Automated Refactoring of Rust Programs

Algorithms and implementations of Extract Method and Box Field

Per Ove Ringdal

Thesis submitted for the degree of
Master in Informatics: programming and systems architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

Automated Refactoring of Rust Programs

Algorithms and implementations of Extract Method and Box Field

Per Ove Ringdal

Automated Refactoring of Rust Programs

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Refactoring is the practice of changing code without altering its behavior. Rust is a language with an *ownership model* where lifetimes are statically resolved and it has macro support. In this thesis we develop algorithms and implementations for the Extract Method and Box Field refactorings for Rust. Automated refactoring tools are often not correct. Here we compose the refactorings by smaller *micro refactorings* which can easily be understood independently. We ran the implementation on open source projects, identified problems with lifetimes, and handled it correctly.

Contents

Acknowledgements

I would like to thank my supervisors, Volker Stolz and Martin Steffen for supervising me in this thesis, providing me valuable feedback.

I would also like to thank my family for supporting me on my work on this thesis.

Chapter 1

Introduction

Refactoring is a technique from software engineering that allows, in general terms, to restructure and reorganize code with the goal of improving the quality of the code base its structure, etc. without changing the semantics of the program.

Software refactoring was invented independently by Bill Opdyke and Bill Griswold in the late 1980's [15, 14]. They define software refactoring as the systematic practice of improving application code's structure without altering its behavior. Opdyke's thesis [26] contains a set of primitive and composite refactorings where each refactoring has a set of preconditions. The preconditions consist of one or more boolean functions that the program must satisfy for the refactoring to be valid. If the refactoring is not valid, it may change the semantics and it should therefore not be applied. Here we will use a similar definition of equivalent semantics as Opdyke did in his thesis. To compare the semantics of two programs we look at the main functions. If the two programs are given the same input and the resulting output is the same then their semantics is equivalent. This should be true for all possible input to the programs. Micro refactorings is a term used by Schäfer [28] where a larger refactoring, such as Extract Method, is composed into a series of small refactorings, that can be understood, implemented and tested independently.

Software refactoring gained popularity in the end of the 1990's and at the start of the 2000's when automated tools were introduced (Smalltalk refactoring browser). A book containing a catalog of refactorings for Java was released in 1999 [13]. New methodologies such as TDD, XP [2] and Agile [4] became popular which also relied on software refactoring which focused on the reuse of software, applying software patterns and the ability to change software within weeks, days or even hours while minimizing the probability to introduce errors.

Refactorings can be done manually by a developer as shown in Fowlers book. Here the developer finds a section in a code that needs improvement, either by looking at the code, finding *code smells*, or using a static analysis tool. Then, there is a series of steps repeatedly modifying code, recompiling and testing to preserve behavior. This process can be tedious, and is prone to introducing errors, and several IDEs such as Eclipse, IntelliJ and Visual

Studio have therefore support for automated refactoring. An example in IntelliJ IDEA with the IntelliJ Rust plugin is shown in Figure 1.1. First, in (a), an `if-else` expression is selected as input to the refactoring. Then the Extract Method refactoring is selected in the user interface. The resulting code is shown in (b) and (c). In (b) the selected expression is replaced with a method call to the new function `get_char`. In (c) the new function declaration is shown, and the body of the function is the expression selected in (a).

Figure 1.1: Extract Method in IntelliJ Rust

This allows the developer both to refactor code faster, and with greater confidence. However, correctness of automated refactoring tools is a problem [12]. Unit tests are often used to test whether a refactoring was semantics preserving or not. The test suite may be incomplete and refactorings may therefore change the semantics of a program [10, 11, 20].

Refactoring is often done as a part of a development cycle, where the developer wants to make changes to the implementation without making changes to the semantics. Here the goal of the code change can be to improve the readability, reduce the complexity or change the implementation from one pattern to another.

Software metrics can be used to analyze the quality or the complexity of a code base. Cyclomatic complexity [23] is an example of a software metrics, and it was first described by Thomas J. McCabe in 1976. He defines the Cyclomatic complexity v of a program as

$$v = e - n + 2p$$

where e is the number of edges, n is the number of vertices and p is the number of connected components in the control-flow graph of the program. For a single function the number of connected components will always be 1 so we can simplify the equation to

$$v = e - n + 2$$

when we deal with a single function. He used this number to measure the structuredness of a program and as an indication to whether a program

needed to be simplified. The Cyclomatic complexity was used as an upper bound of a program, meaning that a program with a high Cyclomatic complexity should be improved to reduce the score. McCabe used 10 as an upper bound for the Cyclomatic complexity in his article.

As an example, an `if` expression, has two possible edges to the next statement, and it will increase the Cyclomatic complexity of a program by 1. To decrease the Cyclomatic complexity for the function containing the `if` expression, one can extract the `if` expression into a new function, using the Extract Method refactoring. This will, however, increase the overall Cyclomatic complexity of the program.

Based on Cyclomatic complexity, there is also Cognitive complexity [27], which also focuses on nesting in a program. The Cognitive complexity may also be reduced using the Extract Method refactoring.

Rust [8] is a language that claims to offer memory safety while having a minimal runtime overhead and no garbage collection. Rust does that using the *ownership model*, which is a static analysis determining the lifetimes of values, allowing the compiler to insert code for allocating and releasing resources at compile time. Rust also has support for *hygienic macros* [19]. Hygienic macros guarantee that identifiers are not accidentally captured. Refactoring macros can however be a problem [24]. Rust has a Module system, allowing code to have a logical structure, placing the code in different files and using paths to refer to different modules. A single file can occur as a submodule in different files, making types occurring in the submodule dependent upon the parent. As the language is relatively new it does not yet have the same support in IDEs as more established languages have.

We actively searched for refactoring opportunities for Rust and found a git-commit on the Rust Language repository[1] containing several refactorings, which we used to base one of the implemented refactorings on.

In this thesis we will specify and implement refactoring algorithms for Rust. We will adopt the Extract Method refactoring in Java by Schäfer. Based on the git-commit, and the Extract Class refactoring by Fowler, we will specify the refactorings Box Field and Unbox Field and develop algorithms for these.

The Extract Method refactoring will be composed of micro refactorings, similar to the steps presented by Schäfer. The main structure will be similar, but some of the steps will be different as the Rust language have some language constructs that are different from Java and a side by side list is shown in Section 3.1.

We will also, based on the git commit presented earlier, present an algorithm for the Box Field and Unbox Field refactorings. Box Field is similar to Extract Class with one field where the new class is the existing type `Box` which is a part of the standard library. We will also present an algorithm for the Unbox Field refactoring which is, conceptually, an undo operation of Box Field.

We will provide an implementation of the Extract Method and Box Field algorithms that uses the Rust compiler library for parsing and type

[1] https://github.com/rust-lang/rust/pull/64374/commits

checking. We also implemented a CLI tool that is used to invoke the refactorings on Rust projects. The refactorings will only work on files that are part of a project, and the project must compile before refactoring. The implementation is available as a public git-repository[2]. We ran experiments where we tested the implementation on two open source projects.

We also developed a client and server communicating over the Language Server Protocol [25]. The Language Server Protocol enables a server to provide language services such as syntax highlighting, refactoring and code completion. The server is not dependent upon an IDE, instead, a client that supports LSP can easily be adapted to use the server. The client was developed for Visual Studio Code. This enables the use of the refactorings from Visual Studio Code. We will not measure software metrics in this thesis.

The refactorings could also have been implemented using the IntelliJ Rust plugin, but we wanted the tool to be invocable from the command line, and the refactoring implementations not to be dependent upon UI-components. Another library that could have been used to implement the refactorings is *Rust analyzer*, an "IDE backend" for Rust, implemented as a language server, but Rust analyzer had its first alpha release 2020-04-20.

Organization In Chapter 2 we will give the necessary background for the terminology used in the thesis, including concepts in compiler theory used for the refactoring algorithms, concepts in Rust that are relevant for this thesis and existing refactoring support for Rust. In Chapter 3 we present the Extract Method refactoring for Rust, and the microrefactorings that it is composed by. The Box Field and Unbox Field refactorings are presented in Chapter 4. Chapter 5 gives an overview of the implementation of the refactorings, the CLI tool and the language server. The experiments for Extract Method and Box Field that were run using the CLI tool are shown in Chapter 6. The thesis is summarized and concluded in Chapter 7.

[2]https://github.com/peroveri/em-refactor

Chapter 2

Background

In this chapter we will give the necessary prerequisites to the refactoring algorithms that are shown later in this thesis.

First, in Section 2.1 we will look at terms used for refactoring from compiler theory. Then, in Section 2.2 we will go through the key features of Rust used in this thesis. In Section 2.3 gives an overview of the existing refactoring support for Rust. Some challenges with refactoring Rust are shown in Section 2.4.

2.1 Compiler theory for refactoring

A compiler, in the context of programming languages, is a program that translates from a higher level language to a lower level language [1]. The phases of a compiler typically consist of:

1. Lexical analysis
2. Syntax analysis (parsing)
3. Semantic analysis
4. Intermediate code generation
5. Machine-independent code optimizer
6. Code generator
7. Machine-dependent code optimizer

The lexical analysis will convert a sequence of characters into a sequence of tokens with some additional information attached. The parser then converts the token sequence to an abstract syntax tree (AST). The parser may also produce a desugared version of the language, where some of the languages features are replaced to simplify the subsequent analysis and code generation. The semantic analysis phase will perform static analysis such as type checking. So far the steps used to apply refactorings are similar.

After that an intermediate representation (IR) may be produced to allow for some machine-independent code optimizations. Finally the code for

the target machine is generated. For refactoring we may just need the two or three first steps as the target language is the same as the input language. The semantic analysis and machine-independent code optimizer phase may consist of some transformations and optimizations which have similar definitions in refactoring.

2.2 The Rust programming language

We will now take a look at some key concepts of the language that are relevant for the refactorings that we seek to support. For a general introduction to the language the Rust programming language book [18] is available online which explains most of the features of Rust.

Rust has a syntax similar to C with curly brackets marking regions such as `Blocks` and semicolons terminating `Statements` inside `Blocks`.

2.2.1 Structs

Rust has support for C-like `structs` which declares a new type with members. We will focus on `structs` with named fields in this thesis. Rust also have the tuple `struct` declaration which have positional members instead of named ones, `union` declaration and `enum` declaration, but they are not focused on in this thesis as they are similar to the named `struct` declarations.

2.2.2 Functions

Functions can be placed inside modules or function bodies, or they can be placed inside `Impl` sections which makes them an *associated* function. Associated functions which have a parameter called `self` are called *methods*, where `self` is an required instance of the `struct`.

Listing 1 shows the function `main` at line 1 occurring at module level. At line 2, a new instance of the `struct` declared at line 5 is instantiated and bound to the identifier `foo`. At line 3, the method `bar` declared at lines 7 to 9 is invoked.

```
1  fn main() {
2    let foo = Foo {field: 1};
3    print!("{}", foo.bar());
4  }
5  struct Foo { field: i32 }
6  impl Foo {
7    fn bar(&self) -> i32 {
8      self.field
9    }
10 }
```

Listing 1: A `struct` with the method `bar`

2.2.3 Types

A type may be seen as a protection against unintended use of the underlying representation of the constructs in a language [3]. For instance, if a and b are integers, then the expression $a + b$ can be interpreted as the sum of the two

variables. But if c is a string, then $a - c$ might not make sense and therefore it could result in a type error.

Strongly typed languages have guarantees that the program will run without type errors. If they are also statically typed, then the type checking is done when the program is compiled. Rust is strongly and statically typed with support for type inference and generics. When a language has support for type inference it means that the types does not have to be annotated and can be left out to be determined by the compiler at compile time. In Rust, the types of local variables and closures can be inferred, but the types of functions and structures must be annotated. Generics allows the types of functions and structures to be specified later in each occurrence. Listing 2 shows an example where the type of the closure `sum` is inferred by the compiler. Variables in Rust cannot be null, they must always have a value. There is, though, the generic `Option` type which makes it possible for a variable to conditionally have a value.

```
fn main() {
  let sum = |a, b| a + b;
  print!("{}", sum(1, 2));
}
```

Listing 2: A closure with types inferred

2.2.4 Ownership and borrowing

When we encounter problems that require data structures with a size that may grow or shrink, or whose size is not known at compile time, programming languages use a heap to manage the allocation of memory that is required to hold the data. Some languages such as C require the programmer to explicitly specify when and how much memory must be allocated and deallocated. This is not always trivial for the programmer and if it is used wrong it may cause the data in the memory to be corrupted.

Some language supports automatic memory management at runtime by discovering unreachable values through either a graph of references or by simply storing a reference counter for each value. These garbage collected languages does not require the programmer to explicitly manage the memory, but they might have a program that runs concurrently with the compiled program and therefore having, in some cases, a negative impact on execution time and memory usage compared to a program written in C.

Rust approaches memory management differently and we will now take a look at the concepts of ownership and borrowing in Rust.

In Rust, all values are owned by a single variable which is called its *owner* [9]. Once the owner goes out of the scope the value can be discarded. This allows for dynamic memory allocation to be determined by the compiler. The compiler will insert the required allocation and deallocation calls where it is required. The ownership of a value can be transferred or borrowed to another variable to allow data to be shared or reused. The ownership is handled by the compiler statically at compile time. When it comes to

references, a value can either have one mutable reference or any number of immutable references.

Listing 3 shows an example where the value of variable s is first borrowed, then moved. After moving the value, the variable `s` is no longer valid.

```
struct Foo;
fn main() {
  let s = Foo;
  {
    let borrow = &s; // borrow s
  } // the borrow goes out of scope
  let consume = s; // move s
  // s is no longer valid here
}
```

Listing 3: Borrow and move

2.2.5 Packages, Crates and Modules

A *crate* is either an executable binary or a library. A *package* contains one or more crates and is described in the *Crate.toml* file. A crate has a *root module*, and it may contain sub modules. Modules can either be inlined in a file or declared outside the file.

Listing 4 shows an example where the module `inlined` is inlined in `main.rs`, and the module `from_file` is located in `from_file.rs`.

```
fn main() {
  inlined::foo();
  from_file::bar();
}
mod inlined {
  fn foo() {}
}
mod from_file;
```

(a)

```
fn bar() {
}
```

(b)

Listing 4: Module example. (a) shows `main.rs`. (b) shows `from_file.rs`

2.2.6 Cargo

Cargo is the package manager of Rust. It resolves the dependencies by fetching and building packages and it sends the correct parameters to the compiler (rustc) when a package is built. When a package contains multiple crates, running `cargo build` will invoke the compiler once for each crate with the appropriate parameters. Cargo supports automated tests either configured as unit or integration tests. A test in Rust is a function marked

with the `#[test]` attribute. Integration tests are put in a folder named `test` on the top level and unit tests can be placed anywhere in the source tree.

