A. R. Hoare. "Algorithm 64: Quicksort". In: *Commun. ACM* 4.7 (July 1961), p. 321. ISSN: 0001-0782. DOI: 10.1145/366622.366644. URL: https://doi.org/10.1145/366622.366644.

[26] Maartje de Jonge and Eelco Visser. "An Algorithm for Layout Preservation in Refactoring Transformations". In: *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers*. Ed. by Anthony M. Sloane and Uwe Aßmann. Vol. 6940. Lecture Notes in Computer Science. Springer, 2011, pp. 40–59. DOI: 10.1007/978-3-642-28830-2_3. URL: https://doi.org/10.1007/978-3-642-28830-2%5C_3.

[27] Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. "Integrated language definition testing: enabling test-driven language development". In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. 2011, pp. 139–154. DOI: 10.1145/2048066.2048080. URL: https://doi.org/10.1145/2048066.2048080.

[28] Lennart C. L. Kats and Eelco Visser. "The Spoofax language workbench: rules for declarative specification of languages and IDEs". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. 2010, pp. 444–463. DOI: 10.1145/1869459.1869497. URL: https://doi.org/10.1145/1869459.1869497.

[29] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. "Pure and declarative syntax definition: paradise lost and regained". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, 2010, pp. 918–932. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869535. URL: http://doi.acm.org/10.1145/1869459.1869535.

[30] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. "Pure and declarative syntax definition: paradise lost and regained". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. 2010, pp. 918–932. DOI: 10.1145/1869459.1869535. URL: https://doi.org/10.1145/1869459.1869535.

[31] Günter Kniesel and Helge Koch. "Static composition of refactorings". In: *Sci. Comput. Program.* 52 (2004), pp. 9–51. DOI: 10.1016/j.scico.2004.03.002. URL: https://doi.org/10.1016/j.scico.2004.03.002.

[32] Donald E. Knuth. "Semantics of Context-Free Languages". In: *Math. Syst. Theory* 2.2 (1968), pp. 127–145. DOI: 10.1007/BF01692511. URL: https://doi.org/10.1007/BF01692511.

[33] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. "Declarative Name Binding and Scope Rules". In: *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*. Ed. by Krzysztof Czarnecki and Görel Hedin. Vol. 7745. Lecture Notes in Computer Science. Springer, 2012, pp. 311–331. DOI: 10.1007/978-3-642-36089-3_18. URL: https://doi.org/10.1007/978-3-642-36089-3%5C_18.

[34] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. "Dependence Graphs and Compiler Optimizations". In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '81. Williamsburg, Virginia: Association for Computing Machinery, 1981, pp. 207–218. ISBN: 089791029X. DOI: 10.1145/567532.567555. URL: https://doi.org/10.1145/567532.567555.

[35] Steve McConnell. *Code complete - a practical handbook of software construction, 2nd Edition*. Microsoft Press, 2004. ISBN: 9780735619678. URL: http://www.worldcat.org/oclc/249645389.

[36] Daniel D Merrill. "Augustus De Morgan's Boolean Algebra". In: *History and Philosophy of Logic* 26.2 (2005), pp. 75–91. DOI: 10.1080/01445340412331323124. eprint: https://doi.org/10.1080/01445340412331323124. URL: https://doi.org/10.1080/01445340412331323124.

[37] Melina Mongiovi. "Scaling testing of refactoring engines". In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. 2016, pp. 674–676. DOI: 10.1145/2889160.2891038. URL: https://doi.org/10.1145/2889160.2891038.

[38] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, and Leopoldo Teixeira. "Detecting Overly Strong Preconditions in Refactoring Engines". In: *IEEE Trans. Software Eng.* 44.5 (2018), pp. 429–452. DOI: 10.1109/TSE.2017.2693982. URL: https://doi.org/10.1109/TSE.2017.2693982.

[39] Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. "A Theory of Name Resolution". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, pp. 205–231. DOI: 10.1007/978-3-662-46669-8_9. URL: https://doi.org/10.1007/978-3-662-46669-8%5C_9.

[40] Martin Odersky. "Defining Context-Dependent Syntax Without Using Contexts". In: *ACM Trans. Program. Lang. Syst.* 15.3 (1993), pp. 535–562. DOI: 10.1145/169683.174159. URL: https://doi.org/10.1145/169683.174159.

[41] William F. Opdyke. "Refactoring Object-Oriented Frameworks". UMI Order No. GAX93-05645. PhD thesis. USA, 1992.

[42] Daniël Pelsmaeker. *Portable Editor Services*. 2017.

[43] Donald B Roberts. *Practical Analysis for Refactoring*. Tech. rep. USA, 1999.

[44] Max Schaefer. "Specification, implementation and verification of refactorings". PhD thesis. University of Oxford, UK, 2010. URL: http://ora.ox.ac.uk/objects/uuid:1a027679-1e2b-4fb5-a6ff-3270f15154a1.

