Property-Based ASTs

Enabling Language Parametricity in Refactoring Tools

Hendy Liang

Property-Based ASTs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Hendy Liang
born in Roosendaal, the Netherlands

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

Property-Based ASTs

Author: Hendy Liang
Student id: 5027268

Abstract

Refactoring legacy systems is essential to maintain and modernize aging codebases, but traditional refactoring tools are often limited by language specificity and lack extensibility. This thesis introduces property-based Abstract Syntax Trees (ASTs), a flexible intermediate representation aimed at enhancing the language-parametric capabilities of refactoring tools. By leveraging Tree-Sitter, a parser generator that creates parsers that produce generic, property-based ASTs, this research adapts Renaissance, an existing industrial refactoring tool, to support multi-language extensibility with minimal additional effort. The adapted tool demonstrates equivalent functionality across C++, Java, and Python, maintaining features such as pattern matching, code rewriting, and placeholder handling. Experiments were performed, including experiments with exercises on an open-source repository, in order to highlight the practical benefits, extensibility, and limitations of this approach. This adaptation aims to showcase the feasibility of using property-based ASTs in enabling language-parametric tooling. This work lays the foundation for more centralized, cost-effective, and scalable tool development for industrial software refactoring.

Thesis Committee:

Chair:	Dr. J.G.H. Cockx, Faculty EEMCS, TU Delft
Committee Member:	Dr. S. Dumančić, Faculty EEMCS, TU Delft
Committee Member:	Dr. C.B. Poulsen, Faculty EEMCS, TU Delft
University Supervisor:	MSc. L. Miljak, Faculty EEMCS, TU Delft
External Supervisor:	Dr. R. Corvino, Senior Research Fellow, TNO-ESI

Preface

This Master's Thesis marks the end of my academic journey at the Delft University of Technology. These past five and a half years have been inspiring and educational. I was able to make friends, learn and study about my interests and grow as a person. I was anxious at first about choosing a career path straight out of high school, but I think I chose the correct path and the courses and concepts I have learned about have been interesting to me and I really enjoyed becoming more knowledgeable in them. Although challenging at times, somehow I have made it to the end with decent grades and an interesting Master's Thesis to close it off.

I want to thank my supervisors Casper, Rosilde and Luka, who have helped me through this thesis and provided me advice and direction at various points during the process. I feel like I learned a lot from them and will retain this critical and analytical approach for the rest of my career.

I also want to thank my parents for supporting me throughout this journey and supporting me and believing in my decisions.

Finally, I want to thank all my friends for making my time as a student an unique and fun experience.

Hendy Liang
Delft, the Netherlands
February 19, 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High-tech industries depend on complex software-intensive systems that must continuously adapt to rapidly changing market demands. However, this ongoing evolution often results in the build-up of technical debt, complicating maintenance efforts. These systems are often long-lived and outdated when compared to modern standards. We call these older systems, legacy systems. This issue with legacy code underscores the need for effective strategies to maintain the adaptability and functionality of these systems.

Legacy systems, maintained for many years, are often too costly to rewrite due to the departure of original developers with specialized knowledge. Making significant changes is risky and expensive, leading to updates through workarounds or wrappers. This increases the codebase's size and complexity, raising technical debt and the cost of the maintenance process.

Understanding the system is already the biggest time-sink of the maintenance process for developers (Minelli, Mocci, and Lanza 2015). It is estimated that 80% of the costs in a software life cycle are spent on maintenance (Telea and Voinea 2011). The US government spent 75% of its 2020 IT budget on legacy systems, of which two-thirds is estimated to be "waste" (Krasner 2021). Waste is defined as a lean term that means all non-value-added activities in an IT organization.

Technical debt (De Groot et al. 2012) consists of the cost of repairing the system to reach an ideal code quality level (debt) and the additional maintenance costs of maintaining a system that is not on an ideal code quality level (interest). This debt makes future development riskier and more expensive, creating a cycle of increasing technical debt. To break this cycle, long-term changes to the legacy system are necessary.

Refactoring, or restructuring code without changing its behaviour, can reduce technical debt. Legacy code often has lower quality due to outdated practices and technologies. The goal of refactoring is to improve the code structure while maintaining its behaviour. However, manual refactoring is repetitive and error-prone, highlighting the need for better methods to improve legacy code quality.

The repetitive nature of refactoring operations makes it an ideal venue for automation. Various refactoring tools (Laar and A. Mooij 2022; Moderne 2024; Schuts et al. 2022) exist that allow for large-scale automatic refactoring, but the issue is that they are usually tailored specifically to one language or a selection of a few.

For example, Renaissance (Laar and A. Mooij 2022) is a tool developed by TNO for large-scale code refactoring and analysis. It is a language-specific tool that currently works only in a selection of a few languages. It has been used at the industrial level in companies such as Philips (A. J. Mooij, Eggen, et al. 2015) and Nexperia ITEC (Laar and A. Mooij 2022) to refactor their legacy systems. It can be used to search for repeating code fragments and rewrite them. It does this by allowing users to specify a concrete syntax pattern, which is a pattern that looks like a piece of code. This pattern can then contain placeholders to capture

variables, values, or statements to use later in the rewriting stage. Using a concrete syntax pattern is more intuitive for software engineers because it looks similar to code that would normally be written. Concrete syntax patterns will be explained in more detail in Section 3.1.

Another example is CLAIR (Schuts et al. 2022), which has been used to perform semi-automated migration of legacy test code written in C and C++. The system to be refactored at hand used two different testing frameworks, a newer one being used for modern components and an older one for legacy components. The goal was to migrate from the older testing framework to the newer one. Without the automation of the process, this migration would not have been conducted, and this is due to several risks such as risk of introducing errors, risk of unexpected rework and a risk of loss of productivity. The language engineers were able to use CLAIR and fit it to their problem at hand, perform automatic transformations. This shows a need for this type of language tool that can perform automated refactoring.

At the core of these tools is a parser that converts source code into an Abstract Syntax Tree (AST), a tree representation where each node corresponds to a code construct like if-statements or for-statements. There are two main types of parsers: black-box parsers, which are handwritten and language-specific, and generated parsers, which are created from a parser generator that uses a grammar to generate a parser. Black-box parsers handle industrial-level legacy code better but are more irregular, while generated parsers produce structured, generic ASTs but struggle with complex legacy code.

The refactoring tools operate on the AST provided by the parser, which can vary in form and information. Two types of ASTs are discussed: rich ASTs, which embed syntactic information in the nodes of the tree itself, and property-based ASTs, which have generic nodes with properties containing syntactic information. Both types store similar information differently, affecting how automatic tools are created based on them. A detailed explanation of these two types and the differences and advantages of both and how property-based ASTs can help make refactoring tools language-parametric is provided in Chapter 2.

The goal of this research is to make language tools more language parametric. Language parametric (or agnostic) refers to being able to perform the same operations regardless of language. If the tool requires zero input and additional effort from the user to work in different languages, we consider it to be language agnostic. If some effort has to be put in (parameters have to be provided), we consider the tool to be language parametric. Ideally, we want the tools we use to be language agnostic, but realistically, language parametric is a more achievable approach.

Enabling refactoring tools to be language parametric is important for several reasons. Having different tools for different languages results in requiring more manpower to maintain all these tools. New features also need to be implemented in multiple tools, even if they share similar logic across languages. Software engineers that want to use the tools also have to learn and understand using a new tool every time if they are refactoring different languages. Having "one tool to rule them all" would result in more centralized development, easier maintenance, and tool knowledge that carries over between languages for the engineers using the tool. It would allow for a feature to be developed and be rolled out to multiple languages simultaneously instead of having to redevelop that feature. Having one tool for all the languages imaginable is too idealistic, but concentrating tool development efforts still remains a valuable cost-saving activity because of the aforementioned reasons. A tool that would work on Java and C++, would allow the engineer to refactor one project in Java and another in C++ without having the extra burden of learning how a tool specific to C++ works. On the tool development side, hypothetically, if a new language feature is released in Java that already exists in some form in C++ and already has tool support, developing that feature for Java would require less effort.

What we aim to demonstrate is that using a property-based AST as the basis for a refactoring tool will make it easier for that tool to be extended with new languages. The transfor-

mation that this tool provides is not guaranteed to be correct by construction. The burden of checking for compilability and correctness should be delegated to the compiler/type-checker and tests. These should follow the refactoring in the development flow. A compiler will already exist for a working language. Additionally, for large industrial projects, tests already exist to test the code, so they can be reused after refactoring to check for correct behaviour. If these tests do not exist, the project could be broken anyhow by any change, manual or automated. We demonstrate the feasibility and effort required of using a property-based AST by adapting Renaissance with Tree-Sitter.

Tree-Sitter (Brunsfeld 2024) is a parser generator that can be used to create parsers for a programming language given a grammar for that language. There are many grammars available for popular programming languages. If there is a grammar for the language we wish to use, the effort of using Tree-Sitter is almost negligible. The AST returned by a parser generated from Tree-Sitter can be classified as property-based ASTs.

We adapt Renaissance to use Tree-Sitter as its parser. We then perform an evaluation on the adapted Renaissance, comparing it to the original for validation of results. We measure the effort it takes to use a new language in the adapted Renaissance in terms of lines of code. We also measure the effort taken for the adaptation, factoring in time and effort to understand the Renaissance codebase.

The results of this evaluation show us that a property-based AST does in fact allow for increased extensibility of the tool and it mostly performs as expected. Some incorrect behaviour is observed, but this is not due to the approach of using a property-based AST, but rather due to implementation issues within Tree-Sitter.

1.1 Research Questions

The main goal of this thesis is to explore the possibilities of making refactoring tools language parametric. This allows for easier extension with newer languages as well as centralized development for language refactoring support for multiple languages. We have narrowed it down to focus on the intermediate representation of the target code on which such tools would work. In this case, the AST is the intermediate representation. For this, we have the following research questions:

1. What AST is best to simplify the addition of new languages in language tools or workbenches? We want to maintain the same functional behaviour whilst we want it to be easy to extend the tool with a new language. What properties do we need to extract from the original AST for the tool components to remain functional?
    - What is the right format for the intermediate AST representation to ensure tool component interoperability without sacrificing performance?
    - Does the needed structure and format of the AST intermediate representation differ for generated parsers and handwritten parsers?
    - What information is lost during this process of mapping information from one representation to another?
2. How do we adapt language tools and workbenches to make them more language parametric? Specifically, how do we make it effective and efficient to provide support for new languages in such tools and workbenches?

1.2 Contributions

To answer these research questions, we make the following contributions:

- We introduce a notion of an intermediate representation (property-based AST) for the refactoring tool to use. We compare this to existing representations and show advantages and disadvantages.
- We investigate how Renaissance (and similar tools) have to be adjusted to make them more language parametric.
  - We provide a proof of concept adapting Renaissance to Tree-Sitter.
  - This proof of concept is used to give an estimate of the amount of effort it took to adapt the existing tool.
  - This also shows possible features that can be achieved with this property-based AST. Features include using concrete syntax patterns to find and replace blocks of code, and rewriting code based on code offsets which helps preserve layout information.
  - We analyse performance of the adapted Renaissance in comparison to the original Renaissance in terms of speed, memory usage and correctness.
  - We also give a rough estimate of how much effort was required for the adaptation in terms of lines of code changed and time taken.