2.2.7 Declarative Macros

$$\text{macro\_rules!} \quad \underbrace{\text{foo}}_{\text{Macro name}} \quad \{\underbrace{(\$v : ty)}_{matcher} => \underbrace{\{bar();\}}_{transcriber}\}$$

Rust has support for macros. When a macro is invoked, the invocation will be replaced with code from the declaration of the macro. The macro is expanded as a part of the parsing. First, the source code is parsed into an AST containing macro invocations. This AST may not be a legal Rust program yet, as items declared in macros are not yet placed in the AST. Then, macro expansion is applied to this AST and we get the resulting AST which is ready for the subsequent steps, such as name resolution and type checking.

Items, such as functions or structs, declared inside a macro are also visible outside the macro. Local variables however, are not visible outside the macro. Name resolution is done at the site of the invocation, which means that paths are sensitive to the context where it is invoked. Declarative macros may be invoked at item, statement or expression level.

Declarative macros is one of the two types of macros that are supported in Rust. The macro definition consists of a name and one or more arms. Each arm has a matcher and a transcriber part. When the macro is invoked, each of these arms will be matched against the tokens occurring in the input of the macro invocation and the first match is used. The matcher consists of a list of tokens, and each of the tokens may be captured in a variable that can be used in the transcriber. Each token may also be restricted to match only against a specific token type, or it can be of the more general 'token tree' type.

2.2.8 Procedural Macros

Procedural macros are another type of macro available in Rust. They are implemented as functions accepting a TokenStream and returning a TokenStream. They must be compiled to binaries and linked to where they are used, so the definition will not be in the same crate as the invocation.

As they are implemented by any valid Rust function, and not the production rules we saw in the declarative macros, it is in general undecidable to predict the outcome of changing inputs to the macro invocation. However, as we will later see, some of the builtin procedural macros follows a pattern to some extent, so that we can make some expectations when changing the input.

Function-like Procedural Macros These macros have the signature `TokenStream -> TokenStream` and can only occur at module level. They can produce any valid Rust code at that level, for instance new or shadowing item declarations and may refer to items that are valid at the call site.

Derive Procedural Macros Derive procedural macros are used on item declarations. They receive the token stream of the item declaration as input and return a new token stream that replaces the item declaration. There are several builtin derive macros in the standard library. These provide standard implementations of `Traits` for a `struct` derived from its members. An example of this is the `Clone` macro, which provides an implementation of the `Clone trait` for a `struct`. Listing 5 shows the code before and after macro expansion for the `Clone` macro. On the left side, line 2 is a token stream which is the input to the macro. On the right side, we show the code after macro expansion.

Before expansion

```
1 #[derive(Clone)]
2 struct S(u32);
```

After expansion

```
1 struct S(u32);
2 #[automatically_derived]
3 impl Clone for S {
4   #[inline]
5   fn clone(&self) -> S {
6     match *self {
7       S(ref __s0) =>
8     S(Clone::clone(&(*__s0)))
9 }}}
```

Listing 5: `derive(Clone)` before and after expansion

Attribute Procedural Macros Attribute macros have the signature `(TokenStream, TokenStream)-> TokenStream` and take in the item and its attributes it is used on as input and return zero or more items which is the item is replaced with.

2.3 Refactoring support for Rust

Rust is a relatively new language first announced in 2010 and does not have the same refactoring support as more established languages like Java, C#, or JavaScript. Here we will go through some of the tools that supports refactoring or similar features such as formatting and linting for Rust.

2.3.1 Rust language server

RLS (Rust Language Server) is an LSP (Language Server Protocol)-server that gives support to IDEs to navigate, compile, lint, format, and refactor code. LSP is a protocol that several different text-editors and IDEs supports, making it easier to give access to a tool from the text-editor or IDE. The IDE can use RLS as an interface where RLS delegates the requests to the compiler and a code completion utility called Racer.

2.3.2 Clippy

Clippy [6] is a tool that gives linting and fixes. Linting is a static analysis of a program that checks for common errors or improvements. The term linting originates from the lint tool [17] from 1978 that "examines C source programs, detecting a number of bugs and obscurities".

The tool will suggest refactoring for some of the lints, but it does not check the safety of the code change, thus semantics may be changed. Clippy can also warn when the cognitive complexity [27] of a function is over a given threshold. As of 2020, Clippy has 390 lints [1].

2.3.3 Rustfmt

Rustfmt [7] is a tool that formats the codebase automatically after a given style guide. The tool should be semantics preserving but not necessarily syntax preserving. An example can be to combine multiple import statements to a single import statement. Rustfmt operates on the AST (abstract syntax tree) instead of a token stream.

2.3.4 IntelliJ Rust

IntelliJ Rust [16] is a plugin for the IntelliJ IDEA. It is, at the time of writing, marked as work-in-progress where bugs and missing features should be expected. It does provide some refactorings as shown in Figure 2.1, such as extract variable and extract method. The plugin is written in Kotlin and open sourced so it is possible to extend the plugin with additional features.

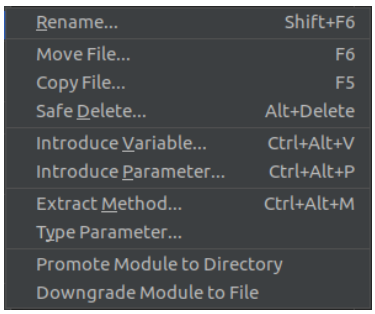

Figure 2.1: Some of the refactorings supported by IntelliJ Rust

2.3.5 Rust Analyzer

Rust Analyzer [5] is an upcoming replacement of RLS. It consists of a server and clients for Emacs and Visual Studio Code with communications going over the Language Server Protocol. It has its own parser and type inference implementation and does therefore not rely on the compiler, like RLS does. It has support for several micro-refactorings, and a list is shown in Figure 2.2. At the time of writing, it does not have an implementation of the Extract Method refactoring.

[1] https://rust-lang.github.io/rust-clippy/master/

```
add_custom_impl                 invert_if
add_derive                      merge_imports
add_explicit_type               merge_match_arms
add_from_impl_for_enum          move_bounds_to_where_clause
add_function                    move_arm_cond_to_match_guard
add_impl                        move_guard_to_arm_body
add_new                         add_hash
add_turbo_fish                  make_raw_string
apply_demorgan                  make_usual_string
auto_import                     remove_hash
change_return_type_to_result    remove_dbg
change_visibility               remove_mut
convert_to_guarded_return       reorder_fields
fill_match_arms                 replace_if_let_with_match
fix_visibility                  replace_let_with_if_let
flip_binexpr                    replace_qualified_name_with_use
flip_comma                      replace_unwrap_with_match
flip_trait_bound                split_import
inline_local_variable           unwrap_block
introduce_named_lifetime        add_missing_impl_members
introduce_variable              add_missing_default_members
```

Figure 2.2: Some of the refactorings supported by Rust Analyzer

Before refactoring

```
let mut x = 0;
x = x + 1;
x = x + 1;
```

After refactoring

```
let mut x = 0;
let y = x + 1;
x = y;
x = y;
```

Listing 6: Extract Local Variable in IntelliJ

2.4 Challenges when refactoring Rust

In this section we will take a look at some general challenges when it comes to refactoring Rust programs.

2.4.1 Change of semantics

An incorrect implementation of a refactoring may change the semantics of a program. In Listing 6 we show an example of such change where the runtime semantics is changed. Here we choose to store the expression $x + 1$ in a variable y, and replace occurrences of the expression with y. Before refactoring x ends up with the value 2. After refactoring x ends up with the value 1.

2.4.2 Ownership model

A more unique challenge to Rust is the ownership model. For instance the ownership model makes the lifetime of variables more explicit and it also restricts aliasing.

The extract local variable refactoring for Java in Eclipse can be unsafe in some cases [10] where it introduces a semantic change. For Rust, we found an example where a similar refactoring was rejected by the compiler because of violations of the rules of the ownership model.

Before change

```
1  struct C { x: X }
2  impl C {
3    fn m(&mut self) {
4      self.x = X{}; }
5    fn n(&mut self) {
6      self.x.n();
7      self.m();
8      self.x.n();
9    }
10 }
```

After change

```
1  impl C {
2    fn n(&mut self) {
3      let tmp = &self.x;
4      tmp.n();
5      self.m();
6      tmp.n();
7    }
8  }
```

Listing 7: Extract Local Variable in IntelliJ

```
error[E0502]: cannot borrow `*self` as mutable because it is
↪  also borrowed as immutable
```

Listing 8: The error message after the code change in Listing 7

Here the `self.x` in line 6 and 8 in Listing 7 is extracted to a local variable inside `fn n`. The result is shown to the right. The problem with applying this code change in Rust is that the `self` variable is borrowed from line 3 until tmp goes out of scope after line 6. But between line 3 and line 6 `self` is also borrowed as mutable. The error message is shown in Listing 8.

2.4.3 Multiple root modules in a single package

In Rust a single package can have multiple binary outputs, where each of them has its own root module. These root modules can use the same or different submodules, resulting in that a module defined in a single file can have different parents depending on how it is included in other modules. There will be one top module for each crate, and one for the unit tests of the crate. There will also be a top module for each integration test in the package. This means that each file occurring in a package may be included in one or more ASTs depending on how many times it is used as a sub module.

An example is shown in Figure 2.3. Here the file contents is shown inside boxes, and arrows indicate use of a file as a submodule. This example has two crates `lib` and `main` with root modules shown in `lib.rs` and `main.rs`. `lib` has the modules `submodule1` and `submodule2`, and `main` has one module `submodule2`. This means that the content of `submodule1.rs` is only included in the AST of `lib`. The content of `submodule2.rs` is included in the ASTs of `lib` and `main`, and the path `super::foo` in `submodule2.rs` binds differently in the two ASTs as `super` refers to the parent module. This means that the type of the local variable `x` will be `String` for `lib` and `u32` for `main`.

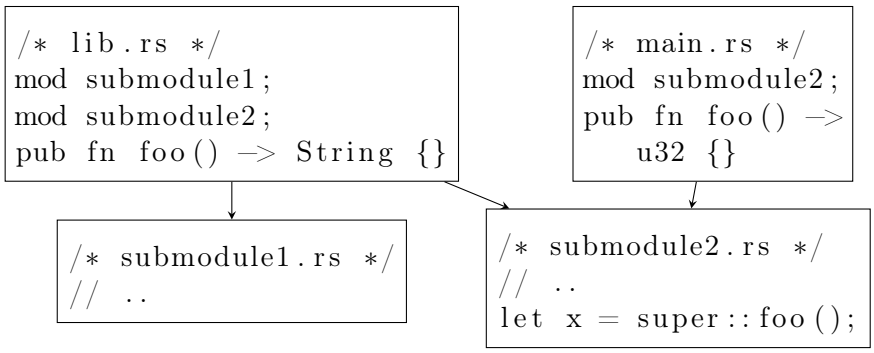

Figure 2.3: Multiple root modules

```
macro_rules! foo {
  ($a: ident) => { }
  ($a: expr) => { }
}
```

(a)

```
macro_rules! foo {
  (* $a: expr) => { }
  ($a: expr) => { }
}
```

(b)

```
macro_rules! foo {
  ($a: ident, $b: tt) => { $a!($b); }
}
```

(c)

Listing 9: Declarative macros

2.4.4 Declarative macros

Depending on the definition of the macro, there are different challenges when modifying source code that contains macro invocations. As the input of the macro invocation consists of tokens, changing the input may change which arm is used to output the content of the macro. An example is shown in Listing 9. In (a) the first arm is used if the input is an `ident` token and the second arm is used if the token is of type `expr`. In (b), the used arm changes when adding or removing the `DereferenceOperator` expression in the front of an expression. In (c) the macro calls another macro, which depends on the content of the input.

Declarative macros can also be *recursive*, defining new macros when it is invoked. We will not consider this in this thesis. The arguments of macro invocations can be `Blocks`, so the arguments of an invocation can be the target of an Extract Method refactoring, but we will not consider this.

Changing the expanded code by modifying the input

If we want to change the expanded code of a macro invocation, we can do so by modifying the inputs of the invocation. This can, as shown, be challenging when there are different arms in the declaration of the macro, or if the declaration is not available.

```
1 let foo: i32 = 0;
2 max!(foo, 0);
```

(a)

```
1 let foo: i32 = 0;
2 if foo > 0 { foo }
3 else { 0 }
```

(b)

```
1 let foo: &i32 = &0;
2 max!(*foo, 0);
```

(c)

```
1 let foo: &i32 = &0;
2 if *foo > 0 { *foo }
3 else { 0 }
```

(d)

Listing 10: Macro expansion example

One way of verifying that a macro expansion changes as expected is to capture the expanded code before and after change and asserting that the expanded code changed as expected.

In Listing 10 we show an example of where the argument is modified. The unmodified program is shown in (a) and its expanded code is shown in (b). If we consider the change in (c), where we change the type of `foo` from `i32` to `&i32`, and the first argument of the macro invocation from `foo` to `*foo`, we should expect the expanded code to be the content of (d).

Changing the expanded code by inlining the macro

Another way of changing code that contains macro invocations is by first replacing the invocation with the expanded code. Local variables declared inside a macro are only visible inside the macro rule. A macro can however expand to multiple nodes at the invocation level, so a macro expanded at statement level can result in multiple statements at the same level. If the macro contains declarations of variables then these cannot shadow declarations outside the macro. So when inlining a macro we need to be aware of this, possibly renaming variables so that bindings are not changed.

An example of this is shown in Listing 11. Here (a) is the code before expansion which contains a macro declaration `foo` which has only a single local variable declaration of `bar` which is initialized to 1. At line 6, a local variable `bar` is also declared, but initialized to 2. At line 8, `bar` binds to the declaration at line 2. The expanded code is shown in (b). In (c) we see an incorrect example where the macro is inlined without performing renaming first, which results in `bar` at line 5 being bound to the declaration at line 3.

Some of the builtin macros such as `print!` results in function calls that are not allowed to be used by the programmer directly. If we expand these macros we get a compile time error, which means that there are macros that cannot be inlined.

Changing the expanded code by modifying the definition

We can also change the definition of the macro. If we choose to this, we should must also assure that each invocation of the macro does not change

```
1 macro_rules! foo {
2   () => {
3     let bar = 1;
4 }}
5 fn main() {
6   let bar = 2;
7   foo!();
8   baz(bar);
9 }
```

(a)

```
1 fn main() {
2   let bar = 2;
3   let bar = 1;
4   // bar = 2
5   baz(bar);
6 }
```

(b)

```
1 fn main() {
2   let bar = 2;
3   let bar = 1;
4   // bar = 1
5   baz(bar);
6 }
```

(c)

Listing 11: Macro inlining example.

the semantics of the program. The macro must also be defined in the same crate.

Handling declarative macros

In this thesis we will handle macro invocations by running the visitor after macro expansion. When needed, we will modify the arguments of macro invocations, but not the definitions.