[45] Robert Endre Tarjan. "A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets". In: *J. Comput. Syst. Sci.* 18.2 (1979), pp. 110–127. DOI: 10.1016/0022-0000(79)90042-4. URL: https://doi.org/10.1016/0022-0000(79)90042-4.

[46] Robert Endre Tarjan and Jan van Leeuwen. "Worst-case Analysis of Set Union Algorithms". In: *J. ACM* 31.2 (1984), pp. 245–281. DOI: 10.1145/62.2160. URL: https://doi.org/10.1145/62.2160.

[47] Applied Software Engineering Research Group DCC UFMG. *What are the most common refactoring operations performed by GitHub developers?* https://medium.com/@aserg.ufmg/what-are-the-most-common-refactorings-performed-by-github-developers-896b0db96d9d. Jan. 2017.

[48] Eelco Visser. *Scannerless generalized-LR parsing*. Universiteit van Amsterdam. Programming Research Group, 1997.

[49] Eelco Visser. "WebDSL: A Case Study in Domain-Specific Language Engineering". In: *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*. Ed. by Ralf Lämmel, Joost Visser, and João Saraiva. Vol. 5235. Lecture Notes in Computer Science. Springer, 2007, pp. 291–373. DOI: 10.1007/978-3-540-88643-3_7. URL: https://doi.org/10.1007/978-3-540-88643-3%5C_7.

[50] Markus Voelter. “Language and IDE Modularization and Composition with MPS”. In: *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 7680. Lecture Notes in Computer Science. Springer, 2011, pp. 383–430. DOI: 10.1007/978-3-642-35992-7_11. URL: https://doi.org/10.1007/978-3-642-35992-7%5C_11.

Acronyms

AST Abstract Syntax Tree

ATerm Annotated Term

DSL Domain-specific Language

GUI Graphical User Interface

IDE Integrated Development Environment

JVM Java Virtual Machine

LOC Lines of Code

SDK Software Development Kit

SDF Syntax Definition Formalism

Appendix A

Source Code

This Appendix contains references to the source code we created or modified during the development of our `Rename` refactoring. All of it is open source and publicly hosted on GitHub.

A.1 Renaming Integration

How we integrated our `Rename` refactoring into Spoofax is described in Section 4.8. The changes to the source code are represented through the following pull requests:

1. `https://github.com/metaborg/spoofax/pull/76`
2. `https://github.com/metaborg/spoofax-eclipse/pull/20`
3. `https://github.com/metaborg/nabl/pull/36`
4. `https://github.com/metaborg/jsglr/pull/94`
5. `https://github.com/metaborg/nabl/pull/37`
6. `https://github.com/metaborg/spoofax/pull/79`
7. `https://github.com/metaborg/nabl/pull/42`
8. `https://github.com/metaborg/nabl/pull/43`
9. `https://github.com/metaborg/nabl/pull/45`
10. `https://github.com/metaborg/spoofax/pull/79`
11. `https://github.com/metaborg/spoofax/pull/79`
12. `https://github.com/metaborg/nabl/pull/46`
13. `https://github.com/metaborg/nabl/pull/47`
14. `https://github.com/metaborg/nabl/pull/48`
15. `https://github.com/metaborg/nabl/pull/49`
16. `https://github.com/webdsl/webdsl-statix/pull/2`
17. `https://github.com/metaborg/documentation/pull/50`
18. `https://github.com/MetaBorgCube/statix-sandbox/pull/3`
19. `https://github.com/MetaBorgCube/metaborg-tiger/pull/12`
20. `https://github.com/metaborg/spoofax/pull/83`

A.2 SPT Extension

Our implementation of the SPT extension is described in Section 5.3. The specific code changes were merged into the master branch through the following pull requests:

1. `https://github.com/metaborg/spt/pull/34`
2. `https://github.com/metaborg/spt/pull/35`
3. `https://github.com/metaborg/spoofax/pull/74`
4. `https://github.com/metaborg/mb-exec/pull/10`

A.3 SPT Unit Tests

In Section 5.4 we only described some of the unit tests we developed to evaluate our refactoring. The links to the test projects containing all the SPT tests we wrote are shown in Table A.1. The MiniJava repository is private, as it is used to grade assignments from the compiler construction course. Therefore the tests for MiniJava are provided on request.

Language	**Link to Test Folder**
Tiger	`https://github.com/metaborg/metaborg-tiger/tree/renaming/org.metaborg.lang.tiger.refactoring.test/test`
Chicago	`https://github.com/metaborg/statix-sandbox/tree/renaming-integrated/chicago/chicago.test/renaming`
WebDSL	`https://github.com/webdsl/webdsl-statix/tree/renaming/webdslstatix.test/renaming`
Statix	`https://github.com/metaborg/nabl/tree/master/statix.test/renaming`

Table A.1: Links to Unit Test Folders