1.3 Overview

The rest of this thesis is split up as follows. Firstly, we explain the rich ASTs and property-based ASTs in more detail and highlight differences and advantages of the property-based AST in Chapter 2. We then dive into our proof of concept that we have built upon these ideas in Chapter 3. Afterwards, we provide an outline of experiments done on this proof of concept and some evaluation based on the outcome of these experiments in Chapter 4. We then go over related works in Chapter 5. We will discuss our findings in Chapter 6. Lastly, we conclude and recommend future work in Chapter 7.

Chapter 2

Rich ASTs versus Property-Based ASTs

In this chapter we describe our conceptual contribution, the definition of property-based ASTs. Section 2.1 will explain what an AST is in the most generic sense. Section 2.2 will explain what a rich AST is. Finally, Section 2.3 will explain what a property-based AST is, highlight its advantages, and compare it to rich ASTs.

2.1 Abstract Syntax Tree (AST)

Source code can be represented in different ways. The representation in which humans create and modify source code is a text string inside some text editor or IDE. This is great for humans because we are able to read the code clearly and make changes easily. Code is inherently structured. For example, in an object-oriented programming language such as Java, we write code in files, that consist of classes. Classes contain field declarations, methods, and constructors. There is a clear structure to the way we write code. The issue for any compiler, or tool, that has to perform operations or analysis on source code is that a text string is not structured, so we need a representation in which we are able to utilize this structure.

A structured and formal way of capturing the syntax of a code string is called an Abstract Syntax Tree (AST). An AST is a tree structure that represents the syntactic structure of the code. An AST consists of nodes in this tree structure. A node can have children, with these children representing smaller pieces of information. Each tree node represents a code construct, such as a declaration or if statement, in the parsed code. We can use parsers to parse code and retrieve an AST as the output of the parser. For example, an AST could have a file as the top node, with classes in the file as the children of that top node. Those classes would each be represented as a node, and would then have their respective children for their field declarations, methods and constructors. A simple AST is shown in Figure 2.1. In this figure, we can see that a variable assignment statement is split up into its constituent parts in the tree.

ASTs were originally designed for static code analysis such as syntactical and semantic checks on a program. However, as a secondary development, ASTs have been used as a basis for pattern matching to enhance what can be achieved with only regular expressions. This is because ASTs usually represent context-free languages, which are a superset of regular languages. Pattern matching involves a pattern, and an instance or some source to match against. The instance is often source code on which we are working. The pattern is some smaller snippet of code that we want to find in this source code and that is specified by the user. The pattern can have placeholder variables that match on anything. This allows the user to specify repetitive generic pieces of code for the pattern. Both pattern code and instance code are parsed to ASTs and we can then compare the nodes in these ASTs to see if they are

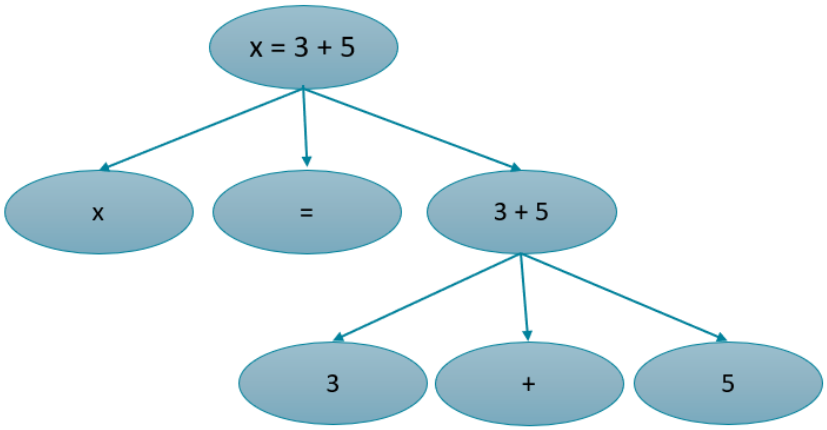

Figure 2.1: Example of a generic AST representing an assignment to the variable x

matching or not. Because of the structure of an AST, this is very efficient. Once a mismatch is detected, we immediately know that there is no match and we can stop searching. Another advantage is that pattern matching can be done with arbitrarily large patterns because of the recursive structure of ASTs. The specific type of pattern matching we are interested in is called concrete syntax pattern matching and will be explained in more detail with code examples in Section 3.1.1.

ASTs abstract away details so that the code can be used for a further purpose such as compiling, or in our case, refactoring. How much gets abstracted away also differs per parser and this is also one of the challenges we face. An example is layout information. Layout includes things such as comments, whitespaces, indentation and parentheses. Layout does not contain any semantical information so this is often lost during parsing because it is not relevant to the functionality of the code. An AST is provided by a parser and is specific to the language, and even to the parser. Different parsers for the same language can return different ASTs. Some examples of parsers are Eclipse C/C++ Development Tooling (CDT) (Foundation 2024) for C++ and libadalang for Ada (AdaCore 2024; TNO 2024).

2.2 Rich ASTs

We define rich ASTs to be ASTs that have syntactic information embedded in its structure. This comes in the form of the types of the nodes itself, as well as the possible children and attributes of those nodes. An example of an attribute could be the visibility of a class declaration (*public*, *private*, *protected*). What attributes and types of children a node will have will depend on the type of the node. For example, not all code constructs will have a visibility specifier so not all node types will contain that attribute. An example of an if-statement can be seen in Figure 2.2. In the figure, we can see that the type of the node itself contains syntactic information and that the relations between the parent and child are defined exactly, and are dependent on the parent node. The node is forced to adhere to some contract that we have defined. One of the means through which we can define this contract is a Java interface or class. An example of a Java class definition of this if statement can be seen in Listing 2.1. When dealing with this if-statement, we know exactly beforehand what possible children there are and what attributes the if-statement can have. These types of ASTs are parser-specific. The syntactic information is efficiently represented in the structure, and this also allows for efficient and powerful code analysis, rewriting, and optimization. The drawback is that this hinders the language-extensibility of a tool chain using them, since they are specific to one parser and therefore to one language.

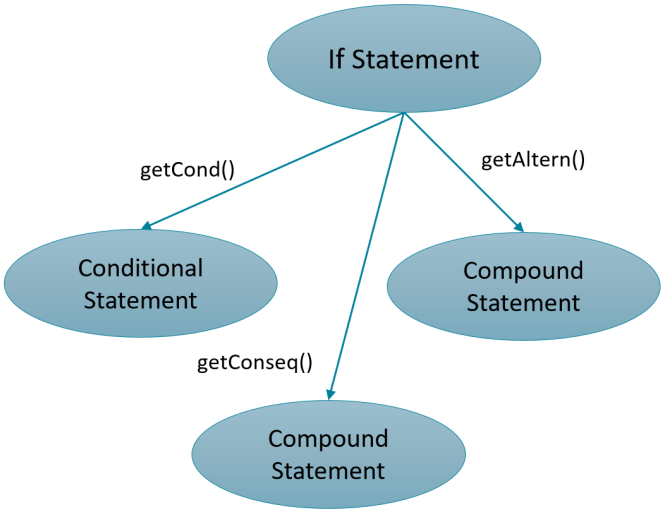

Figure 2.2: Example of a rich AST

```
1  public class IfStmt extends Node {
2
3      private ConditionalStmt cond;
4      private CompoundStmt conseq;
5      private CompoundStmt altern;
6      private String content;
7
8      public Node getCond() {
9          return this.cond;
10     }
11
12     public Node getConseq() {
13         return this.conseq;
14     }
15
16     public Node getAltern() {
17         return this.altern;
18     }
19
20     public String getContent() {
21         return this.content;
22     }
23 }
```

Listing 2.1: Java class that defines an if statement node

An example of a parser that returns a rich AST is the Eclipse CDT parser. The Eclipse CDT parser represents nodes as Java objects and has methods for the possible relations of each node. The pattern matching algorithm in Renaissance will match on the nodes in the following way. Given an instance node and a pattern node, it will first check that the types of the nodes are equal, if they are equal, it will check the possible children of the node and possible attributes such as field visibility or type specifiers. For the sake of brevity, a Java snippet of the equality checking component of a pattern matching algorithm for two node types can be seen in Listing 2.2. The example shows that by first discerning the type of the node, we

then know exactly what attributes or children we have to check to ensure that the match is correct. Pattern matching incorporates this equality checking and has additional logic for the handling of placeholders that are found in the patterns. The handling of placeholders is not in the code snippet. The important point to note is that for every node type, a piece of logic has to be written to match on that node type. This will require effort, and for every new language construct, corresponding logic has to be written if the tool wants to support that construct. We want to avoid having to rewrite specific logic for each possible construct, and that is where property-based ASTs can help us.

```
public boolean match(Node pattern, Node instance) {
    if (pattern instanceof IfStatement && instance instanceof IfStatement) {
        return match(pattern.getCond(), instance.getCond())
        && match(pattern.getConseq(), instance.getConseq())
        && match(pattern.getAltern(), instance.getAltern());
    }
    if (pattern instanceof ClassFieldDeclaration
    && instance instanceof ClassFieldDeclaration) {
        return match(pattern.getVisibilty(), instance.getVisibility())
        && match(pattern.getFieldType(), instance.getFieldType())
        && match(pattern.getFieldName(), instance.getFieldName());
    }

    // ...  cases for every node type

    return false;
}
```

Listing 2.2: Java snippet of a equality checking in a pattern matching algorithm using a rich AST

2.3 Property-Based ASTs

We define property-based ASTs to be ASTs that have a generic node structure, but contain syntactic information in the node properties. These node properties consist of the type of the node, relation to its parent, text content and offset. Language-specific properties can also be added but this is not in the scope of this project. An example of an if-statement can be seen in Figure 2.3. We can see that the properties of the node are added to the node as a sort of dictionary. This allows the language engineers to generalize the functionality of their written code since they do not need to take the possibly different structure and type of nodes into account when extending to new languages. This is in contrast to the rich AST described earlier, where we had to create logic for every type of node possible. A pattern matching algorithm for a property-based AST could look like the following. Given a pattern and instance node, we first check if they have the same type by retrieving the type property of the node from each node. If they match, we can then recursively match each child node in pattern and instance node, and return a match if all the children match. A commented Java snippet for the algorithm just described is available in Listing 2.3. Note that the actual pattern matching algorithm also needs to be able to handle placeholders, so this is a simplified version where we only care about pattern matching regular code. Logic wise, this is very similar to the pattern matching algorithm described for rich ASTs, but code wise, the logic only needs to be written once and can then be applied to all possible node types.