2.4.5 Procedural macros

In this thesis we will only consider the Derive Procedural Macros, as these are commonly used on `struct` declarations. It will be handled in the implementation of Box Field by not visiting methods that are generated by the derive macros `Clone`, `Default`, `Debug`, `Eq`, `Hash`, `Ord`, `PartialEq` and `PartialOrd` as the derived methods for these are identical for fields with type `Box<T>` and `T`.

2.4.6 Attributes and conditional compilation

The `#[cfg]` attribute may be used on statements and item declarations. It will conditionally remove the element that it is used on depending on compilation flags, resulting in potential differences in bindings, control flow and data flow between different compilation units. An example is shown in Listing 12 where the `ItemDeclaration` at line 3 is conditionally included on the compile flag `foo`, which leads to the item binding on line 4 being conditionally resolved based on this. We will not be considering attributes and conditional compilation further in the algorithms or in the implementation.

Before refactoring

```
fn main() {
  #[cfg(foo)]
  use bar::bar;
  bar();
}
```

Listing 12: `cfg` attribute example.

Chapter 3

Extract Method

Extract Method [13] or convert_code_segment_to_function [26] is a classical refactoring that creates a new function declaration from a part of the body of an existing function, and replaces that part with a call to the new function. Schäfer presents Extract Method [28] as a composition of micro refactorings, using a framework to lock control flow and bindings.

In this chapter we will present several micro refactorings that can be composed into the Extract Method refactoring based on the compositions presented by Schäfer.

First, we give an overview of the compositions and different challenges that we handle. Then, each of the micro refactoring is presented in its own section. At the end, a table containing helper functions used in the micro refactorings is shown.

3.1 Overview

In this chapter we will use B for `Block`, C for `Closure`, D for `ItemDeclaration`, E for `Expression`, F and G for `FunctionDeclaration`, M for `Mod`, I for `ImplBlock`, N for `Identifier`, P for `PathExpressions` and S for `Statement`. In the algorithms we will use **pre** to indicate a precondition that must hold, and `*?`, `&?` and `mut?` will be used for conditional `*`, `&` and `mut`.

We will now present how we compose the Extract Method refactoring for Rust, and which challenges they must be aware of.

3.1.1 Composition of micro refactorings

We will now give a high level overview of the composition, before going into detail. The micro refactorings that we use to compose our Extract Method for Rust is shown in a side-by-side list with Schäfers composition. We had to modify some of the micro refactorings, and add some new.

Extract Method for Java by Schäfer.

1. Extract Block
2. Introduce Anonymous Method
3. Close Over Variables
4. Eliminate Reference Parameters
5. Lift Anonymous Method

Extract Method for Rust

1. Pull Up Item Declarations
2. Extract Block
3. Introduce Anonymous Closure
4. Close Over Variables
5. Convert Anonymous Closure to Function
6. Lift Item Declarations
7. Lift Function Declaration

In Rust, `ItemDeclarations` may occur as `Statements` inside a `Block`. To preserve item bindings we will develop *Pull Up Item Declarations* which takes a contiguous sequence of `Statements` as input, reorders these, so that `ItemDeclarations` appear at the top, and returns the modified sequence of `Statements`.

Extract Block takes a contiguous sequence of `Statements` as input and introduces a new `Block` around these. To preserve bindings of variables, a new `let`-declaration is added right before the new block, containing the identifiers of each variable that is declared inside the selection and used after the selection.

Instead of Introduce Anonymous Method, we will develop *Introduce Anonymous Closure*, as closures are a special type of function in Rust which captures the body where it is declared. Introduce Anonymous Closure takes a `Block` as input and returns a `match Expression` containing the new closure. Here control flow `Expressions` are handled by using special `enum` codes.

Close Over Variables operates on a closure and eliminates references to local variables declared outside the closure. Here, annotations for *borrow* and *mutable borrow* (`&` and `&mut`) need to be added to parameters and arguments depending on how the variable is used inside the closure.

Eliminate Reference Parameters is not needed as parameters may be passed by reference in Rust.

Local functions are supported in Rust, and the closure is therefore converted to a local function using *Convert Anonymous Closure to Function*. This refactoring returns a `BlockExpression` containing the new function declaration as the first `Statement` and the identifier of the new function as the `Expression`.

Before moving the new function declaration upwards in the AST, we need to ensure that item bindings will be preserved when moving it. *Lift Item Declarations* ensures that item bindings inside the function declaration, both signature and body, is resolved to `ItemDeclarations` occurring at the place where the function should be moved to, or higher in the AST.

As the final step, the function declaration is lifted upwards in the AST to the closest `Impl`-block with *Lift Function Declaration*. Any invocations

to the function must also be updated to reflect the new placement of the function.

3.1.2 Challenges

Grammars

According to Fowlers book [13], Extract Method turns a code fragment into a method. The code fragments we will consider here are subsequences of blocks or an expression. A block in Rust consists of zero or more statements, and zero or one expression at the end[1]. A subsequence of a block will therefore be zero or more consecutive statements and optionally the block at the end. Blocks are value expressions and evaluate the last expression, or to the unit value `()` if there is no expression. This also means that blocks have a type which is the type of the expression, or the unit type.

Name binding preservation

Name binding is the process of binding names to variables and items. This may accidentally change for instance when moving statements into blocks. We will go into detail in with Pull Up Item Declaration in Section 3.2, Extract Block in Section 3.3, Lift Item Declarations in Section 3.7 and Lift Function Declaration in Section 3.8.

In Listing 13 we show an example where a local variable binds to a different declaration after an incorrect refactoring. It is an example where cutting & pasting a set of statements in a single block accidentally changes the bindings. On the left side the identifier `i` is bound to the declaration at line 3 before refactoring. After changing, it is bound to line 2. In Listing 14 we show a situation where a call expression binds to a different function after refactoring. Before refactoring, `bar()` at line 3 binds to the function declared at line 4, inside `main()`. After refactoring, `bar()` at line 7 binds to the declaration at line 1. Listing 14 was done using the Extract Method refactoring in IntelliJ Rust.

Before change

```
1 fn main() {
2   let i = 0;
3   let i = 1;
4   foo(i);
5   bar(i);
6 }
```

After change

```
1 fn main() {
2   let i = 0;
3   baz();
4   bar(i);
5 }
6 fn baz() {
7   let i = 1;
8   foo(i);
9 }
```

Listing 13: A cut & paste refactoring where variable binding changes.

[1] Grammar: https://doc.rust-lang.org/reference/expressions/block-expr.html

Before change

```
fn bar() {panic!("");}
fn main() {
  bar();
  fn bar() {}
}
```

After change

```
fn bar() {panic!("");}
fn main() {
  baz();
  fn bar() {}
}
fn baz() {
  bar();
}
```

Listing 14: Incorrect Extract Method where an item binding changes.

Control flow preservation

Control flow is the order in which statements are executed. We will look at control flow in Introduce Closure in Section 3.4. IntelliJ Rust will do the refactoring shown in Listing 15 wrong. Here the control flow changes after refactoring. Before refactoring, the `parse` method call at line 4 will return an error. The Error Propagation Operator "?" at the end of line 4 will cause `foo()` to stop executing and return that error, so that `bar()` at line 7 is not invoked. After refactoring, the value of `baz()` at line 2 is not checked, `foo()` continues to execute, so that `bar()` at line 3 is called.

Before change

```
use std::num::ParseIntError;
fn foo() -> Result<i32,
↪  ParseIntError> {
  let j = {
    let i: i32 = "".parse()?;
    Ok(2 * i)
  };
  bar();
  j
}
```

After change

```
fn foo() -> Result<i32,
↪  ParseIntError> {
  let j = baz();
  bar();
  j
}
fn baz() -> Result<i32,
↪  ParseIntError> {
  let i: i32 = "".parse()?;
  Ok(2 * i)
}
```

Listing 15: Incorrect Extract Method where the control flow is changed.

Data flow preservation

Preserving data flow is, according to Schäfer: all variables should have the same reaching definitions throughout the refactoring. The reaching definitions of a variable at a point in a program are all the definitions of that variable which may reach the point. We will come back to this when passing arguments by value or reference in Close Over Variables in Section 3.5.

As shown in Listing 16 the data flow changes as the reaching definition

of `i` at line 4 before refactoring was `bar(i)` from line 3, but after refactoring it is the constant `0` at line 2. This happens because `i` is passed into `baz` by value, and not by reference. So the assignment at line 7 has no effect. This refactoring was done using IntelliJ Rust.

Before change

```
1 fn bar(_: i32) -> i32
  ↪ {1}
2 fn foo() -> i32 {
3   let mut i = 0;
4   i = bar(i);
5   i
6 }
```

After change

```
1 fn main() {
2   let mut i = 0;
3   baz(i);
4   i
5 }
6 fn baz(mut i: i32) {
7   i = bar(i);
8 }
```

Listing 16: Incorrect example of Extract Method with data flow changed.

Ownership model

A more unique challenge with Rust is the ownership model. Changes to the ownership model can alter the lifetimes of variables. These changes are usually not visible to the user, unless the compiler considers the program not legal and rejects it. This is discussed further with Extract Block in Section 3.3 and Close Over Variables in Section 3.5.

With Extract Method, we introduce a new function declaration wrapping a set of statements. These statements then get a new scope, possibly reducing their lifetime. An application of Extract Method in IntelliJ Rust is shown in Listing 17. Here we try to return a borrow from a function, where the value does not outlive the function, resulting in a violation of the ownership model. The code after refactoring will not compile with the following error message: 'error[E0106]: missing lifetime specifier.'.

Before change

```
1 fn main() {
2   let a: &i32 = &0;
3   a;
4 }
```

After change

```
1 fn main() {
2   let a = foo();
3   a;
4 }
5 fn foo() -> &i32 {
6   let a: &i32 = &0;
7   a
8 }
```

Listing 17: Extract Method, where the the ownership model is violated.

3.2 Pull Up Item Declarations

Now we will introduce the Pull Up Item Declarations micro refactoring. Its purpose is to ensure that name bindings for items are preserved when applying Extract Block. The bindings will be preserved by reordering a selection of statements, so that the statements which are item declarations appear before the selection.

In Listing 18 both call expressions at lines 2 and 4 to the left bind to the function declaration at line 3. To the right, lines 3 and 4 bind to the declaration at line 2.

Before refactoring

```
1 fn main() {
2   foo();
3   fn foo() {}
4   foo();
5 }
```

After refactoring

```
1 fn main() {
2   fn foo() {}
3   foo();
4   foo();
5 }
```

Listing 18: Pull Up Item Declaration example.

Definition

This refactoring takes zero or more contiguous `Statements` in the same `Block` as input and returns a set of `Statements` where the `ItemDeclarations` are placed on top.

Correctness

For this refactoring we will use Rusts restrictions on ambiguity when it comes to `ItemDeclarations`. We know that if the program is already legal, then an identifier may only be declared at most once at the same block level. `ItemDeclarations` are visible throughout the `Block`, both before and after where they appear, which means they can be freely moved around the `Block` as long as they stay at the same block level.

Identifiers are strings of letters and digits. They can be used to denote fields of expressions, e.g. `foo().bar` refers to the field `bar` on the value returned by `foo`. They can also be used to denote paths to items such as functions or types, e.g. `std::bool` refers to the boolean type.

Scopes define the visibility of items and variables, and each `Block` has a scope for `ItemDeclarations` and one for variable declarations (`let` -declarations). Items are in scope inside the `Block` they are declared in and it is required that they are resolved without ambiguity, which means that an identifier may only be declared once as an item in the same `Block`. Variables are in scope from the next `Statement` until the end of the `Block` and they may be declared with the same identifier multiple times at the same block level, which results in the latter shadowing the previous declaration as we have seen in Listing 13.

Algorithm

Algorithm 1 finds the closest parent `Block` of a sequence of `Statements`, then reorders these `Statements` in the `Block` and returns the modified `Block`. We do not need to traverse `Blocks` found here, as we are only interested in the `Statements` that are directly `ItemDeclarations`.

Algorithm 1 Pull Up Item Declarations

Input: S – A contiguous sequence of `Statements` within a single `Block`
Output: The modified `Block`

$B \leftarrow$ closestParentBlockOf(S)
for all $S' \in$ itemDeclarationsIn(S) **do**
  delete S' from B;
  insert S' in B before S;
end for
return B

3.3 Extract Block

The purpose of the Extract Block refactoring is to wrap a `Block` around `Statements` or an `Expression` so that the new `Block` can later be converted to an anonymous closure. An example is shown in Listing 19, where `sum` is passed out of the new `Block`.

Before refactoring

```
1 let sum = i + j;
2 print!("{}", sum);
3 return sum;
```

After refactoring

```
1 let sum =
2 {
3   let sum = i + j;
4   print!("{}", sum);
5   sum
6 };
7 return sum;
```

Listing 19: Extract Block example.

Definition

In Rust a `Block` consists of zero or more `Statements` and zero or one `Expression` at the end. Blocks are `Expressions` and have a type. If the `Block` has an `Expression` at the end, then the `Block` has the same type as this `Expression`. If not, then it is of the `Unit` type. This means that values may be passed out of a `Block` and assigned to in the parent `Block`. With

the `Tuple` type, multiple values may be passed without declaring the type of the `Tuple`.

This refactoring should take zero or more continuous `Statements` in a `Block` and optionally the `Expression` at the end, and extract these into a new `Block`, without changing the semantics of the program.

Correctness

Name bindings of variables A variable is visible from the next statement after they are declared, until they go out of scope. In Rust, multiple variable declarations with the same identifier may appear in the same `Block`, which results in the latter shadowing the previous declaration. Here we will say that the name binding is preserved iff every name still refers to the same declaration after refactoring, as before refactoring.

The `match` expression, and its syntactic sugar constructs, `while`, `for` and `if`, may introduce new declarations in parts of their syntax. These declarations however, are only visible inside themselves, and can therefore not change the name bindings at the block level they occur on.

The `let` statement, which declares a set of new variables, may shadow previous declarations until the end of the `Block` where it occurs. For `let` bindings occurring in the extracted selection and used in the post section we will, for each of the identifiers, add a new declaration at immediately before the new `Block`, assign it to the new `Block`, and also add the identifiers as the value of new `Block`, letting it escape the `Block`. For this we can use the `Tuple` type of Rust, letting us pass out multiple values of a `Block` without declaring a new type for the `Tuple`.

With this solution, we must however take special care of the ownership model. If a value is declared in a function, we cannot also return the borrow of that value.

Name bindings of items As a precondition for this refactoring we should require that item declarations occurring in the selection, should not be used outside the selection. If this is the case, then Pull Up Item Declarations should be applied first.

Ownership According to the Rust language definition we have the following definition for owners: Each value has a variable called the owner. There can only be one owner at a time. When the owner goes out of scope, the value will be dropped, meaning that its resources are freed. The ownership is verified at compile time by the borrow checker, meaning that the rules must be decidable at compile time. The ownership model is a separate type system which runs on the MIR (Mid-level Intermediate Representation) with a grammar consisting of basic blocks, locals, place- and value expressions. For this refactoring we will have a precondition that variables declared inside the selection and used outside, should not borrow from the inside of the selection.

Before refactoring

```
1  // 𝕊pre
2  // 𝕊 start
3  let x0
4  // ..
5  let xn
6  // 𝕊 end
7  // 𝕊post start
8  x0
9  // ..
10 xn
11 // 𝕊post end
```

(a)

After refactoring

```
1  // 𝕊pre
2  let (x0, .., xn) =
3  {
4    // 𝕊 start
5    let x0
6    // ..
7    let xn
8    // 𝕊 end
9    (x0, .., xn)
10 };
11 // 𝕊post start
12 x0
13 // ..
14 xn
15 // 𝕊post end
```

(b)

Listing 20: Extract block example, step by step.

Algorithm

Algorithm 2 first collects the identifiers of all locals which are declared inside the selection, but used outside the selection. Then, it adds a new `Block` around the selection, adds new local declarations for the identifiers before the new `Block` and passes the locals out of the `Block`.

An illustrated example is shown in Listing 20. The input to the refactoring is shown in (a) with the selection highlighted. The selection contains declarations x_0 to x_n which are used later in $\mathbb{S}_{post}$. In (b), one or more declarations are used later, and a new tuple declaration is added at line 2 which is initialized to the value of the new `Block` spanning from line 3 to 10. The output in (b) is a `let` declaration which is a `Statement`. The case where no declarations are used later is not shown for simplicity, as it only requires wrapping a `Block` around the selection, and no `let` declaration needs to be added.

A case which is not supported is shown in Listing 21. Here both `t` and a borrow of `t` is used later, so we have to move `t` and the borrow out of the `Block`, but *rustc* does not allow this. The code in (b) does not compile, as we get two error messages, “error[E0597]: ‘t‘ does not live long enough” and “error[E0505]: cannot move out of ‘t‘ because it is borrowed”.

Algorithm 2 Extract block

Input: S – A contiguous sequence of `Statements`, optionally with an `Expression`
Output: A `let`-declaration containing the new `Block`
Variables: X, M – Set of identifiers

1: $X, M \leftarrow \emptyset$
2: $B \leftarrow$ closestParentBlockOf(S)
3: **for all** $L \in$ localsDeclaredIn(S) **do**
4: **if** escapesAsBorrow(S, L) **then**
5: fail
6: **end if**
7: **if** usedMutableLater(S, L) **then**
8: $M \leftarrow M \cup L$
9: $X \leftarrow X \cup L$
10: **else if** usedLater(S, L) **then**
11: $X \leftarrow X \cup L$
12: **end if**
13: **end for**
14: **if** $X \neq \emptyset$ **then**
15: **return** 'let ($m_0\ x_0,\ \ldots,\ m_n\ x_n$) = { S ($x_0,\ \ldots,\ x_n$) };'
 where $x_i \in X$
 $m_i = x_i \in M$ **?** 'mut'
16: **else**
17: **return** '{ S }'
18: **end if**

Before refactoring

```
fn main() {
  let t = T();
  let borrow = &t;
  borrow;
  t;
}
```

(a)

After refactoring

```
fn main() {
  let (t, borrow) = {
    let t = T();
    let borrow = &t;
    (t, borrow)
  };
  borrow;
  t;
}
```

(b)

Listing 21: Extract Block ownership problem

3.4 Introduce Anonymous Closure

The purpose of this refactoring is to create the closure that will later be converted to a function. An example is shown in Listing 22. Here we see the `Block` at the left containing a call to the function `foo`. After refactoring, an empty list of parameters, `||`, is added before the `Block` at line 1 which converts the block to a closure. Parenthesis are added around the `Block`, and at line 3, an invocation is added, `()`, with no arguments.

Before refactoring

```
1 { foo() };
```

After refactoring

```
1 (||
2   { foo() })
3   ();
```

Listing 22: Introduce Anonymous Closure example.

Definition

The Introduce Closure Refactoring will take a `Block` as input and return a `CallExpression` containing the new closure definition as the first `Expression` part and with zero call parameters.

Correctness

Control flow Control flow preservation according to Schäfer: All statements in affected methods should maintain their control flow-predecessors and successors throughout the refactoring. Control flow successors of a statement are statements that may be executed directly after that statement. Predecessors of a statement S are all statements that have S as a successor.

Challenges here are the `break`, `continue`, `return` and the Error Propagation Operator ("?") expressions as they alter the natural control flow of a `Block`. The Error Propagation Operator is only allowed inside functions with a return type of `Result` or `Option` and conditionally stops the execution of the function if the expression evaluates to `Result::Err` or `Option::None`.

`break` and `continue` expressions which point outside the `Block` must be handled, as they are not allowed to point outside of a closure. `return` expressions must also be handled as this would then resolve to the new closure, and not to the outer context of the `Block`, thus changing the control flow. To preserve the control flow for `continue`, `break` and `return` expressions we must identify any of these occurring inside the extract section, while being bound outside. `break` and `continue` expressions may also have a label, but we will not handle labels in this algorithm. `return` expressions also have an expression inside them.

If the `Block` contains the Error Propagation Operator "?", then this affects the type of the closure and all occurrences of the "?" expression and the type of the `Block` must be made equal. Another option is to replace the "?"

expressions with conditional returns before applying this refactoring. We will not handle occurrences of "?" in this algorithm.

Grammar We will allow two types of input to this refactoring. The refactoring will be defined on a single `Block`, but if the input is an assignment expression where the right hand side is a `Block`, then we will use the `Block` at the right hand side.

Algorithm

In Algorithm 3 we first collect all control-flow expressions pointing outside of the `Block`. Then we convert the `Block` to a `CallExpression` containing a `ClosureExpression` with no arguments. If there are any control-flow expressions that points outside of the new closure, we will also wrap a match expression around the new closure.

Listing 23 shows how the algorithm works. Here, the `break`, `continue` and `return` expressions at lines 5, 7 and 9 have their successor state outside the `Block`. After refactoring, they are replaced with a `return` expression, where the expression is an enum type with the contained expression inside as a value within the enum. Outside the new `CallExpression`, a `match` expression is added, which handles the control flow at lines 14 to 17.

Algorithm 3 Introduce Anonymous Closure

Input: B – A `Block`
Output: A `match` expression containing the new `ClosureExpression` inside a `CallExpression`

1: $E \leftarrow$ expressionOf(B)
2: **replace** E in B with 'Expr(E)'
3: **for all** $E_C \in$ cfExprWithNextOutside(B) **do**
4: $E'_C \leftarrow$ returnCodeOf(E_C)
5: **replace** E_C in B with 'return E'_C'
6: **end for**
7: **return** 'match (|| {
B
})() {
(Break(val)) => break val,
(Continue()) => continue,
(Return(val)) => return val,
(Expr(val)) => val
}'

Before refactoring

```
loop {
  // ..
  {
    // ..
    break E;
    // ..
    continue;
    // ..
    return E;
    // ..
    E
  }
  // ..
}
```

After refactoring

```
loop {
  // ..
  match (||
  {
    // ..
    return Break(E);
    // ..
    return Continue();
    // ..
    return Return(E);
    // ..
    Expr(E)
  })() {
    Break(val) => break val,
    Continue() => continue,
    Return(val) => return val,
    Expr(val) => val,
  }
  // ..
}
```

Listing 23: Introduce Anonymous Closure.

3.5 Close Over Variables

This refactoring works on a closure, and eliminates any references to local variables declared outside the closure. In Listing 24 an example is shown, where a parameter is added to the closure, and the local variable `i` is instead passed in as an argument at line 3. This will make it easier to convert the closure to a local function.

Before refactoring

```
(|| { foo(&mut i) })();
```

After refactoring

```
(|i: &mut i32| -> i32
  { foo(i) })
  (&mut i);
```

Listing 24: Close Over Variables example.

Definition

Close Over Variables takes a `CallExpression` as input where the `Expression` part is a `ClosureExpression`. The output is a `CallExpression` where all variable occurrences in the `ClosureExpression` that were resolved outside the closure definition are now resolved to the parameter list of the closure. The arguments of the `CallExpression` are updated to reflect the new parameter list.

```
let i = 0;
let j = (|| {
    &i
})();
```

(a) Before change

```
let i = 0;
let j = (|i: _| {
    i
})(&i);
```

(b) Allowed

```
let i = 0;
let j = (|i: &i32| {
    i
})(&i);
```

(c) Not allowed

Listing 25: Type annotation missing lifetime

```
let t = (0, 1);
(|| {
    t.0;
})();
```

(a) Before change

```
let t = (0, 1);
(|t: _| {
    t.0;
})(t);
```

(b) Not allowed

```
let t = (0, 1);
(|t: (_, _)| {
    t.0;
})(t);
```

(c) Allowed

Listing 26: Type missing `Tuple` annotation

Correctness

Inference of lifetimes and types Here we encounter a difficulty with the type inference of Rust. Inference of type annotated variables on closures is weaker than variables without annotation. An example is shown in Listing 25 where (a) and (b) are allowed, but (c) is rejected with the following message: 'error[E0495]: cannot infer an appropriate lifetime due to conflicting requirements'. This is due to the output being a borrow of the input, and the input should therefore outlive the output. Lifetime variables cannot be introduced on closures. The type of `i` is `i32` outside the closure for all examples. The type of `j` is `&i32`. Inside the closure the type of `i` is `&i32` for (b) and (c).

Another example where a partial type annotation is needed is shown in Listing 26. Here we have a tuple indexing expression occurring in the closure. In (b) we get the following error message: 'error[E0282]: type annotations needed'.

In this algorithm we will fully annotate the types of the parameters and the return type of the closure. This means that we will not support cases where lifetime annotation is required.

Data Flow To preserve data flow, we must correctly pass variables that are mutated *by reference*. We do this by prepending `&mut` to the type in the parameter list and by prepending `&mut` to the identifier in the argument list. If a local is *moved* in the closure, it must be passed in *by value*. If a local is *borrowed* in the closure, and used afterwards, we must pass it in *by reference*. This changes the type of the local in the closure from `T` to `&T`. When adding borrow, `&`, to the type of a parameter we must also update any occurrences of the variable inside the closure, to preserve the type of the expressions by wrapping a `DereferenceOperator` expression, `*`, around it.

Algorithm

Algorithm 4 shows the steps. Here the local variables used inside the closure and declared outside are collected. The type of use is also collected, and if it is not moved, then the variable is passed in by reference and any `PathExpressions` inside the closure resolving to it is updated to preserve the type by wrapping a `DereferenceOperator` expression around it. A general example is shown in Listing 27. Here we see that x_0 to x_n are added to the parameters and arguments of the closure, with optionally adding annotations for borrow and mutable.

Algorithm 4 Close Over Variables

Input: E – A `CallExpression`
Output: The modified `CallExpression`
Variables: M, N, X – Set of identifiers

1: $C \leftarrow$ expressionPartOf(E)
2: **pre** isClosureExpression(C)
3: $B \leftarrow$ bodyOf(C)
4: $M, N, X \leftarrow \emptyset$
5: $R \leftarrow$ inferredTypeOf(E)
6: **for all** $L \in$ localsUsedInAndDeclaredOutside(B) **do**
7: $X \leftarrow X \cup L$
8: **if** movedInside(B, L) **then**
9: $M \leftarrow M \cup L$
10: **continue**
11: **else if** usedMutableInside(B, L) **then**
12: $N \leftarrow M \cup L$
13: **end if**
14: **for all** $E \in$ pathExpressionsResolvedTo(B, L) **do**
15: **replace** E in B with $(*E)$
16: **end for**
17: **end for**
18: **return** '(|m_0 x_0: T_0, .., m_n x_n: T_n| -> R {
B
})(|m_0 x_0, .., m_n x_n|)'
where $x_i \in X$
$m_i = x_i \in M$ **?** ''
$x_i \in N$ **?** '&mut'
otherwise '&'

Before refactoring

```
let x0: T0
// ..
let xn: Tn
(|| {
  // ..
  x0
  // ..
  xn
})()
```

After refactoring

```
let x0: T0
// ..
let xn: Tn
(|x0: &? mut? T0, .., xn: &? mut? Tn| {
  // ..
  (*? x0)
  // ..
  (*? xn)
})(&? mut? x0, .., &? mut? xn)
```

Listing 27: Close Over Variables.

3.6 Convert Anonymous Closure to Function

This refactoring works on a closure converting it to a local function declaration. In Listing 28 we see that a closure is replaced with a block which has the value of a function.

Before refactoring

```
|i: &mut i32| -> i32 {
  foo(i)
}
```

After refactoring

```
{ fn bar(i: &mut i32) -> i32 {
    foo(i)
  }
  bar }
```

Listing 28: Convert Anonymous Closure to Function example.

Definition

This refactoring takes a closure expression as input and returns a block where the closure expression is converted to a function declaration and the new function declaration is set as the expression of the block.

Correctness

For this refactoring we have two preconditions. The closure should be closed over all variables and the parameters and the return type of the closure should be type annotated. We will not handle generic type parameters or lifetime parameters in this algorithm.

Algorithm

Algorithm 5 consists of collecting the parts of the anonymous closure and replacing it syntactically with a function declaration. In addition we find a fresh item identifier to be used as the new function name. A general example is shown in Listing 29.

Algorithm 5 Convert Anonymous Closure to Function

Input: C – A closure expression
Output: The changed code

1: $P \leftarrow$ parameterListOf(C)
2: $R \leftarrow$ returnTypeOf(C)
3: $B \leftarrow$ bodyOf(C)
4: $N \leftarrow$ freshItemIdentifierIn(B)
5: **pre** isClosedOverVariables(C)
6: **return** '{
fn N (P) -> R { B }
N
}'

Before refactoring

```
|x0: T0, .., xn: Tn|
    -> R {
  // ..
}
```

After refactoring

```
{
  fn ident (x0: T0, .., xn: Tn)
      -> R {
    // ..
  }
  ident
}
```

Listing 29: Convert Anonymous Closure to Function.

Before refactoring

```
fn bar() {
  use baz;
  // ..
  fn foo() {
    baz();
  }
}
```

After refactoring

```
use baz;
fn bar() {
  // ..
  fn foo() {
    baz();
  }
}
```

Listing 30: Lift Item Declarations.

3.7 Lift Item Declarations

This refactoring works on the `ItemDeclarations` and item bindings, preparing for a function declaration to be moved upwards in the AST.

Definition

Lift Item Declarations takes a `FunctionDeclaration` as input, and ensures that item bindings in the `FunctionDeclaration` are resolved inside the declaration itself, or to the closest parent `Mod` or `ImplBlock` or higher in the AST, so that the `FunctionDeclaration` can be moved while preserving item bindings.