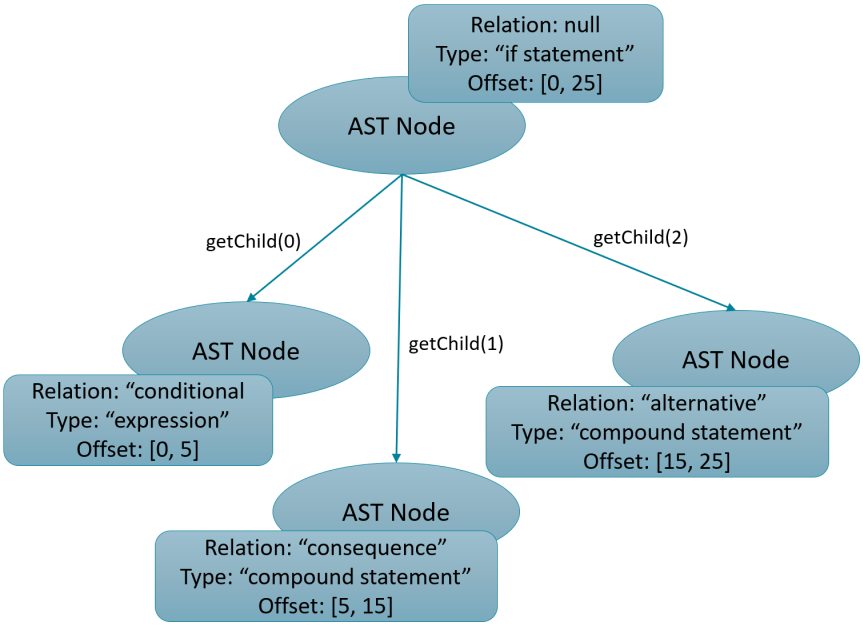

Figure 2.3: Example of a property-based AST

```java
public boolean match(Node pattern, Node instance) {
    // Check type
    if (!pattern.getType().equals(instance.getType())) {
        return false;
    }
    // Check amount of children
    if (pattern.getChildren().size() != instance.getChildren().size()) {
        return false;
    }
    // Match children
    int children = pattern.getChildren().size();
    for (int i = 0; i < children; i++) {
        // Match per child
        if (!match(pattern.getChild(i), instance.getChild(i))) {
            return false;
        }
    }
    // Passed all the checks so return true
    return true;
}
```

Listing 2.3: Java snippet of a simplified pattern matching algorithm using a property-based AST

This representation still allows for customised code analysis and rewriting but is less efficient than its rich counterpart. More checks have to be performed to ensure that we are in a valid state, whilst when using a rich AST, this is ensured by the structure of the node itself. An example is during the process of rewriting a node. When replacing or changing an existing node with a rich AST, we are forced to create a node that satisfies the structure of the node. When doing this with a property-based AST node, there is more freedom and as a result, more caution has to be taken when working with this. To give a concrete example, if we want to rewrite a for-loop into a while loop, with a rich AST, we are forced to provide a

conditional statement for the while loop because this is the structure of that specific node. This is not the case for a property-based AST. We can provide a parent node that is typed as a while loop without any conditional statement as its children. This would result in an ill-formed AST. This problem can arise when we want to rewrite the AST itself. If we want to rewrite the source code based on text rewriting, this issue disappears. Section 3.1 will give some reasoning as to why text rewriting is preferred over AST rewriting for transformations.

The light design of the property-based AST allows other ASTs to be mapped to a property-based AST in an easy manner. Engineers would need to extract the type of the node, the relations between each node and the names of those relations, offset and text content and they could then create a property-based AST by populating the respective property fields for each node. Because the node itself is generic, we do not need to translate the type of the node exactly to a matching type in this property-based representation, we can just store the type as a property.

When comparing these two types of ASTs and the pattern matching algorithm described with each of them, we notice some differences and similarities. First the similarities, the two ASTs contain essentially the same information, but represented differently. For pattern matching, we are also interested in the same information, namely, is the instance the same syntactically as the pattern? The main difference is in the code written in the language tool that uses these ASTs. In the case of a rich AST, we get a lot of guarantees from the structure of the nodes and we are able to safely assume properties and children of each node, resulting in clean and safe code. This comes at the cost of having to specify behaviour for every type of node, since they are not generic. On the other hand, a property-based AST requires more checks in the code to make sure that the instance matches the pattern, but the advantage here is that this can be done generically, and therefore does not require writing specific logic for every possible node type. However, the main advantage is the language extensibility of a tool written with a property-based AST because it is easier to map a rich (or any) AST to a property-based AST than it is to map any AST to a rich AST. Property-based ASTs can also be obtained through the means of generated parsers, such as Tree-Sitter (Brunsfeld 2024). These two factors combined mean that property-based ASTs are easy to obtain for multiple languages. Therefore, a tool that uses a property-based AST as its base will be easier to utilise for multiple languages.

Chapter 3

Renaissance Adaptation with Tree-Sitter

This chapter describes the proof of concept that we have built. We have adapted an existing tool called Renaissance with a parser generator that returns ASTs that can be seen as property-based ASTs. This adaptation shows a possible application of our conceptual contribution and also allowed us to measure the amount of time and changes needed to adapt an existing tool to our idea. It will also show that certain desired features are possible to be implemented with this approach. Section 3.1 will go into detail about the desired features themselves and the reasoning behind them. Section 3.2 will explain the adaptation.

3.1 Desirable Features

This section will explain some features that Renaissance already has and why we would want to preserve those features.

3.1.1 Concrete Syntax Patterns

A way to help the engineers refactor code is through the use of concrete syntax patterns. Concrete syntax patterns are blocks of code, that look like real code but can have placeholders for parts of it. The advantage of this is that it is very familiar to the user. And at the same time, it allows for specifying repetitive parts of the code. One alternative would be to specify an abstract syntax pattern, but working with abstract syntax patterns is more cumbersome and results in having to write large patterns for small pieces of code. To give an example, for a simple piece of code such as the code in Listing 3.1, the Eclipse CDT parser abstract syntax pattern looks like Listing 3.2. Another alternative would be to use regular expressions, but the learning curve with regular expressions is quite steep (Andersson and Hansson 2020) and it is much more intuitive for the users to work with something that looks similar to what they have worked with before.

```
1 int x = 5;
```

Listing 3.1: Simple line of code

```
1 -CPPASTSimpleDeclaration
2  -CPPASTSimpleDeclSpecifier
3  -CPPASTDeclarator
4    -CPPASTName
5    -CPPASTEqualsInitializer
6      -CPPASTLiteralExpression
```

Listing 3.2: Abstract syntax pattern for a simple line of code

There are two types of placeholders that can be used in a concrete syntax pattern. There are single placeholders and multiple placeholders. An example with a single placeholder is shown in Listing 3.3 and Listing 3.4. In this example, we want to match on any if-else-statement. In Listing 3.3, we have the code that is being analysed, and in it, a fragment of the code that we want to find. Listing 3.4 shows the code fragment that we want to find. This code fragment is written by the user. The dollar sign indicates a single placeholder variable. For example, `$COND` is a single placeholder that will match on any single statement in the conditional clause. This placeholder and its value can later be used when refactoring the code. `$TRUEBRANCH` and `$FALSEBRANCH` will both match on any single statement in that block of code. In this example, the tool will match on lines 2-6 of Listing 3.3, since this matches the pattern specified.

```
1 public boolean biggerThanFive(int x) {
2     if (x > 5) {
3         return true;
4     } else {
5         return false;
6     }
7 }
```

Listing 3.3: The code in which we want to match

```
1 if ($COND) {
2     $TRUEBRANCH;
3 } else {
4     $FALSEBRANCH;
5 }
```

Listing 3.4: The concrete syntax pattern with three single placeholders

The second type of placeholder is the multiple placeholder. The multiple placeholder will match zero or more nodes. We are able to match on multiple placeholders using a double dollar sign. An example of this is given in Listings 3.5 and 3.6. In the example, we can see that we have used a multiple placeholder to match on the arguments passed to the function call. In this case, the value of the multiple placeholder will be `x, y`.

Multiple placeholders can also match on whole lines of code. Another pattern is shown in Listing 3.7. This pattern will match on any block of code that starts with `int x = 5;` followed by an arbitrary amount of statements. In this case, the value of the multiple placeholder will be the two lines of code on lines 3 and 4.

```
1 public void createLocalVariables() {
2     int x = 5;
3     int y = 10;
4     int z = addTwoValues(x, y);
5 }
```

Listing 3.5: The code in which we want to match

```
1 int z = addTwoValues($$ARGUMENTS);
```

Listing 3.6: The concrete syntax pattern with a multiple placeholder capturing multiple arguments

```
1 int x = 5;
2 $$STATEMENTS;
```

Listing 3.7: The concrete syntax pattern with a multiple placeholder capturing multiple statements

3.1.2 Matching Based on Abstract Syntax Tree

When searching for code fragments, we want the user to be able to input concrete syntax patterns, but under the hood we want to match on the AST that corresponds to the concrete syntax pattern. This is because the AST will abstract away from details that are syntactically unimportant. This allows multiple concrete syntax patterns to map to the same AST. Things

such as layout, spacing and comments can be (optionally) ignored. This ensures that the tool will look for code fragments that are syntactically equivalent, regardless of how the code looks. Listings 3.8, 3.9, and 3.10 show three examples of a piece of code that parses to the same AST but are different in concrete syntax due to extra spaces or indents.

```
1 int x = 5;
```

Listing 3.8: Example 1

```
1 int      x    =     5   ;
```

Listing 3.9: Example 2

```
1    int   x    =   5  ;
```

Listing 3.10: Example 3

3.1.3 Code Rewriting Based on Offsets

The previous subsection explained why matching based on ASTs is desirable, however this approach ignores certain elements of the code in the matching process. Most notably, comments and layout are ignored. If we are only searching for specific patterns, this is not a problem but if we want to also rewrite matched code fragments, it becomes a slight problem. Rewriting a code fragment by modifying the AST can cause the layout (amount of indentations, spacing, etc) to be incorrect. This is caused by the pretty-printer that converts the AST back into a block of code. Since ASTs do not contain such layout information, the original layout information will be lost and replaced with layout information that is determined by the pretty-printer. Although this is not a problem for the functionality of the program, it is a problem for the developers maintaining it because it affects the readability of the code. However AST nodes contain information about the location of the node itself. This is referred to as an offset. An offset is either a combination of the row and column of the start and end of the AST node in the source file, or two integers indicating the start and end point of the AST node in the source file in terms of characters. Using the second definition, an AST node that has offsets 15 and 25 means that the node starts at the 15^{th} character (inclusive) and ends at the 25^{th} character (exclusive) of the source file.

To minimize losing layout information, rewriting based on the offsets of a matched pattern in the target is the preferred approach. The text to be rewritten to is specified as raw text and is used to rewrite the matched pattern. The location of the text to be rewritten is specified by the offset. The entire matched pattern gets replaced with the text to be rewritten to. If this is done in a back-to-front manner, the rewriting only changes the matched code fragments and leaves the rest of the code as it was.

The trade-off for using text rewriting when compared to using rewriting based on a modified AST to implement transformations is the correctness of the code for the compiler. The text rewrite does not need to adhere to any compiler rules and could be anything. This may result in code that does not even compile, or pass tests. This is where the code compilation check and test run that follows a refactoring come into play. Industrial software systems will have these things incorporated in their development pipeline based on standard industrial software practices (Fluri, Fornari, and Pustulka 2024).

3.2 Proof of Concept

Our proof of concept involves adapting Renaissance (Laar and A. Mooij 2022) and replacing the CDT parser with a generated parser from Tree-Sitter (Brunsfeld 2024). Tree-Sitter ASTs are akin to the notion of propety-based ASTs that we have defined in the previous chapter, and Tree-Sitter already supports a large number of languages (more than 80) that are maintained by a community.