In this Section, we will use F for the local `FunctionDeclaration`, G for the parent `FunctionDeclaration`, and M for the closest parent `Mod` of F.

Correctness

We have to ensure that all paths bound to `ItemDeclarations` before refactoring, are bound to the same declarations after refactoring.

Paths in M, but not in G We have to check whether moving the `ItemDeclaration` to M alters the item bindings of paths in M that are not in G, by shadowing item declarations occurring in M or higher in the AST.

Paths in G Paths in G may be bound to other declarations inside G, as the `ItemDeclarations` that are moved have lower priority after change. This may be done by collecting the item bindings in M before and after change, and asserting that they are the same. Depending on the signature and use of the type, this could either result in a change of behavior or making the program being rejected by the compiler. We will, however, not do this in this thesis as we believe this is a rare case.

Algorithm

Algorithm 6 shows the steps of this refactoring. In Listing 30, a general example is shown.

Algorithm 6 Lift Item Declarations

Input: F – A local `FunctionDeclaration`
Output: List of modifications

1: $M \leftarrow$ parentModuleOf(F)
2: $G \leftarrow$ parentFunctionOf(F)
3: **for all** $P' \in$ pathExpressionsIn(F) **do**
4: $D \leftarrow$ itemDeclarationResolvedBy(P')
5: **if** isInside(D, G) **and** isOutside(D, F) **then**
6: **delete** D from G
7: **insert** D to M
8: **end if**
9: **end for**

3.8 Lift Function Declaration

This refactoring converts a local `FunctionDeclaration` to an associated `FunctionDeclaration` by moving it upwards in the AST to the closest parent `ImplBlock`.

Definition

This refactoring takes a local `FunctionDeclaration` as input and moves it to the parent `ImplBlock`. Here we will use F for the local `FunctionDeclaration`, G for the parent `FunctionDeclaration` where F is placed, and I for the parent `ImplBlock` where G is placed.

The preconditions for this refactoring are that paths occurring in F and bound to `Items` are resolved to declarations in I or higher in the AST. Also, the identifier of F, should not be declared as an `Item` in I.

Correctness

Item binding preservation is ensured by the precondition. If F is not placed inside an `ImplBlock` but inside a `Mod`, F is moved to the `Mod` instead.

Algorithm

Algorithm 7 shows the steps of this refactoring. Here the paths resolved to F are prepended with `Self::` to reflect the new placement. In Listing 31, a general example is shown.

Algorithm 7 Lift Function Declaration

Input: F – A local `FunctionDeclaration`
Output: List of modifications

1: $I \leftarrow$ parentImplBlock(F)
2: $B \leftarrow$ parentBlockOf(F)
3: **for all** $Q' \in$ qualifiedPathsResolvedTo(F) **do**
4: **replace** Q' with 'Self::Q''
5: **end for**
6: **delete** F from B
7: **insert** F to I

Before refactoring

```
impl S {
  fn bar() {
    fn foo() { }
    // ..
    foo();
  }
}
```

After refactoring

```
impl S {
  fn bar() {
    // ..
    Self::foo();
  }
  fn foo() { }
}
```

Listing 31: Lift Function Declaration.

3.9 Helper functions

Table 3.1 shows an overview of the visitors and helper functions used by the algorithms in this chapter.

Name	Description
bodyOf(C)	returns the `Body` of `ClosureExpression` C
cfExprWithNext-Outside(B)	Returns the control flow expressions in B, that have their next `Statement` outside of B
closestParentBlockOf(S)	returns the closest parent `Block` of S
escapesAsBorrow(S, L)	Returns true iff a borrow of L is used after S
expressionOf(B)	Returns the `Expression` part of `Block` B
expressionPartOf(E)	returns the `Expression` part of `CallExpression` E
freshItemIdentifierIn(B)	returns a fresh item identifier in B
isClosedOverVariables(C)	returns iff there are no references inside C to locals outside C
isClosureExpression(E)	returns true iff E is a `ClosureExpression`
isInside(D, F)	returns true iff D is inside F
isOutside(D, F)	returns true iff D is outside F
itemDeclarationResolved-By(Q)	returns the `ItemDeclaration` that Q is bound to
itemDeclarationsIn(S)	returns the statements in S that are item declarations
localsDeclaredIn(S)	Returns the set of variables that are declared in S
localsUsedInAnd-DeclaredOutside(B)	Returns the set of variables that are used inside B, but declared outside
movedInside(B, L)	returns true iff L is *moved* inside B
parameterListOf(C)	returns the parameter list of `ClosureExpression` C
parentFunctionOf(F)	returns the closest parent `FunctionDeclaration` of F
parentModuleOf(F)	returns the closest parent `Mod` of F
pathExpressionsIn(F)	returns all `PathExpressions` in F
pathExpressions-ResolvedTo(B, L)	returns all `PathExpressions` bound to L
returnCodeOf(E)	returns the corresponding enum value of control flow expression E
returnTypeOf(C)	returns the return type of `ClosureExpression` C
usedLater(S, L)	Returns true iff the local L is used after S
usedMutableLater(S, L)	Returns true iff the local L is used mutable after S
usedMutableInside(B, L)	returns true iff L is *mutated* inside B

Table 3.1: Helper functions used in Extract Method.

Chapter 4

Box & Unbox Field

In this chapter, we will introduce the Box Field and Unbox Field refactorings. Based on a commit[1] on the Rust Language repository, Fowlers *Extract Class* [13] and Opdykes *Moving Members into a Component (Pointer to Aggregate Stored in Component)* [26], Box Field adds the builtin `Box` type around the type of a field of a `struct`. Box Field is a refactoring that changes the structure of a type whereas Extract Method changes the body of a function.

`Box` is a language type in Rust and it is the standard way of allocating values on the heap. A variable of type `Box` will be a pointer to where on the heap the actual value is stored. This can be useful when declaring recursive types or when we want to introduce indirection, for instance if a type is passed by value, one could reduce the number of bytes which has to be copied.

We will only consider structs with named fields, so tuple structs, enums and unions are not covered here. Unbox Field, which is conceptually an "undo operation" of Box Field, removes the `Box` type from a field of a `struct`.

In this chapter we will use A for `MatchArms`, B for `Blocks`, E for `Expressions`, F for `StructFields`, I for `Identifiers`, P for `Patterns`, S for `StructDeclarations` and T for `Types`.

4.1 Overview

Fowlers Extract Class moves one or more members of an existing class to a newly created class. It first creates the new class, then moves each of the fields and methods to the new class. Extract Class uses the Move Field refactoring to move a field from the existing class to the new class. Move Field first creates the new field on the target class, then removes the old field and replaces all references to the source field with references to the new field. Box Field will be similar to this, but here we only move a single field to an existing type declaration (`Box`). `Box` already has one generic field which will be the target of the Move Field operation, so the first step with creating a new field on the target class should be omitted. Box Field is therefore similar to Extract Class with one field, where the target class already exists. In Java,

[1] https://github.com/rust-lang/rust/pull/64374/commits

however, all objects are reference types and some extra considerations must be done as we will see later.

Opdykes *Moving Members into a Component* moves a set of members from an aggregate class to one of its components. With the set of members being a single field, and the component begin the `Box` type, Box Field will be similar to this refactoring.

In the C language, if we introduce a pointer to the type of a field, we also have to update any occurrences of the field with either the dereference operator `*` or the reference operator `&`. For Box Field, we will do changes similar to this to reflect the changed type of `Expressions`.

The purpose of the git commit, as indicated in the conversation, was to improve runtime characteristics, reducing memory traffic where the `struct` is passed out of a function. The aim of the refactoring was not to improve the readability of the code, which makes it a different kind of refactoring compared to Extract Method.

Composition of micro refactorings

Here we present the composition of the Box Field refactoring.

Box Field

1. Split Match Arms With Conflicting Bindings
2. Move Sub-pattern to if-part
3. Box Named Field

When changing the type of a `structs` field, we must also update places where the field appears to reflect the changes. A field may be used in `Patterns`, and we therefore have *Split Match Arms With Conflicting Bindings* and *Move Sub-pattern to if-part* which modifies `Patterns` where the field is used in ways that are not allowed with the `Box` type.

Box Named Field is the refactoring that changes the type of the field on the `StructDeclaration`, updates `StructExpressions` which creates new instances of the `struct` and updates accesses to the field in `FieldAccessExpressions`. It also updates accesses to the field in `PathExpressions` which is possible when the field introduces a new binding with `Patterns`.

Changes in the git commit

The git commit that Box Field was based on contained 55 additions and 37 deletions across 4 files. Listing 32 shows the different kind of changes in the commit. Here we will go through the changes to show what we used for the Box Field refactoring and to show what will be different.

Change 1: `StructDeclaration` This change can be summarized in the following steps:

1. A new `StructDeclaration` `DiagnosticBuilderInner` is added.

Before refactoring

```
// 1:
struct DiagnosticBuilder {
  <fields>
}


// 2:
DiagnosticBuilder {
  <fields> }


// 3:
self.<field>
// 4:
self.diagnostic
  .$n($($name),*);
// 5:



// 6:
db.handler.<expr>
// 7:
```

After refactoring

```
// 1:
struct DiagnosticBuilder(
  Box<DiagnosticBuilderInner);
struct
  DiagnosticBuilderInner{
    <fields> }
// 2:
DiagnosticBuilder (
  Box::new(
  DiagnosticBuilderInner{
    <fields> }))
// 3:
self.0.<field>
// 4:
self.0.diagnostic
  .$n($($name),*);
// 5:
fn handler(&self) ->
 &'a Handler {
  self.0.handler }
// 6:
db.handler().<expr>
// 7:
#[cfg(target_arch = "x86_64")]
static_assert_size!(PResult<'_,
  bool>, 16);
```

Listing 32: A summary of the changes made in the git commit.

2. `DiagnosticBuilderInner` is added as a field to `DiagnosticBuilder`.
3. All fields, except `DiagnosticBuilderInner` in `DiagnosticBuilder` are moved to `DiagnosticBuilderInner`.
4. The `DiagnosticBuilder` declaration is changed to a tuple declaration.
5. The type of the `DiagnosticBuilderInner` field is changed to `Box< DiagnosticBuilderInner>`.

Instead of adding a new type containing the boxed field, we will simply wrap `Box` around the type of the source field.

Change 2: `StructExpression`

1. `DiagnosticBuilderInner` is added in the initialization of `DiagnosticBuilder`
2. Fields in the `DiagnosticBuilder` init are moved to the `DiagnosticBuilderInner` init
3. The `DiagnosticBuilder` `StructExpression` is changed to a `CallExpression` (the constructor function for `DiagnosticBuilder`).
4. A `CallExpression` to `Box::new` is inserted around the first field of `DiagnosticBuilder`.

Change 3: `FieldAccess` Here the `FieldAccess` expression is updated to access the new field, and we will instead add a `DereferenceOperator` expression around `FieldAccess` expressions.

Change 4: `Macro` Here a macro definition is updated to reflect the layout of the `struct`. We will not support refactorings that requires changes in macro definitions.

Change 5: new `FunctionDeclaration` Here a *getter* function is added. We will not use this, but instead access the field using the dereference operator.

Change 6: `FieldAccess` In the commit, the new field is no longer public, and accesses to it is instead done through the new *getter* function. We will not change visibility of fields in the Box Field refactoring.

Change 7: static assertion Adds compile time assertions to the size of `PResult`, which contains `DiagnosticBuilder` for the `x86_64` architecture.

4.2 Split Match Arms With Conflicting Bindings

A field may introduce new bindings when used in `Patterns`. With the *or* pattern, denoted by `|`, a single binding may originate from different sources depending on which `Pattern` succeeded in the matching. An example of

this refactoring is shown in Listing 33. Before refactoring, the variable `f` is conditionally bound to field `f` from the `Pattern` at line 2 if `g` has the value `0`. If not, then `f` is bound to the field `g` from the pattern at line 3. The purpose of Split Match Arm With Conflicting Bindings is to rewrite such `Patterns` so that Box Field can be applied afterwards.

The body of the `MatchArm` is duplicated, which is not a good solution when the body is larger than a few lines. The duplicated bodies will not go back to the simplified version, and this solution was chosen because it enables more cases of Box Field.

Before refactoring

```
1 match .. {
2   S {f: f, g: 0} |
3   S {f: _, g: f}
4     => { /* body */ }
5 }
```

After refactoring

```
1 match .. {
2   S {f: f, g: 0}
3     => { /* body */ },
4   S {f: _, g: f}
5     => { /* body */ }
6 }
```

Listing 33: Split Match Arms With Conflicting Match Arms

Definition

Split Match Arms With Conflicting Bindings takes a `StructField` and a `MatchArm` as input and splits *or* `Patterns` where the field occurs in bindings with dual sources. The `Body` of the `MatchArm` is duplicated.

Correctness

Discovering bindings with different sources Here we should first find `MatchExpressions` with *or* `Patterns`. For these, we will visit each `Pattern` collecting the set of bindings it introduces and its source. Then, if one of the `Patterns` introduces a binding to the field, and another `Pattern` introduces a binding with the same identifier but to a different source, we should split out that other `Pattern` to a separate `MatchArm`.

Or-`Patterns` not in `MatchArms` The Or-`Pattern` is not supported in Function and Closure Parameters and in `let` declarations, but there is an RFC [2] for them and they may appear in the future. Or-`Patterns` occurring in `if let`, `while let` and `for` expressions will not be supported.

Algorithm

Algorithm 8 shows the steps. The refactoring should be applied to each `MatchArm` where the `StructField` occurs. Listing 34 shows a general example where P1 and P2 are conflicting `MatchArms` and B is the body of the `MatchArm`.

[2] https://github.com/rust-lang/rust/issues/54883

Algorithm 8 Split Match Arms With Conficting Bindings

Input: A – A `MatchArm`
F – A `StructField`
Output: The modified `MatchArm`

1: $P \leftarrow$ patternsWithConflictingBindingsTo(A, F)
2: $B \leftarrow$ bodyOf(A)
3: **return** 'P_0 => B,
…,
P_n => B, '
where $P_i \in P$

Before refactoring

```
1 match E {
2   // ..
3   P1 | P2 => B,
4   // ..
5 }
```

After refactoring

```
1 match E {
2   // ..
3   P1 => B,
4   P2 => B,
5   // ..
6 }
```

Listing 34: Split Match Arms With Conflicting Bindings.

4.3 Move Sub-pattern to if-part

`Patterns` may be matched on ranges using the `@` pattern as shown in Listing 35 where the field `f` is matched on the range 0 to 1. When changing the type of `f` to a `Box` type, this pattern is no longer supported and we will therefore apply Move Sub-pattern to if-part first. This moves the `@ Pattern` to the `MatchArmGuard` part of the `Pattern`, shown at lines 3 to 5 after refactoring.

Before refactoring

```
1 match .. {
2   S { f: f @ 0..=1}
3     => { /* body */ }
4 }
```

After refactoring

```
1 match .. {
2   S { f: f }
3     if match f {
4       0..=1 => true,
5       _ => false}
6     => { /* body */ }
7 }
```

Listing 35: Move Sub-pattern to if-part example.

Definition

This refactoring takes a `StructField` as input and returns a program where all `MatchArms` matching a sub-pattern on this field are moved into a new `MatchExpression` in the `MatchArmGuard` of the `MatchArm`.

Correctness

Pattern used in match arm If the binding is already used in the `MatchArmGuard`, then the identifier must be replaced with a path. If the binding is used in the body, then the binding must be introduced with a let-binding at the start of the body. This because bindings introduced in the if-part are not visible in the body of the match arm.

Pattern used in if let expression We will not handle cases where the pattern occurs in an `if let` expression.

Fields matched on wildcard and identifier pattern `Box` values can be matched on wildcard and identifier patterns, so we do not need to change these.

Changing the refutability By default, match arms must be exhaustive meaning that all possible values of the expression matched against should be covered in at least one of the arms. The if-parts of the arms are however, not part of this check when the compiler verifies it. This means that if we move a subpattern to the if-part of a match arm, we may change the refutability, possibly breaking the build. As an example `match 1 {_ => {}}` is a legal expression in Rust, but `match 1 {_ if true => {}}` is not.

Algorithm

The algorithm is shown in Algorithm 9, and a general example is shown in Listing 36.

Algorithm 9 Move Sub-pattern to if-part

Input: F − A `StructField`
Output: The modified program

1: **for all** $I \in$ identifierPatternWhereFieldOccurs(F) **do**
2: $M \leftarrow$ parentMatchArm(I)
3: **delete** I from M
4: **add** I to matchArmGuardOf(M)
5: **end for**

Before refactoring

```
match E {
  // ..
  S { F: Id @ Pattern } => B,
  // ..
}
```

After refactoring

```
match E {
  // ..
  S { F: Id }
    if match Id {
      Pattern => true,
      _ false
    } => B,
  // ..
}
```

Listing 36: Move Sub-pattern to if-part.

4.4 Box Named Field

Box Named Field wraps the `Box` type around a named `StructField` as shown in Listing 37. It will also update `Expressions` to reflect the new layout of the `struct`.

Before refactoring

```
struct S { t: T }
```

After refactoring

```
struct S { t: Box<T> }
```

Listing 37: Box Field example.

Definition

Given a `StructField`, this refactoring changes the type of the field from `T` to `Box<T>`. Any occurrences of the struct must also be updated so that the semantics of the program does not change. In addition to the `StructDeclaration`, the field may also occur in `StructExpressions`, `StructPatterns` and `FieldAccess` expressions.

The refactoring has these preconditions. The field should not occur in patterns, except for wildcard-patterns and identifier patterns. The type of the field should not already be `Box<T>`. The `struct` should not implement the `Copy` trait.

Correctness

When we change the type of the field to `Box<T>` we should consider each occurrence of the field in the program.

Struct Declaration The `StructDeclaration` declares the members of a struct and their types. It may also contain generics and lifetime parameters. Here we should insert the `Box` type around the type of the field.

Struct Expressions `StructExpressions` creates a new instance of a `struct`. With this, we should wrap the value of the field with a call to

Before refactoring

```
1 S {field};
2 S {field: 0};
3 S {..other};
4 S {field: other.field};
```

After refactoring

```
1 S {field: Box::new(f)};
2 S {field: Box::new(0)};
3 S {..other};
4 S {field: other.field};
```

Listing 38: `StructExpressions`

the function `Box::new` as shown in Listing 38. If the field is not mentioned here we do not need to change this expression. This may happen if the `StructBase` construct is a part of the expression as shown at line 3. If the field occurs as a field init shorthand, we should also insert the identifier of the field and a semicolon before the new call to `Box::new` as shown at line 1. If the value resolves to the same field that we are applying Box Named Field to we can omit the change, as the type of the value also changes to `Box<T>` as shown at line 4.

Field Access Expressions `FieldAccessExpressions` are used to access members of a struct. The syntax is: `Expression.Identifier` and if `Expression` resolves to the struct in change and `Identifier` is the field we should wrap this `FieldAccessExpression` in a `DereferenceOperator` expression to preserve the type of the expression.

Struct Patterns If the field is matched on a wildcard pattern, no changes are needed. If it is matched on an identifier pattern, it introduces a new binding. The type of this binding will change from `T` to `Box<T>`. Here we have two options. We can either dereference all occurrences of this variable or we can introduce a new binding with the same identifier at the top of the body of the match arm. If the pattern occurs as a part of a `Match-Expression`, we have to visit the `MatchGuard`, as the new binding is visible here. An example is shown in Listing 39 where the variable `f` changes type to `Box<T>`. In this thesis we will go for the second option, replacing occurrences of `f` with `(*f)`.

If the binding resolves to multiple sources, with at least one not being the field, then the Split Match Arms With Conflicting Bindings should be applied first. If it is any other pattern, then the Move Sub-pattern to if-part should be applied first.

Copy and Drop traits `Copy` is a special trait that allows the compiler to use the `Clone` implementation to copy a value to a new destination instead of moving it. The integer types are examples which have the `Copy` trait, and it can also be added for user defined types such as `structs`. The `Copy` trait has a special rule which says that it can only be implemented for types which does not have the `Drop` trait, and all of its fields should have the `Copy` trait. As `Box` implements `Drop` this means that structs which have the `Copy` cannot have fields of type `Box<T>`. It is therefore a precondition that the `struct` that contains the field should not have the `Copy` trait.

Before refactoring

```
match .. {
  S { f } => {
    // f: T
    let g = f;
} }
```

Option 1

```
match .. {
  S { f } => {
    // f: Box<T>
    let f = *f;
    // f: T
    let g = f;
} }
```

Option 2

```
match .. {
  S { f } => {
    // f: Box<T>
    let g = *f;
} }
```

Listing 39: `StructPatterns`

Before change

```
1 #[derive(Clone, Copy)]
2 struct S { f: i32 }
```

After change

```
1 #[derive(Clone, Copy)]
2 struct S { f: Box<i32> }
```

Listing 40: Box Field example which should not be allowed.

Listing 40 shows an example of a field that cannot be boxed. We get the following error message: 'the trait 'Copy' may not be implemented for this type. rustc(E0204)'

Derive macros Derive macros are a type of procedural macros which may be used on struct declarations. Examples of builtin derive macros are `Clone`, `Copy` and `Debug` which all adds implementations of traits with the same names. These macros uses the traits of the fields, so the source code generated by some of the builtin macros are identical for types of `T` and `Box<T>`.

Method-call expression candidates When the compiler resolves method calls it also performs some type conversions. In some situations, we may accidentally change the binding of a method call when changing a field from `T` to `Box<T>`. In Listing 41 a trait `T` with a single function `foo` is implemented both for `V` and `Box<V>`. After boxing field `v` on `U`, the method call on line 7 on the right side changes binding, but line 9 is correct. In this thesis we will always add the `DereferenceOperator` expression around `PathExpressions` that resolves to the `StructField`.

Default traits where `Box<T>` delegates to `T` `Box` comes with implementations for some builtin traits for types which also implements the traits. In Listing 42, calling clone on an expression of type `S` or `Box<S>` provides the same result. Another example is the `PartialEq` trait, where `Box<T>` delegates to the implementation of `T` if `T` has the `PartialEq` trait. This means that if the `StructField` is used on the left- and right hand side of an `OperatorExpression` where the operator is `==`, then we can omit inserting the `DereferenceOperator` on both sides. For this thesis, we will support

Before change

```
trait T { fn foo(&self); }
impl T for V {
  fn foo(&self) {
  print!("V::foo()");
}}
impl T for Box<V> {
  fn foo(&self) {
  print!("Box<V>::foo()");
}}
struct V;
struct U{v: V}
// ..
let u1 = U{v: V};
// prints V::foo()
u1.v.foo();
```

After change

```
// ..
struct U{v: Box<V>}
// ..
let u1 = U{v: Box::new(V)};
// incorrect,
// prints Box<V>::foo()
u1.v.foo();
// correct, prints V::foo()
(*u1.v).foo();
```

Listing 41: Change of binding with a Method Call Expression.

```
#[derive(Clone)]
struct S {f: i32}
// ..
s.clone();
Box::new(s).clone();
```

Listing 42: Calling `clone` on `S` or `Box<S>` provides the same result.

some of these builtin traits by not visiting the code generated as it does not require any changes.

`std::ops::Deref` and omitting deref changes `Box<T>` has by default an implementation for the `Deref` trait for the type `T`. This means that a borrow of `Box<T>`, with type `&Box<T>` can be implicitly converted to `&T`. So the insert of the `Deref` expression around the `FieldAccessExpression` can be omitted where there is a borrow around the `FieldAccessExpression`. This is an optimization, so we will not handle this in this thesis.

Algorithm

Algorithm 10 shows the steps. First the `StructDeclaration` is updated, then bindings introduced by patterns, `FieldAccessExpressions` and `StructExpressions` are updated. Listing 43 shows a general example. At line 1 we see the `StructDeclaration`, at line 3 a `FieldAccess` and at line 4 a `StructExpression`. At line 6 a `Pattern` introduces a new binding, `Id`, to the field, so we have to add a dereference to all occurrences of `Id`.

Algorithm 10 Box Named Field

Input: F – A `StructField`
Output: List of modifications

1: $S \leftarrow$ closestParentStructDeclaration(F)
2: **pre** collectStructPatterns(F) $= \emptyset$
3: **pre not** hasTrait(S, $Copy$)
4: **replace** F with 'I: Box<T>'
where $I =$ ident(F), $T =$ typeof(F)
5: **for all** $B \in$ structPatternsBinding(F) **do**
6: **for all** $E \in$ pathExpressionsResolvedTo(B) **do**
7: **replace** E with '($*E$)'
8: **end for**
9: **end for**
10: **for all** $E \in$ fieldAccessExpressions(F) **do**
11: **replace** E with '($*E$)'
12: **end for**
13: **for all** $E \in$ structFieldExpressions(F) **do**
14: **replace** E with 'Box::new(E)'
15: **end for**

Before refactoring

```
struct S { field: T }
fn foo() {
  E.field
  S { field: E }
  match E {
    S { field: Id } => {
      Id
    }
  }
}
```

After refactoring

```
struct S { field: Box<T> }
fn foo() {
  (*E.field)
  S { field: Box::new(E) }
  match E {
    S { field: Id } => {
      (*Id)
    }
  }
}
```

Listing 43: Box Named Field.

4.5 Unbox Named Field

Unbox Named Field removes the `Box` type from a `StructField`. An example of this is shown in Listing 44.

Before refactoring	After refactoring
`struct S { t: Box<T> }`	`struct S { t: T }`

Listing 44: Unbox Named Field example.

Definition

Unbox Named Field takes a `StructField` of a named `StructDeclaration` as input and returns a program where the fields type is changed from `Box<T>` to `T`. The preconditions for this refactoring are that the type of the field should be `Box` and that the field should not be recursive.

Correctness

Struct Declaration On the `StructDeclaration`, we should change the type of the field from `Box<T>` to `T`. But before that, we should check if the field is recursive, meaning that it contains the `struct` itself, but not inside a `Box`.

Field Access Expressions Similarly to Box Field, we need to change occurrences which bind to the field in change. But here we need to check whether the `FieldAccessExpression` occurs on the left hand side of an `AssignmentExpression`. If this is the case, then we should add the `DereferenceOperator` expression around the right hand side of the `AssignmentExpression`. Otherwise, we add a call to `Box::new` around the `FieldAccessExpression`.

Struct Expressions `StructExpressions` creates new instances of `structs` and here we need to add the `DereferenceOperator` expression around the `Expression` assigned to the `StructField` that is unboxed. An example of this is shown at line 5 in Listing 45.

Struct Patterns As `StructPatterns` can introduce new bindings, we need to collect all cases where the field introduces a new binding in a `StructPattern`, and then finding updating occurrences of this binding, similar to the case for `FieldAccessExpression`. This is shown at lines 7 to 10 in Listing 45.

Algorithm

The steps are shown in Algorithm 11, and a general example is shown in Listing 45. The general example shows the different changes that are needed. Line 1 shows a `StructDeclaration`. Line 3 shows a

`FieldAccessExpression` that occurs on the left side of an assignment. Line 4 shows the case when the `FieldAccessExpression` occurs on the right side. Line 5 shows a `StructExpression`. Lines 6 to 10 shows a `MatchExpression` that introduces a new binding to the field, where lines 8 and 9 has `FieldAccessExpressions` similar to lines 4 and 5.

Algorithm 11 Unbox Named Field

Input: F − A `StructField`
Output: List of modifications

```
1: pre not isRecursive(F)
2: replace F with 'I: T'
                where I = ident(F), T = box_typeof(F)
3: for all B ∈ structPatternsBinding(F) do
4:   for all E ∈ pathExpressionsResolvedTo(B) do
5:     if inLhsAssign(E) then
6:       E_r ← rhs(E)
7:       replace E_r with 'Box::new(E_r)'
8:     else
9:       replace E with '(∗E)'
10:    end if
11:  end for
12: end for
13: for all E ∈ fieldAccessExpressions(F) do
14:   if inLhsAssign(E) then
15:     E_r ← rhs(E)
16:     replace E_r with 'Box::new(E_r)'
17:   else
18:     replace E with '(∗E)'
19:   end if
20: end for
```

4.6 Helper functions

The visitors and helper functions used in algorithms in this chapter are shown in Table 4.1.

Before refactoring

```
struct S { f: Box<T> }
fn foo() {
  path.f = E
  E = path.f
  S { f: E }
  match .. {
    S { f: Id } => {
      E = Id
      Id = E
    }
  }
}
```

After refactoring

```
struct S { f: T }
fn foo() {
  path.f = (*E)
  E = Box::new(path.f)
  S { f: (*E) }
  match .. {
    S { f: Id } => {
      E = Box::new(Id)
      Id = (*E)
    }
  }
}
```

Listing 45: Unbox Named Field.

Name	Description
collectStructPatterns(F)	Collects all `StructPatterns` where the field F occurs, and is matched against an expression that is not the wildcard expression
fieldAccessExpressions(F)	Collects all `FieldAccessExpressions` resolved to F
identifierPatternWhere-FieldOccurs(F)	Collects `@` patterns matched on F
inLhsAssign(E)	Returns true iff E is in the left hand side of an `AssignementExpression`
hasTrait(S, T)	Returns true iff the struct S has the trait T
pathExpressions-ResolvedTo(B)	Collects all `PathExpressions` bound to variable B
patternsWithConflicting-BindingsTo(A, F)	Collects *or* patterns in A introducing conditional bindings to F
rhs(E)	Returns the right hand side of an `AssignmentExpression` where E occurs
structFieldExpressions(F)	Collect all fields in `StructExpressions` resolved to F.
structPatternsBinding(F)	Collects all `StructPatterns` where the field F occurs, and is not matched against an expression, but introduces a new binding

Table 4.1: Some helper functions for the Box field refactoring.