3.2.1 Renaissance

Renaissance is an existing language tool developed by TNO (Laar and A. Mooij 2022; A. J. Mooij, Eggen, et al. 2015; A. J. Mooij, Ketema, et al. 2020). This approach has previously been

used with success for code transformations on large-scale industrial codebases (A. J. Mooij, Eggen, et al. 2015; A. J. Mooij, Joy, et al. 2016). Renaissance supports various features such as concrete syntax pattern matching, code refactoring, and creating code graphs from multiple source files. Renaissance can detect single and multiple placeholders and use them in the rewriting. The default for this is the single- and double-dollar sign. Extending Renaissance to different languages has been a major time sink in the past, so this also shows a very applicable use case of using property-based ASTs to make refactoring tools more language-parametric.

The pattern matching component of Renaissance is called Rejuvenation and currently utilises a language-specific representation of the AST to function. This makes it harder to extend the Rejuvenation library to other languages a big part of it would have to be rewritten to fit to another parser. The components of the library are handwritten to match these specific parsers. If we want to plug in a different language or even a different parser for the same language, we would need to rewrite the whole tool to fit to this parser.

There are currently two versions of Renaissance, one for C++ using the CDT parser (Foundation 2024) and one for Ada (TNO 2024) using Libadalang (AdaCore 2024). The one that we will be using for the adaptation is the one for C++.

3.2.2 Tree-Sitter

Tree-sitter (Brunsfeld 2024) is a parser generator tool. Given a grammar that defines a language, it will parse the source code into a tree of that language. It parses the source code into a Concrete Syntax Tree (CST). This entails that it parses every individual token, even things such as parentheses or semicolons. It is possible to treat this CST as an AST because Tree-sitter differentiates between named and anonymous nodes. Named nodes contain the important details that are needed for an AST to function whilst unnamed nodes contain less important information, such as string literals like parentheses or semicolons. Anonymous nodes are represented in the grammar as simple strings while named nodes have been given explicit names in the grammar. If the user only uses the named nodes, the CST that Tree-sitter provides functions as an AST.

In the Java binding (Bonede 2024) that we use, nodes are represented as Java classes with fields for type, children, text content, offsets and other properties. This makes it adhere to our definition of a property-based AST and makes it a good fit for testing the language parametricity of property-based ASTs. It also contains utility functions for tree-related operations such as retrieving siblings or finding descendants within a given byte (offset) range. Our focus is on the genericness of the AST and the representation of types, text content and offsets as fields (properties) of the node.

The advantage of Tree-sitter is that the tree provided is generic, even for different languages and thus we are able to reuse code written for the tree of one language for another. It also has written grammars for most common languages, so they are supported out of the box when using Tree-sitter.

3.2.3 Adaptation

We can use Tree-Sitter to make Renaissance more easily extendable whilst maintaining access to the useful features it provides. Our aim was to replace the CDT parser in Renaissance with the Tree-Sitter generated parsers. A version of Renaissance is available in Java, and there is a Java binding for Tree-Sitter (Bonede 2024). To plug in the Tree-Sitter generated parsers, we had to adapt the Renaissance tool itself to be more generic because it was written in a way to work on specific CDT nodes. Figure 3.1 shows the structure of the original Renaissance and Figure 3.2 shows the structure of the adapted Renaissance. By giving Tree-Sitter a grammar, it will generate a parser for us based on that grammar. We can then use this parser as the base of Renaissance. These figures also highlight the increased language-extensibility of the

adapted Renaissance because all that we need to provide is a grammar for a new language. Most of the changes for the adaptation had to be made in the Rejuvenation component of Renaissance.

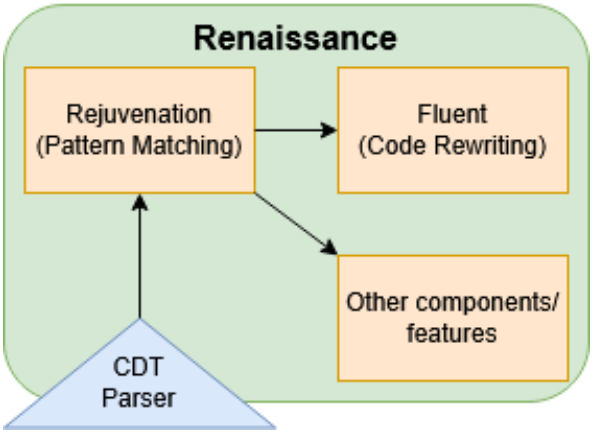

Figure 3.1: Simplified diagram of the structure of the original Renaissance

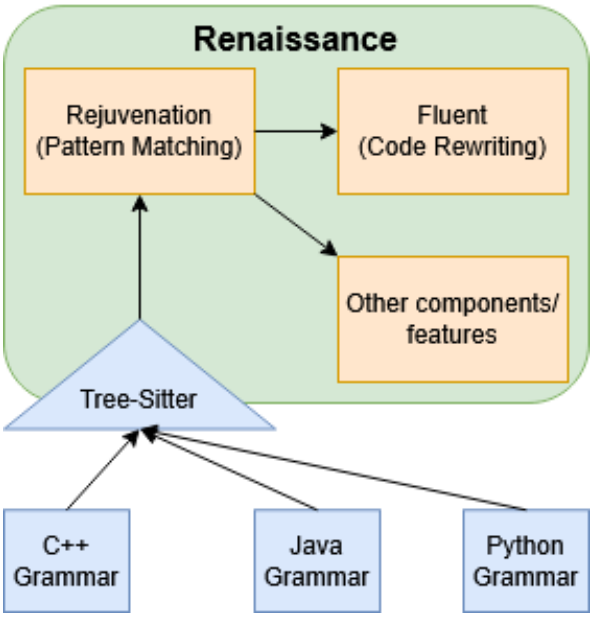

Figure 3.2: Simplified diagram of the structure of the adapted Renaissance. These three grammars only serve as examples and Tree-Sitter is not limited to only these three.

The original implementation used CDT nodes that were represented as Java objects to perform the various features. There was logic for every possible CDT node type. We copied the classes that dealt with these CDT node types and changed them to use Tree-Sitter nodes. The logic that was not related to the pattern matching on the nodes was left as it was. The only exception being the logic that detects placeholders. Because some languages do not support dollar signs in their syntax, an extra check has to be added to check if a placeholder passed is valid in the language undergoing refactoring. This resulted in a large portion of duplicate code, but this allowed us to easily switch between the original implementation and the adapted implementation for experimentation purposes.

Issues

There was an issue with the Java binding that involved concurrent memory accesses. For example, a small case that only matched a single statement pattern to a single statement target code worked. But as soon as we added more complexity in the form of more statements or placeholders, we ran into the JVM throwing a memory access violation. We tried accessing the tree in a separate project with the same pattern and target codes, but without all the complexity that came from the Renaissance implementation, and it worked as expected. So we speculate that the issue is caused by something in Renaissance that clashes with the Java binding. We conjecture that this is based on concurrent memory accesses in the JVM. The Java binding uses native methods to provide its functionality. This means that the actual methods are written in a different language, in this case it is C. The way the JVM handles this is that it allocates memory for the non-Java part of the application separately. We think that the issue is that the Java binding does not allow for multi-threading because it throws memory access violation exceptions when trying to use the binding in the tool. To circumvent this, we implemented a workaround where we copied the retrieved AST from the Tree-Sitter Java binding into a custom Java object. This ensures that all the memory accesses remain in the Java-part of the JVM. However, this is slower and takes up more memory because we are duplicating the tree.

Chapter 4

Evaluation

This chapter will go over the experiments we have performed and the results of those. The focus of the experiments is to measure various metrics, such as effort required, correctness of output and ease of extendibility with a new language. We will also look at the amount of effort it took for the adaptation itself, measured in time, and lines of code. The goal of this evaluation is to test on a few quality metrics. These quality metrics will be explained in Section 4.1. The experiments will be explained in more detail in Section 4.2. Section 4.3 will go over the results of performing the experiments.

4.1 Quality Metrics

The goal of the evaluation is to test two things. Firstly, we want the adapted Renaissance to be easily usable in multiple languages, so we want to test the ease of plugging in a new target language for refactoring. Secondly, we also want to make sure that the adapted Renaissance exhibits the same behaviour as the original.

We measure the effort to plug in a new language based on how many lines of code need to be changed, and how much knowledge is required from the developer. We measure the correctness of the adaptation by comparing the two versions and performing refactoring operations with them. If they both return the same results, we know that the observable behaviour is the same.

We also measure the speed and memory usage of the two versions. This is in order to give an impression of the performance of the adapted Renaissance in comparison to a benchmark (the original Renaissance). We aim to measure the speed in milliseconds and the memory usage in megabytes.

We also measure the effort taken to develop the adaptation. This will be done in terms of lines of code changed and time taken in months.

4.2 Experiments Outline

We have created experiments that test various features, and we measure the effort and the correctness of those experiments. Correctness here means the difference between the expected output versus the actual output and we are interested in what might cause the disparity.

An overview of the experiments is given in Table 4.1. The idea is that we want to compare our adapted Renaissance to the baseline, which is the original Renaissance implementation that works on C++. We measure the amount of lines of code that need to be added or changed. We realize that this is not a perfect measurement, but it should give a general idea of how much effort it takes to perform these experiments. The first two experiments mentioned in the table are the simpler experiments and test the basic functionality of the tool. The last experiment involves using JsonCPP exercises to test the refactoring and pattern matching

capabilities on bigger codebases and with more complicated cases. The JsonCPP exercises we refer to are some exercises that are made to get users that are unfamiliar with Renaissance up to speed with it. The exercises contain several refactoring and analysis problems that are to be performed on an open source library called JsonCPP (Lepilleur 2024).

Experiment	Features Tested
Extending the Renaissance Java implementation with a new language via Tree-Sitter	Extensibility of tool
Comparative analysis of simple code rewriting (i.e. function parameter swap) in C++, Java and Python	Pattern matching, replacing, pattern creation
JsonCPP exercises in C++ and Python	Pattern matching, replacing, pattern creation

Table 4.1: Experiments and their tested features

4.2.1 Extending Renaissance

The first experiment covers the effort of extending Renaissance with a new language. Specifically, we want to measure what the refactoring engineer needs to do to be able to use Renaissance with their desired language. This is measured in lines of code and in a qualitative estimation of the knowledge required to do so. We extend Renaissance with Java and Python because we require it for the subsequent experiments.