Chapter 5

Implementation

The codebase for the refactorings presented in this thesis is organized as a single git repository. In this chapter, we will go through each subfolder in the repository. In Section 5.1 we will look at the CLI which contains two executable binaries. The data used for the tests of the refactorings is shown in Section 5.2. The CLI developed for the experiments is shown in Section 5.3. The library containing the refactorings are shown in Section 5.4. The definitions of the output of the CLI is shown in Section 5.5, and the language server is shown in Section 5.6.

5.1 `em-refactor-cli`

This folder contains two executable binaries from the files `driver.rs` which s output as `em-refactor-driver` and `main.rs` which is output as `cargo-em-refactor`.

Executable: `cargo-em-refactor`

`main.rs` is the refactoring CLI intended to be called by the language server or from the terminal and is output as `cargo-em-refactor`. It has two subcommands, `candidates` and `refactor`.

It accepts the flag `--target-dir` which sets the directory in which rustc will write files to. This is useful to prevent conflicts with other tools. It also accepts the flag `--workspace-root` which sets the directory of the project. If this flag is not set, the current directory is used.

It will use `cargo metadata` to get information about dependencies in the crate, and then use this information to call `cargo clean` to clean local packages, so that `rustc` will be invoked on the local package.

It then calls `cargo check` with the `em-refactor-driver` as a "compiler wrapper". This means that `cargo` will call our driver instead of the Rust compiler. If the refactoring is a composite one, then `cargo check` is called once for each micro refactoring in the composite refactoring, while appending the previous resulting diff to the output of the refactoring.

Subcommand. The `candidate` subcommand is on the format `candidates <refactoring>` and it takes a refactoring name as input and outputs a list of text ranges, where the refactoring can be applied.

Subcommand. The refactorings can be invoked using the `refactor` subcommand, which is on the format `refactor <refactoring> <file> <selection>`. It takes a refactoring name, file name and a selection as input and returns a list of list of changes if it is a composite refactoring, and a list of changes if it is a micro refactoring.

As an example, we can run `cargo-em-refactor refactor extract-method src/main.rs 20:40` to apply the Extract Method refactoring in file `src/main.rs` from byte 20 until byte 40.

Executable: `em-refactor-driver`

This executable is the output of the file `driver.rs` and it is invoked once by `cargo` for each executable binary, library or test in the project that is the target of the refactoring. It will get the same argument as if it was `rustc`, and here we have to pass these arguments to the Rust compiler, and also configure the callbacks that we use for our refactorings. The refactoring arguments are passed using an environment variable. The callbacks we use are `after_expansion` which occurs after both parsing and macro expansion is done, and `after_analysis` which occurs when the AST is converted to the simpler High-Level Intermediate Representation (HIR), and type checking is available.

Tests

The folder also contains test invocation for the refactoring unit tests and some integration tests for the CLI which target various structured Rust projects and tests for the output of the CLI. The test data is located in the folder `em-refactor-examples` which is explained later.

Composition of micro refactorings

Composition of micro refactorings, which has to be done for the Extract Method refactoring, is done in the `em-refactor-cli` project. Since we have to rebuild the AST from source after modifying it, we collect the changes that a micro refactoring produces, and invoke the next micro refactoring with those changes as an additional argument.

We also have to handle the input to each micro refactoring, as the first selection in the source code changes as we modify it. The micro refactorings therefore supports to output comments around AST-nodes that it modifies or inserts. In Listing 46 we show an example where the Pull Up Item Declaration refactoring is applied on (a). The result is shown in (b) where it inserts a special comment around the highlighted line. Extract Block is then applied on (b), using the range indicated by the comment. The output of Extract Block is shown in (c), where it marks the newly created `Block`,

which will be used for the Introduce Closure refactoring. As a last step of the composite refactoring, all of these comments are removed.

```
fn foo() {
  let i = 0;
  fn bar() {
    .. }
  ..
}
```

(a)

```
fn foo() {
  fn bar() {
    .. }
  /*stmts:start*/
  let i = 0;
  /*stmts:end*/
  ..
}
```

(b)

```
fn foo() {
  fn bar() {
    .. }
  let (i) =
  /*block:start*/
  {
    let i = 0;
    (i)
  };
  /*block:end*/
  ..
}
```

(c)

Listing 46: Compositions

Handling multiple crates

As a package may contain multiple crates, resulting in multiple top level root modules, we have to visit several ASTs to find the correct place that the refactoring was invoked on. This is expected to be common, since unit tests are often placed inside the same source file as the code that they test are in. This results in one AST where the `test` flag is set to true, and another where it is set to false.

To handle the problem with multiple ASTs in a package, we run the refactoring once for each AST and combine the errors or file changes that they make. Also, we ensure that only identical file changes are allowed, so changes that overlap will result in an error.

5.2 `em-refactor-examples`

This folder contains the data used for the tests of the refactorings and also some crates used to test the CLI. In the integration tests, we test the tool against workspaces and packages containing multiple crates.

A refactoring test contains three files placed in a subfolder named after the refactoring. The format of the files are `<test-name>.rs`, which contains the program that is refactored, `<test-name>.after.rs`, which contains the expected code after refactoring and `<test-name>.json` which contains the arguments to the refactoring. The expected program, which is in the `<test-name>.after.rs` file, is compared as a string against the output of the refactoring in test.

5.3 em-refactor-experiments

In this folder, we find the CLI used for the experiments. It takes a Rust project and a refactoring as an input. It then queries the candidates for the refactoring in the project and for each candidate it will execute the refactoring. More details are shown in Chapter 6.

5.4 em-refactor-lib

This folder contains the implementation of the refactorings and the candidate queries. It also contains the code needed to configure the compiler with callbacks.

Refactorings

The refactoring implementations are located in the `refactorings` subfolder. A refactoring is a function with a signature of either `(&AstContext, Span)-> QueryResult<AstDiff>` if the refactoring only uses the AST or `(&TyContext, Span)-> QueryResult<AstDiff>` if the refactoring requires type checking.

Type: `AstContext`**.** This contains references to the `rustc` types `Compiler` and `Queries`. It has methods commonly used by the refactorings and visitors.

Type: `TyContext`**.** This contains a reference to the `rustc` type `TyCtxt`. Similarly to `AstContext`, it has methods commonly used by the refactorings and visitors.

Type: `Span`**.** This is a `rustc` type representing a range in the source code. It has information on whether it was the result of a macro expansion, a desugaring or not. It can also be converted to a `String` using `SourceMap` in `rustc_span`.

Type: `QueryResult<AstDiff>`**.** `QueryResult<T>` is an alias for `Result< T, RefactoringError>` used to pass out errors from the refactorings and visitors. This is the return type of all of the refactorings and most of the visitors. `AstDiff` is a list of replacements.

Visitors

The visitors are either located in the `visitors` folder when used by multiple refactorings or in the same folder as the refactoring when used only by one refactoring. The visitors implement one of three `Traits` shown here, provided by the `rustc` library.

Trait: `rustc_ast::visit::Visitor`. With an implementation of this we can visit the nodes of the AST, either before, or after macro expansion. In this project we only visit the AST after macro expansion, checking whether the `Spans` of nodes are produced by macros or not.

Trait: `rustc_hir::intravisit::Visitor`. This kind of visitor traverses the HIR, which is a simplified version of the AST. The origins of the desugarings can in most cases be reconstructed, using helper functions based on code found in the Clippy [6] project. Types of `Expressions` can be queried using the `typeck_tables_of` method on `TyCtxt` with the id of the closest parent `Function` body as an argument.

Trait: `rustc_typeck::expr_use_visitor::ExprUseVisitor`. This is a special kind of visitor, which visits all `Expressions`, providing whether the `Expression` was *borrowed*, *copied* or *moved*.

Candidates

The implementation of the candidate queries are located in the `candidates` subfolder. They visit the AST and outputs all possible invocations of a given refactoring. Candidate queries are implemented for Extract Method and Box Field.

Rustc invocation

The library also contains the code required to use the compiler as a library with callbacks. The code for this was based on both Clippy and Rerast [22].

File: `run_refactoring.rs`. This file contains functions for invoking the Rust compiler from the library. It configures the custom callbacks, calls `rustc_driver::run_compiler` and outputs the result as JSON.

File: `my_refactor_callbacks.rs`. This file contains the callbacks for `after_expansion` and `after_analysis`. Here we call the query on either the AST or HIR, depending which arguments were passed in.

Using the `rustc` library

To use the `rustc`, we need to use the `nightly` version of the compiler. The documentation is publicly available[1]. We also need to add `extern crate` declarations in the root module for each of the libraries that we are using. The most important `crates` that we use are `rustc_ast` and `rustc_hir` which provides definitions for AST and HIR nodes, and visitors on those.

The `rustc_driver::Callbacks` have four methods that can be implemented. `config` is used to configure the compiler before it is created. `after_parsing` is called after parsing is done, but before macros are expanded. `after_expansion` is called after macro expansion is done. In this

[1] https://doc.rust-lang.org/nightly/nightly-rustc/

project we use `after_expansion`, and instead check while visiting whether the `Spans` was a result of macro expansion. `after_analysis` is called after the analysis is done. The refactorings that require type checking uses this method, and they operate on the HIR.

The compiler then converts the HIR to a simpler structure called Mid-level Intermediate Representation (MIR). The MIR is based on a control flow graph, consisting of basic blocks. Here, the borrow checking is done. We do not use the MIR in this project, but we include it here for completeness. In Listing 47 we show an example of the syntax of the HIR and MIR. The HIR code were generated running `rustc +nightly -Zunpretty=hir main.rs` and the MIR code with `rustc +nightly -Zunpretty=mir main.rs`.

Rust

```
fn main() {
  if true { }
}
```

HIR

```
#[prelude_import]
use ::std::prelude::v1::*;
#[macro_use]
extern crate std;
fn main() {
  match { let _t = true; _t }
  { true => { } _ => { } }
}
```

MIR

```
fn main() -> () {
 let mut _0: ();     // return place in scope 0 at
 ↪  if.rs:1:11: 1:11
 let mut _1: bool;  // in scope 0 at if.rs:2:5: 2:9

 bb0: {
  StorageLive(_1); // bb0[0]: scope 0 at if.rs:2:5: 2:9
  _1 = const true; // bb0[1]: scope 0 at if.rs:2:5: 2:9
                   // ty::Const
                   // + ty: bool
                   // + val: Scalar(0x01)
                   // mir::Constant
                   // + span: if.rs:2:5: 2:9
                   // + literal: Const { ty: bool, val:
 ↪  Scalar(0x01) }
  StorageDead(_1); // bb0[2]: scope 0 at if.rs:4:1: 4:2
  return;          // bb0[3]: scope 0 at if.rs:4:2: 4:2
 }
}
```

Listing 47: Rust - HIR - MIR example.

5.5 em-refactor-lib-types

This folder contains types that are used by `em-refactor-cli`, `em-refactor-experiments` and `em-refactor-lib`. This is mostly the definition of the output of the `cargo-em-refactor` binary. It also contains the names of the refactorings, and the composition of Extract Method.

5.6 em-refactor-ls

The `em-refactor-ls` folder is a npm project written in TypeScript which contains both the language client and server. They enable the use of the refactorings in an IDE, and the client is an extension for Visual Studio Code. It was based on an official example [2] found on github. The client and server communicate over the Language Server Protocol.

Server

The language server is located in the subfolder `server`. Dependency Injection is used to organize the code in the server. Here we will go through some of the most important files for the server.

File: `services/connection.ts` Here we find the initialization of the server. Here the server returns which `CodeActions` it supports and which `Commands` it supports.

File: `services/CodeActionService.ts` The `CodeActionService` class in this file handles `textDocument/codeAction` requests from the client. These request are sent from the client to compute the commands that are available for a given document and range in the document.

For this project, we return a `Command` which can be executed on the server, but it is also possible to return `WorkspaceEdits` directly from this method. Here we return one `Command` for each available refactoring for the given text range.

File: `services/ExecuteCommandService.ts` Here we find the `ExecuteCommandService` class which handles `workspace/executeCommand` requests in the `handleExecuteCommand` method. The refactoring `Commands` are found in the file `services/commands/RefactorCommand.ts`. Here the refactoring CLI is invoked by the server, and if it is successful, one or more `workspace/applyEdit` requests are sent from the server to the client, with the file changes from the refactoring that the client should perform.

Client

The client, which is a Visual Studio Code Extension, is located in the subfolder `client`. The file `extension.ts` contains the code for the client.

[2] https://github.com/microsoft/vscode-extension-samples/tree/master/lsp-sample

Here the server is started, and the client is configured to only watch Rust files, which are files that ends with `.rs`.

Chapter 6

Experiments

In this chapter we will show the experiments that were run on the Extract Method refactoring from Chapter 3 and Box Field from Chapter 4. Both refactorings were run on two open source projects, *RustyXML*[1] and *tokenizers*[2]. The goal of the experiments was to demonstrate the correctness of the algorithms and implementations of the refactorings.

6.1 Experiment setup

The experiments were run by first collecting a list of all candidates. Then, for each candidate, the refactoring was run, and if it was successful, the file changes were applied and unit tests were run. Before running an experiment, the project was compiled and tested running the command `cargo + nightly-2020-04-15 test --target-dir=./target/refactorings`, both to ensure that the initial test run was passing, and that external dependencies were downloaded and compiled before running the experiment.

Candidates

A refactoring candidate is a place in the source code where a given refactoring can be applied. Software metrics can be used to rank candidates, and to compare the quality of a program before and after refactoring. We did not use software metrics for the experiments as there is not much support for Rust yet and due to time constraints. For the Extract Method refactoring, all subsequences of blocks were considered to be candidates similar to what Kristiansen [20, 21] used, but without further reducing the number of candidates. Although extracting a method from a single line is not useful in most cases, we chose to include these candidates as we consider them to be tests of the implementation. For the Box Field refactoring, all fields of `struct` declarations with named fields were considered candidates.

[1] https://github.com/Florob/RustyXML/tree/9e88e2f43ad3c0ff91953fedcfa5bebd002950c5
[2] https://github.com/huggingface/tokenizers/tree/02cc97756ffb9193b5d6d8dfcdeb7bf08adf2516

Metric	Description
Candidates found	Number of places where the refactoring is applicable
Successful refactorings	Number of applied refactorings where the refactored program compiles and all tests passes
Internal errors	Number of attempted refactorings where the refactoring made an internal error
Introduced Rustc error	Number of attempted refactorings where the refactoring resulted in code that did not compile
Introduced unit test failure	Number of attempted refactorings where the refactoring resulted in code made at least one unit test fail
Total duration	The total duriation of the experiment
Time spent compiling and refactoring	The time spent compiling and refactoring
Unit tests duration	The time spent writing changes to disk, running unit tests and reverting the changes

Table 6.1: The metrics captured in the experiment.