4.2.2 Simple Code Rewriting

The second experiment covers using adapted Renaissance to do basic pattern matching and code transformations. We use small excerpts of code and small patterns to find and replace code. We want to do this in multiple languages, so we aim to use the same code fragments in different languages, even if the syntax is different. The approach was to create test cases in C++ first. Then translate these test cases to other languages, specifically Java and Python. Listing 4.1 shows a piece of code in C++ before rewriting, with Listing 4.3 showing the pattern we want to find in this piece of code. We want to replace the found pattern with the pattern in Listing 4.4. After applying this, we want to get the code in Listing 4.2.

```
void main() {
    some_func(1, 3);
    other_func(4, 2);
    x = 3;
}
```

Listing 4.1: The test case before rewriting (C++)

```
void main() {
    other_func(3, 1);
    some_func(2, 4);
    x = 7;
}
```

Listing 4.2: The expected test case after rewriting (C++). Notice the switched parameters

```
$CALL1($ARG1, $ARG2);
$CALL2($ARG3, $ARG4);
$VAR = 3;
```

Listing 4.3: The pattern we want to find (C++)

```
$CALL2($ARG2, $ARG1);
$CALL1($ARG4, $ARG3);
$VAR = 7;
```

Listing 4.4: The pattern we want to replace with (C++)

The same execution can be seen for Java and Python in Listings 4.5, 4.6, 4.7, 4.8 and 4.9, 4.10, 4.11, 4.12, respectively. Note that for Python, we use a different placeholder because

Python does not allow for dollar signs in its syntax. The single dollar sign is replaced by `SS__` and the double dollar sign is replaced by `MM__`. Caution has to be taken with this, because it could conflict with variable names.

```
public static void main() {
    some_func(1, 3);
    other_func(4, 2);
    x = 3;
}
```

Listing 4.5: The test case before rewriting (Java)

```
public static void main() {
    other_func(3, 1);
    some_func(2, 4);
    x = 7;
}
```

Listing 4.6: The expected test case after rewriting (Java). Notice the switched parameters

```
$CALL1($ARG1, $ARG2);
$CALL2($ARG3, $ARG4);
$VAR = 3;
```

Listing 4.7: The pattern we want to find (Java)

```
$CALL2($ARG2, $ARG1);
$CALL1($ARG4, $ARG3);
$VAR = 7;
```

Listing 4.8: The pattern we want to replace with (Java)

```
def main():
    some_func(1, 3)
    other_func(4, 2)
    x = 3
```

Listing 4.9: The test case before rewriting (Python)

```
def main():
    other_func(3, 1)
    some_func(2, 4)
    x = 7
```

Listing 4.10: The expected test case after rewriting (Python). Notice the switched parameters

```
SS__CALL1(SS__ARG1, SS__ARG2)
SS__CALL2(SS__ARG3, SS__ARG4)
SS__VAR = 3
```

Listing 4.11: The pattern we want to find (Python)

```
SS__CALL2(SS__ARG2, SS__ARG1)
SS__CALL1(SS__ARG4, SS__ARG3)
SS__VAR = 7
```

Listing 4.12: The pattern we want to replace with (Python)

In this manner, we can test the correctness of the C++ transformations with the original Renaissance implementation to compare. This does not work for other languages directly, but we can know from the C++ test case what the results should be for the other languages.

4.2.3 JsonCPP exercises

The following exercises were performed:

1. Print the AST of one file.
2. Find a specific statement in all the files and report its location and show its AST. The specific statement can be seen in Listing 4.13.
3. Create an inheritance tree of all the classes, using classes and their parents.
4. Match on a specific pattern and replace it with a different code fragment using placeholders. The specific pattern with placeholders can be seen in Listing 4.14. We want to replace it with the pattern in Listing 4.15.
5. Count the amount of pre and post-increments/decrements in the update expression of for loops. Afterwards, change post-increments/decrements to pre-increments/decrements.

6. Match on a pattern that describes a code fragment that joins elements using a separator. Afterwards, transform them into more complex or simpler code fragments. The different patterns with differing levels of complexity can be seen in Listings 4.16, 4.17 and 4.18. The transformations were done with the goal of transforming the code fragment into one of the other patterns.
7. Count the amount of four different if statements without else clauses: The amount in general, if statements with brackets, if statements with brackets and a single statement in those brackets, and finally if statements without brackets.

```
document_ += ',';
```

Listing 4.13: The statement we want to find for Exercise 2

```
Token token;
token.type_ = $TYPE;
token.start_ = $START;
token.end_ = $END;
```

Listing 4.14: The pattern we want to find for Exercise 4

```
Token token {$TYPE, $START, $END};
```

Listing 4.15: The pattern we want to replace with in Exercise 4

```
for ($T $INDEX = $BOUND; $INDEX < $SIZE; ++$INDEX) {
    if ($INDEX > $BOUND)
        $SEPARATOR_STATEMENT;
    $$ASSIGNMENT_STATEMENTS;
}
```

Listing 4.16: The simple pattern we want to find for Exercise 6

```
for ($T $INDEX = $BOUND; $INDEX != $SIZE; ++$INDEX) {
    $$INITIAL_STATEMENTS;
    if ($INDEX != $BOUND)
        $SEPARATOR_STATEMENT;
    $$ASSIGNMENT_STATEMENTS;
}
```

Listing 4.17: The medium pattern we want to find for Exercise 6

```
$T $INDEX = $BOUND;
for (;;) {
    $$INITIAL_STATEMENTS;
    if (++$INDEX == $SIZE) {
        $$LAST_STATEMENTS;
        break;
    }
    $$SEPARATOR_STATEMENTS;
    $$LAST_STATEMENTS;
}
```

Listing 4.18: The complex pattern we want to find for Exercise 6

4.2.4 Time Taken and Memory Usage Measurements

To measure the time taken and memory used for the various tasks, we used a simple approach. Before each task, the time was noted using Java's built-in `System.nanoTime()` and

the memory was noted using the `Runtime` class in Java. We also collect garbage first to attempt to minimize the memory in use and to avoid inaccuracies. The same metrics were measured again after the task was completed, and the difference shows us the time taken and the amount of bytes used. An example of the code written for this is shown in Listing 4.19. We do this five times for each task and take the average. This is in order to minimize noise through factors such as garbage collection or JVM optimization. The results will initially be retrieved in nanoseconds and bytes, but we convert them to milliseconds and megabytes, respectively. The goal of taking these measurements is to give a realistic impression of the performance of the adapted Renaissance and to compare it to the performance of the original. This can be used to gauge whether this approach is feasible in a real-world situation. Since these micro-measurements will not give a good insight on a small task, we will not use these for the simple code rewriting tasks.

```
long startTime = System.nanoTime();
Runtime runtime = Runtime.getRuntime();
// Collect garbage first
runtime.gc();
long beforeUsedMemory = runtime.totalMemory() - runtime.freeMemory();

runTask();

long afterUsedMemory = runtime.totalMemory() - runtime.freeMemory();
double memoryUsed = (afterUsedMemory - beforeUsedMemory) / (Math.pow(10, 6));
double elapsedTime = (System.nanoTime() - startTime) / (Math.pow(10, 6));
System.out.println("Time used by task: " + elapsedTime + " ms");
System.out.println("Memory used by task: " + memoryUsed + " megabytes");
```

Listing 4.19: The code to measure time taken and memory used

4.3 Results

This section will go over the result that we obtained. Firstly, the results of the experiments are explained in Subsection 4.3.1. The adaptation effort is also covered in Subsection 4.3.2.

4.3.1 Outcome of Experiments

Supporting new languages

The adaptation allows for easy plugging in of new languages given an existing Tree-Sitter grammar for the language. The users only need to provide the node type name of the node that syntactically represents a compound statement or block. This is needed for some matching logic in the tool. Specifically, the matching algorithm needs to detect when we are dealing with a compound statement to match on all the statements that are in the compound statement. Other than this, the other potential thing that has to be manually added is placeholder names for languages that do not allow for the dollar sign in their syntax. By default, the single and double dollar sign are used for single and multi placeholders, respectively. However, some languages, such as Python, do not allow dollar signs and will therefore not parse correctly. The addition of this language parametric information has to be done in two files and will then be propagated throughout the tool. It only requires adding about 20 lines of code.

Simple code rewriting

Simple code rewriting was tested and found to be correct and easy to switch between languages. One test case and pattern in multiple languages was described in Section 4.2.2. Things such as (single and multiple) placeholders, statements, variables, and replacing all worked as expected.

From the perspective of an engineer using adapted Renaissance, their knowledge of how Renaissance works and functions on one language can be carried over to another one. Naturally, they would need to know the syntax of the new language but there would not be the need of learning or setting up a new tool (with the potential exception of the small changes mentioned previously).

JsonCPP exercises

For each of the exercises, the experience and results are explained here.

Print AST This was a straightforward exercise that worked. All we had to do was use the adaptation and its classes instead of the original classes, with C++ as the language.

Find a specific statement We could use the same pattern and code used in the original version here. The same locations were matched, so that means the behaviour was identical.

Create inheritance tree This exercise did not give identical behaviour. This was due to the pattern used and an error in the logic of our adaptation. The pattern used is shown in Listing 4.20. This pattern should match on any class and remember its parents in the `$$PARENTS` placeholder. More specifically, it can be 0 or more parents. However, our adaptation requires at least one parent for it to function, so the resulting inheritance tree is incorrect.

```
1 class $CLS : $$PARENTS { $$DECLARATIONS; };
```

Listing 4.20: The pattern used to find classes and their parents

Match on a specific pattern and replace This exercise works as expected. It also showed that rewriting files works, although a noteworthy thing is the indentation. The indentation looks different, but this also applies to the original version.

Count pre and post-increments/decrements in for loops This exercise does not work as expected. This is due to the fact that the adaptation does not differentiate between `--$ITER` and `$ITER--`, and identically for `++$ITER` and `$ITER++`. This is due to the way the matching algorithm is implemented. A different approach was used to circumvent this problem. Instead of having separate patterns for pre-increment, post-increment, pre-decrement, post-decrement, a single pattern capturing them all, and afterwards filtered to count the pre and post-increments. A more detailed explanation of this will be given in Section 4.3.1.

Match on a generic pattern and replace This generally works as expected, but there are some incorrect matches that have the same root cause as the errors in the increment counting in the for loops. On the correctly matched patterns, it did perform the rewriting correctly.

Count if statements The exact amount of matches can be found in Table 4.2. The table shows the results of the original Renaissance and the adapted Renaissance. The table shows a disparity between the two versions. The disparity can be attributed to various factors, one of them being the macros in C++. Upon closer inspection of some of the matches, we noticed that there are matches that originally originated from a macro. The pre-processing step of C++ is done in the original Renaissance implementation, but for the adaptation, this was left as a lower priority task and, therefore, not implemented. This explains some of the difference between the two versions. An important point to note is that the amount of if statements with brackets and without brackets should add up to the total amount of if statements found. This holds for both versions of Renaissance.

If statement type (without else clauses)	Original	Adapted
Any if statement	1065	816
If statement with brackets	496	323
If statement with brackets and a single statement	304	147
If statement without brackets	569	493

Table 4.2: Counts of different if statement types in the original Renaissance versus the adapted Renaissance

Further Explanation of Errors

This section will explain the errors found during the JsonCPP exercises. The errors are caused by implementation faults or unimplemented components, so it is not an inherent fault of our approach of using property-based ASTs as the base of the tool. As explained before, Tree-Sitter has two types of nodes, named and anonymous. We have written our adaptation to work on the named nodes of the AST provided because it was believed that all the important information is stored in named nodes.

The problematic part here is that it parses binary operators like the plus and minus sign as unnamed nodes. This results in the pattern matching algorithm finding matches when it is not supposed to. For example, `int x = 2 + 3` and `int x = 2 - 3` would return a match, even though they are different pieces of code. The binary operator is essentially ignored during the pattern matching algorithm.

In the exercise in which the aim was to count the pre and post-increments in for loops, the tool matched on too many instances. As said before, the tool did not differentiate between the different pre and post-increments, and this was because the `+` and the `-` in the pattern were ignored, and thus the pattern consisted only of a placeholder, matching on anything.

We originally used four different patterns for this, the only difference between these patterns being in the update clause of the for loop. These were meant to capture pre and post-increments/decrements separately and then process them accordingly. But facing the problem of named and unnamed nodes in Tree-Sitter, we instead used a single pattern capturing all for loops, and then retrieved what was captured by the placeholder in the update clause. The result is a string, containing the actual code of the update clause. We use this to determine whether it was a pre or post-increment/decrement and keep track of the count in this way. We were also able to use this to write the pattern for the rewriting of post-increments/decrements to pre-increments/decrements. The only caveat is that the indentation was not correct for the cases where no changes were made. But the cases where changes were required were manually checked to be correct.

In the exercise in which we wanted to match on patterns with differing levels of complexity and transform them, the issue was in the part of the pattern that detects for loops. In the medium pattern, the for loop segment of the pattern looks like `$T $INDEX = $BOUND; $INDEX`

`!= $SIZE; ++$INDEX`. The issue here is in the condition part of the for loop. The inequality operator `!=` is a binary operator and is therefore seen as an unnamed node. This makes it so that this part of the pattern is equal to a lot of different things, and the extra matches that we found matched on code that should have matched with `$INDEX < $SIZE` as the pattern for the condition instead of `$INDEX != $SIZE`.

Lastly, we did not have the pre-processing step that can happen in C++ code implemented in the adapted Renaissance. As mentioned before, this also caused some of the results to be different. Especially in the exercise counting the amount of if-statements but we suspect that this would also cause inaccuracies for the exercise creating the inheritance tree, if the pattern matching worked correctly.

Speed and Memory Measurements

As explained in the previous Section, we took the average of five runs for each JsonCPP Exercise where we measured speed and memory usage. The results can be found in Table 4.3. The full results can be found in Appendix A.1.

Exercise	Time (Original)	Memory (Original)	Time (Adapted)	Memory (Adapted)
Print AST of one file	409.78 ms	20.37 MB	412.39 ms	74.18 MB
Find a specific statement	2211.38 ms	65.34 MB	11702.13 ms	270.04 MB
Create inheritance tree	3161.54 ms	95.04 MB	19504.20 ms	248.68 MB
Match on a specific pattern and replace	2337.42 ms	49.14 MB	11924.95 ms	154.29 MB
Count pre and post-increments/decrements	4109.68 ms	218.85 MB	20588.36 ms	299.20 MB
Match on a generic pattern and replace	3039.61 ms	90.38 MB	32332.01 ms	254.08 MB
Count if statements	4223.42 ms	164.38 MB	52646.76 ms	157.17 MB

Table 4.3: Time and memory measurements for the JsonCPP Exercises (average over five runs)

From the results, we can see that the adapted Renaissance takes longer to run and also uses more memory. A closer look at some of the runs in Appendix A.1 also shows some variance in the measurements for memory usage. This is likely due to various factors in the JVM that are difficult to control such as when it decides to collect garbage. We have tried to minimize this as much as possible by telling the JVM that it should collect garbage before running the task, but this is seen as a hint and not an explicit command to the JVM.

The overall takeaway from these results is that the adapted Renaissance takes more time to run and also uses up more memory while doing so. This is an expected outcome, given that the adaptation is not optimized and also duplicates the tree because of issues in the Tree-Sitter Java Binding. It cannot be said with certainty that a tool using a property-based AST is slower and more memory intensive to the extent showcased by our results. But we theorise that it can not be faster than a tool based on rich ASTs because the same information has to be checked. Logically, we need to perform the same comparisons to determine whether or not a pattern matches in the source code.

4.3.2 Adaptation Effort

The amount of code changes is shown in Table 4.4. An important thing to note is that the number of lines of code added is inflated due to copying existing classes to use for the adap-

tation. Another consideration to make is that prior to working with Renaissance, the team member in charge of the adaptation had no experience with working with such tools or writing them. Lastly, the adaptation was simply aimed at providing a proof of concept, so it is not yet usable for production in the way that Renaissance is. Further development would be required to make it fully fault-proof.

The majority of the development time was spent on understanding the original Renaissance and finding the right places to insert Tree-Sitter. One of the challenges was understanding what the code does and performing equivalent operations but with Tree-Sitter nodes instead of the original CDT nodes.

We estimate that the amount of lines actually changed/added/deleted (so excluding copying existing files) is closer to around 1500 lines of code. This shows that a rudimentary adaptation to using property-based ASTs is manageable, but further care has to be taken to fully retain the original functionality of the tool.

Type of change	Effort
LOC added	6704
LOC deleted	348
Files changed	73
Time taken	4 months

Table 4.4: Code changes during the adaptation effort

Chapter 5

Related work

There are many ways to automatically refactor code or help with refactoring code more efficiently and research has already been conducted on this. This chapter aims to provide some context around our approach and shine a light on what has already been done towards making refactoring easier. The chapter is split up into three sections, Section 5.1 for techniques and research on those techniques. Section 5.2 is focused on tools that are usable for a developer. Finally, Section 5.3 is on other works on intermediate representations.

5.1 Techniques

Our work uses an AST-based approach on code analysis and refactoring. Other approaches are possible and this section highlights a few approaches.

5.1.1 Clustering

Clustering tries to create clusters of classes or functions by looking at the relationships between them. In other words, trying to create clusters by finding pieces of code that interact with each other a lot. Clustering can be used to refactor code at the function level (Lung et al. 2006). Functions are divided into entities (statements) and each entity has its own attributes, which consist of the things a statement of code would exist of such as, but not limited to, variables, constants, function calls, operators and semicolons. They are then clustered based on a similarity measure based on these entities and attributes. The resulting clusters are shown as a tree, and ill-structured code fragments can be identified by inspecting the tree and these become the candidates for refactoring. Although the clustering tree provides heuristic advice on how to restructure the function, ultimately this decision is taken by the software designer. Afterwards, the low-cohesive functions will be decomposed into several smaller fragments, some of which will become their own function. This process happens manually.

5.1.2 Machine Learning

Another way of refactoring is through the use of machine learning. Kumar, Satapathy, and Murthy (2019) has used 10 different machine learning classifiers on 25 source code metrics extracted from 5 open source Java projects to perform a comparative analysis between the classifiers. They also compared 3 different data sampling methods. The task for the comparison was predicting whether or not a refactoring was necessary at the method level. They observe that there are statistically significant differences between the performance of some of the classifiers. They also conclude that method level refactoring prediction using metrics from the source code with machine learning classifiers is possible. Aniche et al. (2020) has done something similar but also included process metrics, such as number of commits in a

class, and ownership metrics, such as number of authors. They have a list of 20 possible refactoring operations, spread out between the class-level, method-level and variable-level. They then trained binary classifiers for each of these possible operations to predict whether or not that operation is necessary. The scale of their research is bigger because they use a variety of different codebases, mainly from GitHub, but also from other sources. They found that process and ownership metrics play an important role in the creation of better models and that models trained with heterogeneous projects generalize better and have a good performance.

5.1.3 Large Language Models

Large Language Models (LLMs) have been accumulating interest, especially by the introduction and subsequent widespread exposure of ChatGPT (OpenAI 2022). An empirical study has been conducted on the code refactoring capabilities of ChatGPT (DePalma et al. 2024). The study found that ChatGPT is able to consistently refactor code and enhance it based on quality attributes such as readability or complexity. It is also able to discern between code before and after refactoring and identify the purpose of the refactoring. However, the authors conclude with saying that ChatGPT should be used as an aide during refactoring and that a human programmer should still be overseeing the overall operation because we can not depend on ChatGPT just yet. This is due to the fact that it was not able to perfectly refactor all the cases used in the study correctly and because it was not able to understand the broader context of the code fragments.

The approach of using a LLM is different from using a tool based on an AST or some underlying representation of the code. The main difference is that a LLM is essentially a black box, and users can not be certain about why the model responded in a certain way. Whereas a tool based on an AST is written by an engineer and there is code that represents the logic used for the decisions of the tool. The advantage of using a tool like ChatGPT is that it is easy to use and does not require learning any new skills. However, this can come with unexpected costs, such as loss in hand-on skills in the long term, as argued by the empirical study (DePalma et al. 2024).

5.2 Tools

This section will go over some tools and place them in the context of our designed proof of concept of Renaissance with Tree-sitter. The presence of these different tools for different languages also shows that tooling already exists, but requires knowledge of different tools for different languages. All of these tools also require maintenance from different developers.

5.2.1 Rascal

Rascal (Bos et al. 2011) is a meta-programming language that is used for source code analysis and transformation. It is used for legacy system rejuvenation, reverse engineering, reengineering and development of Domain Specific Languages (DSLs). It currently provides support for C, C++, Java, PHP, Python and JVM bytecode. Users are also able to define their own grammar and use Rascal with their own defined language.

Rascal aims to provide an easy-to-use and easy-to-combine set of primitives for the user to work with because this makes it easier to zoom in on the details whenever necessary. According to the creators, during meta-programming, the details are especially important, so not abstracting them away from the user allows the user to easily address issues when they arise. Rascal's strength is that it provides all the tools you need to perform code analysis and transformation in one package.

5.2.2 Spoofax

Spoofax (Wachsmuth, Konat, and Visser 2014) is a language workbench. It is used to develop new programming languages and provide IDE features for the newly developed language. Spoofax provides an environment that integrates syntax definition, name binding, type analysis, program transformation, and code generation. It provides metalanguages that abstract over the components that handle these features, therefore letting the language designer focus on the design. All of these components are provided if the designer specifies a grammar for the language that the designer wants to create.

Spoofax uses Stratego (Bravenboer et al. 2008) to perform code transformations. The user specifies the rules that operate on the AST. This AST structure is dependent on the grammar that defines the language, which is provided by the user. The user is able to use strategies to describe their code transformations. Strategies manipulates how and when rules are used during code transformations.

The main purpose of Spoofax is to aid in the development of new languages, and thus it is not very suited for working with existing languages. If one would want to utilise all of Spoofax's capabilities in an existing language, a grammar would have to be specified for that language. If the language is popular enough, the language might already have IDE-support and tools that can perform code-based operations such as type analysis or code transformation.

5.2.3 Ast-grep

Ast-grep (Darkholme 2024) is an open-source code tool that allows for the users to syntactically search through ASTs and rewrite code if matches are found. It uses Tree-Sitter as its base for parsing. It is used for searching, linting and rewriting. It supports using metavariables (placeholders) and can handle a wide array of languages due to the availability of Tree-sitter grammars.

Users are able to search for code in two ways. The first is to write a concrete syntax pattern as described before in Figure 3.3 and Figure 3.4. The second method is specifying a rule, and this rule can specify various properties of the AST node we want to match on. Including, but not limited to, node type, relation to its parent, order of child relative to siblings and concrete syntax. This is quite expressive, but during testing, it was found to be rather cumbersome to specify slightly more complex rules. It was also not possible to specify concrete syntax patterns that consists of multiple statements on the top-level, which was deemed a big shortcoming.

The main similarity between our adapted Renaissance and ast-grep is the utilization of Tree-Sitter as its core. The pattern matching algorithm is similar, but when diving deeper into it, the way patterns are specified by the user are vastly different. Ast-grep has a bigger focus on using rules to express a pattern, whilst adapted Renaissance solely uses concrete syntax patterns.

Ast-grep functions as a Command-Line-Interface (CLI), but also provides a Python and JavaScript API for users to utilize ast-grep. It also hosts a web playground to test and get acquainted with ast-grep.

5.2.4 OpenRewrite

OpenRewrite by Moderne (2024) is an automated refactoring ecosystem designed for large-scale transformations on source code. It has an automatic refactoring engine that is able to run prepackaged, open-source recipes for common large-scale transformations such as framework migrations, security fixes, stylistic fixes, and version upgrading/downgrading. These are meant to be generic and help out developers that want to perform common refac-

toring operations in a quick and easy manner. Developers are also able to create their own recipes if they have more specific needs.

OpenRewrite currently supports a small selection of languages, build tools and frameworks. Notably, Java, Kotlin, Groovy, Maven and Gradle. Recipe authors can contribute to the recipe catalog for other engineers to use.

This shows the limited nature of some of these refactoring tools. They are built with specific languages in mind, and as a result, are hard to extend to new languages.

5.2.5 Structural Search and Replace

Structural Search and Replace (SSR) (Mossienko 2004) is a tool integrated in JetBrains IntelliJ IDEA. It allows users to search through their code structurally, similar to what ast-grep provides. The advantage of SSR is that it comes with the IDE, so users do not need to install a separate third-party tool to use it. This makes it accessible. However, it is only available in Java, Kotlin and Groovy, hence why it is only available in the IntelliJ version of JetBrains' Integrated Development Environments (IDEs).

This again, shows the limited reach of language tools. In this case, it makes sense because the tool only supports the languages supported by the IDE it is packaged into.

5.3 Intermediate Representations

Our research is mainly focused on the intermediate representation underlying a language tool. There are other intermediate representations that have been proposed for a more language-agnostic approach, and this section will go over some of them.

5.3.1 Language Agnostic Abstract Syntax Tree

Language Agnostic Abstract Syntax Trees (LAASTs) (Curtis 2022) are ASTs that are designed to be an abstraction away from language-specific elements to a common interface for refactoring tools. The goal of a LAAST is to simplify static analysis of source code in different languages, which is akin to our goal of making refactoring tools easier to extend with new languages. The way they approach this is different from ours. They define a set of common language constructs that most languages contain, such as classes and fields, functions, blocks, and control flow statements, to give a few examples. Programming languages may contain constructs that are unique to that language, but usually these are representable using some set of instructions using the common constructs.

The goal of the LAAST is to serve as a common layer of abstraction for multiple programming languages. Depending on the languages chosen, the structure of the LAAST will vary. This depends on the language constructs that have to be supported, which depends on the languages themselves. A LAAST needs to be designed for the set of languages that the we want to work on. This differs case by case, and there is no single LAAST code representation that will work on all languages.

To use a LAAST, users are required to map from their language-specific AST to a LAAST. After this, code analysis can be performed generically and does not require redevelopment of the analysis component of the tool.

Whilst significantly decreasing the workload of using a new language with a refactoring tool, there is still some language-parametric work that has to be done for each language for them to work. This comes in the form of providing the mapping from a language-specific AST to a LAAST. Next to this, if the LAAST designed does not support the language with which we want to extend, we are unable to use the LAAST unless we alter the design of the LAAST to work with the new language.

5.3.2 Separator Syntax Tree

A middle point between CSTs and ASTs are Separator Syntax Trees (SSTs) (Aarssen and Van Der Storm 2020). The main goal of SSTs is to perform high-fidelity source code transformations. In other words, preserving layout whilst transforming source code. The SST by storing the text content between the nodes, the separators, along with the nodes. The SST can be seen as an AST augmented with additional information about the separators. Because of this, it is possible to pattern match syntactically on the source code by using the AST portion of the SST and then use the separators to preserve the original layout of the source code after applying some transformations.

SSTs can be reconstructed from ASTs (or CSTs) if the nodes are annotated with source location information, such as offsets and length. Using this information, all the separators that are not part of an AST node can be retrieved and stored.

Chapter 6

Discussion

This chapter will discuss our findings and compare our approach to some of the related works from the previous chapter.

6.1 Comparison to Other AST representations

The goal of our property-based ASTs and Language Agnostic Abstract Syntax Trees (LAASTs) (Curtis 2022) is similar, but there are some differences worth pointing out. First of all, a LAAST has to be designed for a set of languages and multiple LAAST representations can exist. Each of them with the ability to handle different languages. This hinders extensibility if the language is not supported by the LAAST. If the language is supported, a converter from the language-specific AST to the LAAST is required. In contrast, a property-based AST is a looser representation, aimed to fit any (object-oriented) programming language. This makes it able to be extended with any language as long as a property-based AST representation is available through a parser (or parser generator such as Tree-Sitter) or a converter from a language-specific AST to the property-based AST. The core difference is in the language-extensibility of the tool utilizing either of these ASTs. With a LAAST, the set of languages supported has to be determined before hand for the design of the LAAST. After this phase, the set of languages supported is set in stone. With a property-based AST, there is no limitation set from the beginning. The tool can be extended with any language if one of the two conditions (parser or converter) mentioned before hold. This disparity is due to the LAAST having specific node types for every language construct, while the property-based AST has generic nodes with the type stored as a property of the node.

The SST (Aarssen and Van Der Storm 2020) is essentially an augmented AST and we think that with some minor adjustments, it can also be treated as a property-based AST with the added benefit of having access to the separators in the usage of code rewriting.

6.2 Converting to a Property-Based ASTs

Another point of discussion is that we claim that a property-based AST can also be obtained through the means of a converter from a language-specific (or rich) AST to a property-based AST. We have not shown a concrete implementation of this; however, we have briefly described a possible rudimental methodology of a converter. A converter for Java and C++ to a LAAST (Curtis 2022) has been made to show the applicability of a LAAST. Although not exactly the same converter, this suggests the possibility of a convert that is able to translate between ASTs.

6.3 Performance and Issues of Adapted Renaissance

This research has shown a proof of concept (adapted Renaissance) using a parser generator to obtain property-based ASTs for a refactoring tool. Our proof of concept involved adapting Renaissance with Tree-Sitter, and then testing it with a few experiments to check for correctness. Some of the experiments returned results that were incorrect. This might be seen as a flaw of the property-based ASTs, but this is not the case. As argued before in Section 4.3, this is not due to an inherent fault in our approach with property-based ASTs, but rather an implementation issue due to some details in Tree-Sitter.

Our proof of concept is slower and uses more memory than the original Renaissance. This is not necessarily due to the use of property-based ASTs because it can also be attributed to the workaround that we used with the Java binding for Tree-Sitter.

Other tools exist that can work with multiple languages, notable ones have already been mentioned in Section 5. Rascal (Bos et al. 2011) and Spoofax (Wachsmuth, Konat, and Visser 2014) are both tools that allow for working with multiple languages. However, Spoofax is designed for creating new DSLs and programming languages and is not suited for refactoring purposes. Rascal is designed for multiple languages and provides this language parametricity in the form of grammars, similar to Tree-Sitter. However, mapping from an AST from any given parser to an AST that Rascal can use is not possible. In this sense, the "entry point" differs. Rascal allows for new languages by specifying a formal grammar for that language. A property-based AST can be used by either using a converter or a parser that returns such an AST. The parser that returns such an AST can be retrieved from a parser generator, and those also require a formal grammar to provide their parsing functionality. This shows that there are more ways to start using a property-based AST than there are ways to start using Rascal, allowing for more flexibility on the side of the refactoring engineer.

6.4 Multi-Language Systems

The use of property-based ASTs allows for increase language support in refactoring tools. This is advantageous because of factors such as less development effort, less maintenance efforts and easier access and better understanding for refactoring engineers working with multiple languages. Where a refactoring tool that can work on multiple languages really shines is in multi-language systems. This allows the refactoring engineer to efficiently analyse and transform codebases written in multiple languages. Whereas, normally, multiple tools have to be used that each target a different portion of the codebase written in a different language. Having this functionality in one tool increases the overview and clarity of the refactoring and also speeds up the refactoring process.

Chapter 7

Conclusion and Future Work

This chapter will first conclude the paper. Afterwards, we will recommend some future work that could prove to be interesting venues of research based on our findings.

7.1 Conclusion

We have introduced the problem of legacy systems in large software systems. The issue with these legacy systems is that they are often outdated and lack in code quality. This is because they were written in times with different code practices and technologies. Another factor is that they were often extended in a sub-par manner. As a result of this, the technological debt of these systems is very high. These systems have to be refactored with the goal to make them last in the long term.

Refactoring large systems is not cost-efficient to do manually. Since refactoring often involves repetitive tasks, this is the ideal venue to use an automatic refactoring tool. Many refactoring tools exist for many languages, but refactoring tools often only cater towards one or a selection of a few languages. This results in requiring the refactoring engineer to learn and understand more tools. On the tool development side, this requires maintenance on multiple tools and can result in duplicated logic for different languages.

To combat this, we would like to make existing tools easier to extend. This allows us to leverage the logic of the tool that is written for one language, for another language. This decreases the tool development effort and allows the tooling engineers to pool their efforts into one tool instead of multiple tools for multiple languages.

We introduce the notion of a property-based AST to help tooling engineers make their tools more language parametric. This property-based AST consists of generic nodes with any amount of children. Each node contains properties that reflect the semantic information that is in the source code from which the AST was parsed. This generic representation allows us to abstract away from language-specific ASTs such as rich ASTs. Rich ASTs contain the syntactical information in its structure and are language-specific. One way of obtaining property-based ASTs is through mapping existing ASTs to a property-based variant. This is done by retrieving the relations, node types, and other properties and converting them into a generic version where each node only has properties and connections to its parents and children. Property-based ASTs might also be obtained through a parser itself. A good example of one such parser is a parser generated by Tree-Sitter. These parsers are generic, and the AST returned by them adhere to our definition of a property-based AST.

To show an use case and the feasibility of property-based ASTs, we have adapted an existing Renaissance implementation in Java with Tree-Sitter. This Renaissance implementation originally was meant to work on C++, and with the adaptation, it should work on any language for which a Tree-Sitter grammar exists. Experiments were performed on this adaptation to test its extensibility to other languages and to test if the adaptation was performed

correctly. The results are promising and show that with little effort, a new language can be plugged in and used when refactoring. This shows that using a property-based AST can aid in making refactoring tools more language-parametric.

7.2 Future Work

7.2.1 Other Programming Paradigms

The type of programming languages that were focused on during this research were object-oriented programming languages. This was due to their popularity in the industry, and widespread applicability. Different programming paradigms exist, and it would be interesting to look into refactoring those. Investigating how a tool works for a functional programming language and seeing if a property-based AST could be used to make it more language-parametric is a proposed next step for research if we want to explore different paradigms.

7.2.2 Extending Property-Based ASTs to Property-Based SSTs

SSTs allow for code rewriting based on AST transformations instead of text rewriting like we currently do with the adapted Renaissance. The main drawback of AST rewriting is the issue of layout, but this is no longer an issue with SSTs. It would be interesting to see if a tool could be built that uses a property-based SST as its foundation. Combining the language parametricity of the property-based AST with the high-fidelity transformations of the SST.

7.2.3 Extensions to Adapted Renaissance

Another venue for future work is to use a converter to get a property-based AST for our adapted Renaissance to use. This will show, with a concrete example, that a property-based AST can be obtained from any AST through the use of a converter. An extra bonus for doing this with the adapted Renaissance would be the access to the exact same environment for the experiments. Allowing for the same experiments to be ran in the same environment, but with a different method of obtaining a property-based AST.

Another extension to a tool using a property-based AST could be the usage of language-specific properties. What we mean by this is that a tool can use more property fields that are language specific when dealing with that language. For example, property fields specific to C++ could exist, and these would not exist in Java. Since information is represented as properties, a language-specific property field could be empty for ASTs originating from other languages but exist for the language that we are interested in. The tool itself can be made to know the language with which it is working and specifically look for those language-specific properties when possible. This would mean that more language-specific features or shortcuts would exist within the tool, but this would not be visible to the user utilizing the tool. This would add additional complexity of language-specific properties to the user in the form of having to specify more properties in the AST. However, language-specific properties can be made optional by having the tool check for the existence of such properties. This makes this feature an extension to the property-based ASTs that engineers can opt out from if the additional complexity is not worth the gain.

An existing feature in the original Renaissance was the creation of code graphs. Code graphs are graphs that represent the connectivity of the code system being analyzed. It contains information about what parts of the system call or use what other parts of the system. This is used for static analysis of the system and allows the refactoring engineers to gain more insight regarding the system under analysis. We also wanted to implement this and retrieve such a code graph using the adapted Renaissance, but due to time constraints, this was not possible. This is a potential direction for exploration using Renaissance.

There is also an experimental version of Renaissance in Python that is being developed parallel to this project at TNO. This experimental version also uses a property-based AST as its base and is currently integrated with Clang, showing that the idea works with black-box parsers. If Tree-Sitter can also be used in this version, it will show that our concept of property-based ASTs can work with both black-box and generated parsers. However, this is a recommendation for future work because of the scope of this project.

Bibliography

Aarssen, Rodin TA and Tijs Van Der Storm (2020). "High-fidelity metaprogramming with separator syntax trees". In: *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pp. 27–37.

AdaCore (2024). *Libadalang*. URL: https://github.com/AdaCore/libadalang.

Andersson, Adam and Ludwig Hansson (2020). *Modernizing the Syntax of Regular Expressions*.

Aniche, Mauricio et al. (2020). "The effectiveness of supervised machine learning algorithms in predicting software refactoring". In: *IEEE Transactions on Software Engineering* 48.4, pp. 1432–1450.

Bonede (2024). *Tree-sitter Java binding*. URL: https://github.com/bonede/tree-sitter-ng/tree/main.

Bos, Jeroen van den et al. (2011). "Rascal: From algebraic specification to meta-programming". In: *arXiv preprint arXiv:1107.0064*.

Bravenboer, Martin et al. (2008). "Stratego/XT 0.17. A language and toolset for program transformation". In: *Science of computer programming* 72.1-2, pp. 52–70.

Brunsfeld, Max (2024). *Tree-sitter*. URL: https://tree-sitter.github.io/tree-sitter/.

Curtis, Jacob (2022). "On language-agnostic abstract-syntax trees: student research abstract". In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pp. 1619–1625.

Darkholme, Herrington (2024). *ast-grep*. URL: https://ast-grep.github.io/guide/introduction.html.

De Groot, Jelle et al. (2012). "What is the value of your software?" In: *2012 Third International Workshop on Managing Technical Debt (MTD)*. IEEE, pp. 37–44.

DePalma, Kayla et al. (2024). "Exploring ChatGPT's code refactoring capabilities: An empirical study". In: *Expert Systems with Applications* 249, p. 123602.

Fluri, Jasmin, Fabrizio Fornari, and Ela Pustulka (2024). "On the importance of CI/CD practices for database applications". In: *Journal of Software: Evolution and Process*, e2720.

Foundation, Eclipse (2024). *Eclipse CDT (C/C++ Development Tooling)*. URL: https://projects.eclipse.org/projects/tools.cdt.

Krasner, Herb (2021). "The cost of poor software quality in the US: A 2020 report". In: *Proc. Consortium Inf. Softw. QualityTM (CISQTM)* 2.

Kumar, Lov, Shashank Mouli Satapathy, and Lalita Bhanu Murthy (2019). "Method level refactoring prediction on five open source java projects using machine learning techniques". In: *Proceedings of the 12th innovations on software engineering conference (formerly known as India Software Engineering Conference)*, pp. 1–10.

Laar, Piërre van de and Arjan Mooij (2022). "Renaissance-Ada: Tools for Analysis and Transformation of Ada code". In: *ADA USER* 43.3, p. 165.

Lepilleur, Baptiste (2024). *JsonCpp*. URL: https://github.com/open-source-parsers/jsoncpp.

Lung, Chung-Horng et al. (2006). "Program restructuring using clustering techniques". In: *Journal of Systems and Software* 79.9, pp. 1261–1279.

Minelli, Roberto, Andrea Mocci, and Michele Lanza (2015). "I know what you did last summer-an investigation of how developers spend their time". In: *2015 IEEE 23rd international conference on program comprehension*. IEEE, pp. 25–35.

Moderne (2024). *OpenRewrite*. URL: https://docs.openrewrite.org/.

Mooij, Arjan J., Gernot Eggen, et al. (2015). "Cost-effective industrial software rejuvenation using domain-specific models". In: *Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings 8*. Springer, pp. 66–81.

Mooij, Arjan J., Mabel M. Joy, et al. (2016). "Industrial software rejuvenation using open-source parsers". In: *Theory and Practice of Model Transformations: 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings 9*. Springer, pp. 157–172.

Mooij, Arjan J., Jeroen Ketema, et al. (2020). "Reducing Code Complexity through Code Refactoring and Model-Based Rejuvenation". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 617–621. DOI: 10.1109/SANER48275.2020.9054823.

Mossienko, Maxim (2004). "Structural search and replace: What, why, and how-to". In: *OnBoard Magazine*.

OpenAI (2022). *Introducing ChatGPT*. URL: https://openai.com/index/chatgpt/.

Schuts, Mathijs TW et al. (2022). "Large-scale semi-automated migration of legacy C/C++ test code". In: *Software: Practice and Experience* 52.7, pp. 1543–1580.

Telea, Alexandru and Lucian Voinea (2011). "Visual software analytics for the build optimization of large-scale software systems". In: *Computational Statistics* 26.4, pp. 635–654.

TNO (2024). *Renaissance Ada*. URL: https://github.com/TNO/Renaissance-Ada.

Wachsmuth, Guido H, Gabriël DP Konat, and Eelco Visser (2014). "Language design with the Spoofax language workbench". In: *IEEE software* 31.5, pp. 35–43.

Acronyms

AST Abstract Syntax Tree

LAAST Language Agnostic Abstract Syntax Tree

SST Separator Syntax Tree

CST Concrete Syntax Tree

CDT C/C++ Development Tooling

CLI Command-Line-Interface

IDE Integrated Development Environment

LLM Large Language Model

DSL Domain Specific Language

SSR Structural Search and Replace

JSON JavaScript Object Notation

JVM Java Virtual Machine

Appendix A

A

A.1 JsonCPP Exercises - Speed and Memory Measurements

	Time (Original)	Memory (Original)	Time (Adapted)	Memory (Adapted)
Run 1	398.43 ms	21.69 MB	405.58 ms	73.40 MB
Run 2	382.13 ms	21.65 MB	406.98 ms	78.69 MB
Run 3	414.24 ms	17.18 MB	413.53 ms	74.52 MB
Run 4	441.15 ms	24.30 MB	429.78 ms	65.33 MB
Run 5	412.93 ms	17.05 MB	406.07 ms	78.96 MB
Average	409.78 ms	20.37 MB	412.39 ms	74.18 MB

Table A.1: Time and memory measurements for JsonCPP Exercise 1: Print AST of one file

	Time (Original)	Memory (Original)	Time (Adapted)	Memory (Adapted)
Run 1	2221.67 ms	64.28 MB	11204.28 ms	226.77 MB
Run 2	2196.77 ms	67.26 MB	11493.59 ms	247.91 MB
Run 3	2191.29 ms	64.25 MB	11540.90 ms	234.90 MB
Run 4	2253.77 ms	65.54 MB	12320.58 ms	304.43 MB
Run 5	2193.39 ms	65.36 MB	11951.28 ms	336.17 MB
Average	2211.38 ms	65.34 MB	11702.13 ms	270.04 MB

Table A.2: Time and memory measurements for JsonCPP Exercise 2: Find a specific statement

	Time (Original)	Memory (Original)	Time (Adapted)	Memory (Adapted)
Run 1	3325.25 ms	95.04 MB	19023.75 ms	172.03 MB
Run 2	3138.55 ms	113.64 MB	20197.03 ms	132.05 MB
Run 3	3180.94 ms	109.28 MB	19350.02 ms	341.71 MB
Run 4	3355.67 ms	96.45 MB	20174.50 ms	350.04 MB
Run 5	3307.28 ms	60.78 MB	18775.69 ms	247.59 MB
Average	3161.54 ms	95.04 MB	19504.20 ms	248.68 MB

Table A.3: Time and memory measurements for JsonCPP Exercise 3: Create inheritancee tree

	Time (Original)	Memory (Original)	Time (Adapted)	Memory (Adapted)
Run 1	2258.62 ms	33.53 MB	12099.93 ms	144.04 MB
Run 2	2221.72 ms	66.38 MB	11908.03 ms	211.77 MB
Run 3	2393.93 ms	48.63 MB	11817.00 ms	122.47 MB
Run 4	2349.68 ms	49.45 MB	11954.57 ms	102.58 MB
Run 5	2463.17 ms	47.71 MB	11791.22 ms	190.57 MB
Average	2337.42 ms	49.14 MB	11924.95 ms	154.29 MB

Table A.4: Time and memory measurements for JsonCPP Exercise 4: Match on a specific pattern and replace

	Time (Original)	Memory (Original)	Time (Adapted)	Memory (Adapted)
Run 1	4084.95 ms	221.62 MB	22424.10 ms	302.12 MB
Run 2	4348.98 ms	263.28 MB	20349.04 ms	345.10 MB
Run 3	4049.47 ms	221.58 MB	20746.52 ms	339.78 MB
Run 4	4025.53 ms	167.94 MB	19658.92 ms	295.03 MB
Run 5	4039.48 ms	219.82 MB	19763.20 ms	213.95 MB
Average	4109.68 ms	218.85 MB	20588.36 ms	299.2 MB

Table A.5: Time and memory measurements for JsonCPP Exercise 5: Count pre and post-increments/decrements

	Time (Original)	Memory (Original)	Time (Adapted)	Memory (Adapted)
Run 1	2943.48 ms	84.69 MB	32488.31 ms	219.84 MB
Run 2	2942.81 ms	96.04 MB	32576.10 ms	302.43 MB
Run 3	3240.29 ms	79.45 MB	32091.18 ms	244.55 MB
Run 4	3035.34 ms	84.69 MB	32447.77 ms	256.14 MB
Run 5	3036.15 ms	107.02 MB	32056.68 ms	247.42 MB
Average	3039.61 ms	90.38 MB	32332.01 ms	254.08 MB

Table A.6: Time and memory measurements for JsonCPP Exercise 6: Match on a generic pattern and replace

	Time (Original)	Memory (Original)	Time (Adapted)	Memory (Adapted)
Run 1	4290.70 ms	158.96 MB	52968.47 ms	234.67 MB
Run 2	4232.13 ms	196.21 MB	52504.58 ms	130.74 MB
Run 3	4223.11 ms	161.90 MB	52301.89 ms	143.74 MB
Run 4	4216.54 ms	110.05 MB	52681.46 ms	116.12 MB
Run 5	4154.64 ms	194.34 MB	52777.39 ms	160.57 MB
Average	4223.42 ms	164.38 MB	52646.76 ms	157.17 MB

Table A.7: Time and memory measurements for JsonCPP Exercise 7: Count if statements