Metrics

The following data shown in Table 6.1 were collected. An important note here is that we were not able to separate execution times of compilation and refactoring so we cannot compare the average duration of a refactoring between the two projects.

Selection of projects

The following criteria were used. The project should be open sourced and available as a git repository. It should have unit tests, with all tests passing. There should be few manual steps required before being able to compile and test the project. As we are testing all candidates, we do not want the project to be large, both to keep the number of candidates to a reasonable amount, and because we are compiling the project between each micro refactoring. Unicode characters should not occur in comments or strings as they are not supported by the CLI. Heavy use of generics, futures or the "?" operator which is not supported were also a criteria.

As for finding Rust projects, this [3] list of projects is useful. We selected *RustyXML*, an XML parser, and *tokenizers*, a tokenizer. *RustyXML* had 22 unit tests and *tokenizers* had 51 unit tests.

6.2 Results

Here the results of the experiments are presented. The results of Extract Method is shown in Table 6.2, and the results of Box Field is shown in Table

[3] https://github.com/rust-unofficial/awesome-rust

6.3.

Extract Method in RustyXML

Out of 933 candidates found in RustyXML, 738 were successful, meaning that all the microrefactorings resulted in code that compiled, and all the unit tests succeeded after applying the changes of the microrefactorings. 11 of the errors were internal errors, meaning that an error was caught in the tool while applying the microrefactoring. 184 of the errors were Rustc errors which means that we found a refactoring, but after applying it, the Rust compiler rejected the program with an error code. None of the applied refactorings introduced unit test failures. The total time spent compiling and refactoring were 32 minutes and 43 seconds, with an average of 2.1 seconds per refactoring. Figure 6.1 shows the number of candidates and successful refactorings grouped by number of non-empty lines in the candidate. A line is considered empty here, if it contains only whitespace characters.

Table 6.2 also shows an overview of compiler errors[4] that were introduced, grouped by the microrefactoring that introduced them. Here we see that 10 applications of Introduce Closure resulted in code that did not compile, because it incorrectly introduced `[E0106]` 10 times. `[E0106]` occurs when a lifetime is missing from a type. 19 applications of Introduce Closure resulted in the compiler error `[E0308]`, meaning that the compiler was unable to infer the concrete type of a variable. 20 applications of Close Over Variables caused the error `[E0495]` which occurs when a lifetime cannot be determined, and 60 applications caused `[E0596]` which is caused by mutably borrowing a non-mutable variable.

An example of a successful Extract Method refactoring is shown in Listing 48. The refactoring outputs absolute paths for types, and they have been shortened here to make the example shorter. The selected statement, spanning from line 3 to line 9 is highlighted. Here we see that a `return` expression occurs within the selection. The resulting code is shown in "After refactoring". Here we see the method invocation at line 3, and a `match` expression around it, which handles the `return` expression. The new method declaration is at line 9, with two parameters `prefix` and `self_`. The return type of the method is the `enum ReturnFoo` declared at line 30.

Extract Method in tokenizers

Extract Method was only run on the file `src/models/bpe/word.rs` inside the subfolder `tokenizers`. This file has several methods, and one of them had almost 100 lines. As shown in Table 6.2 this file had 283 candidates where 223 applications of Extract Method were successful, 0 applications introduced unit test failures, and 60 applications introduced code that did not compile. The total duration of the experiment was 63 minutes and 21 seconds, where 36 minutes and 27 seconds were spent compiling and refactoring and 26 minutes and 53 seconds were spent running unit tests.

[4]Compiler Error Index: https://doc.rust-lang.org/error-index.html

Summary of Extract method	RustyXML	tokenizers
Candidates found:	933	283
Successful refactorings:	738	223
Internal errors:	11	0
Introduced Rustc error:	184	60
Introduced unit test failure:	0	0
Total duration:	38m 43s	63m 21s
Time spent compiling and refactoring:	32m 43s	36m 27s
Unit tests duration:	6m	26m 53s
Failed at: Pull Up Item Declarations		
Introduced internal errors:	7	0
Failed at: Extract Block		
Introduced Rustc error [E0308]:	1	0
Introduced Rustc error [E0597]:	3	0
Failed at: Introduce Closure		
Introduced internal errors:	1	0
Introduced Rustc error (no code):	35	0
Introduced Rustc error [E0061]:	3	0
Introduced Rustc error [E0106]:	10	15
Introduced Rustc error [E0308]:	19	0
Introduced Rustc error [E0373]:	1	0
Introduced Rustc error [E0495]:	3	9
Introduced Rustc error [E0500]:	1	0
Introduced Rustc error [E0502]:	3	0
Introduced Rustc error [E0658]:	2	0
Failed at: Close Over Variables		
Introduced internal errors:	3	0
Introduced Rustc error (no code):	4	0
Introduced Rustc error [E0312]:	1	0
Introduced Rustc error [E0382]:	0	15
Introduced Rustc error [E0495]:	20	3
Introduced Rustc error [E0502]:	0	14
Introduced Rustc error [E0594]:	1	0
Introduced Rustc error [E0596]:	60	4
Failed at: Convert Closure to Function		
Introduced Rustc error [E0106]:	2	0
Introduced Rustc error [E0401]:	6	0
Introduced Rustc error [E0495]:	1	0
Failed at: Lift Function Declaration		
Introduced Rustc error [E0425]:	6	0
Introduced Rustc error [E0599]:	2	0

Table 6.2: Results of the Extract Method experiments.

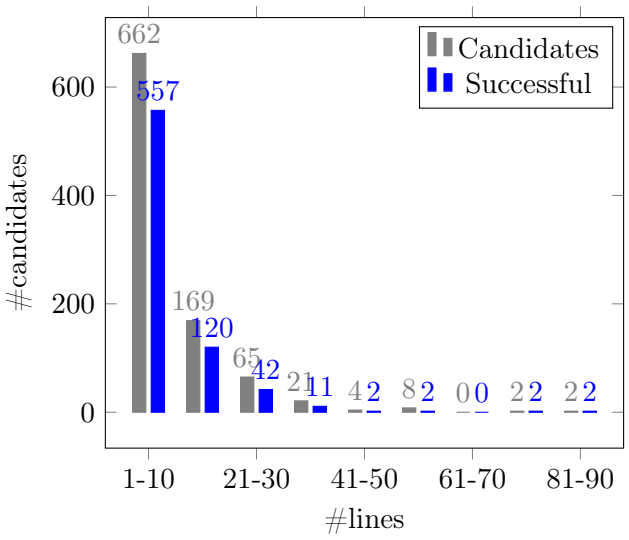

Figure 6.1: Extract Method in RustyXML

A refactoring in tokenizers, which includes compilation, had an average duration of 7.7 seconds.

Figure 6.2 shows the candidates grouped by number of non-empty lines. Here we see that on average, there were more candidates spanning 20 lines or more, than what we had in RustyXML. We also see that many of the longer candidates succeeded.

Box Field in RustyXML

The total duration of the experiment was 33 seconds, with 16 seconds spend compiling and refactoring. This gives an average of 0.5 seconds per refactoring. 34 candidates were found for Box field. Of these, 30 were successful, 3 lead to compile errors, 1 caused an internal error and 0 introduced unit test failures. The internal error was related to a macro invocation, as the tool attempted to modify the file where the macro definition was stored. The Rustc errors were all `[E0063]`, which means that a struct's field was not provided. Listing 49 shows one of the failed attempts. Here Box Field is applied on the field `name` of `StartTag`. The struct is used in a `MatchPattern`, and inside the `MatchArm` the field is used in a `StructExpression` as a shorthand. A correct change here would be to change line 7 from `name` to `name: (*name)`.

Box Field in tokenizers

Box Field was only run on the project `tokenizers` inside the repository. Here 132 candidates were found, where 105 of the applications of Box Field were successful. 23 of the applications caused internal errors, 4 caused compile errors and 0 caused unit test failures. The total duration was 17 minutes and 11 seconds and 3 minutes and 18 seconds were spent compiling and refactoring, giving an average of 1.5 seconds per refactoring.

Before refactoring

```
impl Parser {
fn in_tag_name // ..
    let ns = match prefix {
      None => self.namespace_for_prefix(""),
      Some(ref pre) => match self.namespace_for_prefix(&pre) {
        None => return self.error("..."),
        ns => ns,
      },
    };
    // ..
}
```

After refactoring

```
impl Parser {
  fn in_tag_name // ..
    let ns = match ({ Self::foo })(&prefix, self) {
      ReturnFoo::Expr(e) => e,
      ReturnFoo::Return(e) => return e
    };
    // ..
  }
  fn foo(prefix: &Option<String>, self_: &mut Parser) ->
↪  ReturnFoo {
    let ns = match (*prefix) {
      None => self_.namespace_for_prefix(""),
      Some(ref pre) => match self_.namespace_for_prefix(&pre) {
        None => return ReturnFoo::Return(self_.error("...")),
        ns => ns,
      },
    };
    ReturnFoo::Expr(ns)
  }
}
enum ReturnFoo {
  Expr(Option<String>),
  Return(Result<Option<Event>, ParserError>)
}
```

Listing 48: Extract method on parser.rs (range 11242:11568)

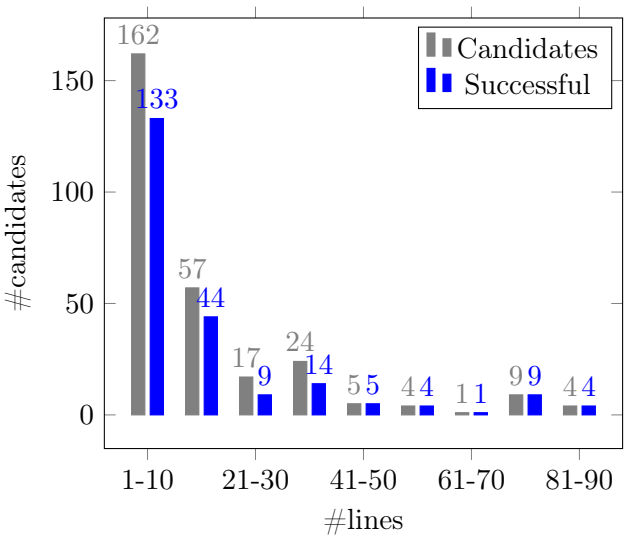

Figure 6.2: Extract Method in tokenizers

Summary of Box field	RustyXML	tokenizers
Candidates found:	34	132
Successful refactorings:	30	105
Internal errors:	1	23
Introduced Rustc error:	3	4
Introduced unit test failure:	0	0
Total duration:	33s	17m 11s
Time spent compiling and refactoring:	16s	3m 18s
Unit tests duration:	17s	13m 53s
Failed at: Box field		
Introduced internal errors:	1	23
Introduced Rustc (no code):	0	1
Introduced Rustc error [E0063]:	3	0
Introduced Rustc error [E0204]:	0	1
Introduced Rustc error [E0507]:	0	2

Table 6.3: Results of the Box Field experiments.

Before refactoring

```
match e {
Event::ElementStart(StartTag {
    name
    // ..
  }) => {
    let mut elem = Element {
      name,
      // ..
```

After refactoring

```
match e {
Event::ElementStart(StartTag {
    name
    // ..
  }) => {
    let mut elem = Element {
      (*name),
      // ..
```

Listing 49: Box field. Invalid refactoring.

6.3 Threats to validity

The correctness of the implemented refactorings was tested by recompiling and running unit tests. There might still be errors introduced by the refactorings in the experiments that we did not catch, as the test suite may be incomplete, and does not cover the whole semantics of the program. Test coverage was not measured due to the lack of tool support and time constraints.

A total of 1216 candidates were run for Extract Method, were many of these were short functions spanning less than 10 lines. It is possible that many of the succeeding cases are trivial ones. Software metrics and an improved candidate search could have better narrowed the set of candidates to a more realistic one, but we did not use software metrics in this thesis due to the lack of tool support and time constraints.

Some projects were excluded from the criteria specified in the introduction of this chapter, and the correctness may be lower for other projects. Programs are structured differently due to developers using different paradigms, such as object-oriented, functional and procedural programming, or different use of Rust language constructs, such as lifetimes and macros for instance. The paradigms, may to some degree, be indicated by the grouping of candidates by line number in Figures 6.1 and 6.2, but we did not investigate this further due to time constraints.

The performance is expected to depend on the compile time of the package or workspace being refactored as we recompile all local crates to ensure that `rustc` is invoked on all crates. Projects containing more crates or code than those in the experiments are therefore expected to have a performance which is worse.

Chapter 7

Conclusion

In this chapter, we will summarize the results of this thesis and describe some possible future work.

In the thesis we adapted the Extract Method refactoring for Java by Schäfer to Rust. Here we identified a Rust specific problem, lifetimes, which we handled correctly. As for the micro refactorings, we added Pull Up Item Declarations which reorganizes statements in a block. In Extract Block we used the `Block` construct to pass out variables instead of altering the placement of declarations. Introduce Anonymous Closure handled the control flow expressions by introducing special `enum` values. Close Over Variables had to add annotations for borrowing and mutating variables, in addition to closing the closure over variables. Convert Closure to Function was added, as Rust has support for local function declarations. Lift Item Declarations was added to move item declarations upwards in the AST. Lift Function Declaration moves a local function upwards to the closest `Impl`-block.

Based on Fowlers Extract Class for Java and a git commit found on the Rust Language GitHub repository we developed algorithms for Box Field and Unbox Field where the type of the field of a `struct` were changed from `T` to `Box<T>`.

The algorithms for Pull Up Item Declarations, Extract Block, Introduce Closure, Close Over Variables, Convert Closure to Function and Lift Function Declaration were implemented and composed to the Extract Method refactoring from Chapter 3 and Box Field from Chapter 4 were implemented. A Command Line Interface tool was developed, which allowed the refactorings to be invoked on Rust projects, where it handled the occurrence of multiple crates in a project. When we ran Extract Method on two projects and 79% of the refactorings were successful. For Box Field, 81% of the refactorings were successful. This is an indication that the algorithms have a high degree of correctness, and that the implementation can be used as a development aid while expecting it to introduce compile time errors in roughly 20% of the cases and changing the run time semantics in few of the cases. The performance of Extract Method, using almost 8 seconds on average in one project is not satisfactory as for using it as a development aid in an IDE, but here we are limited by the performance of the compiler.

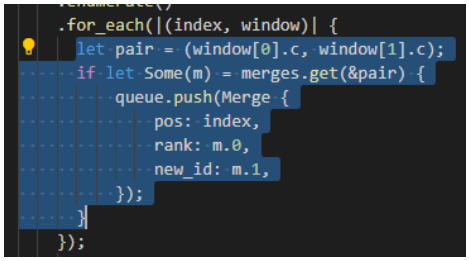

Figure 7.1: The selection

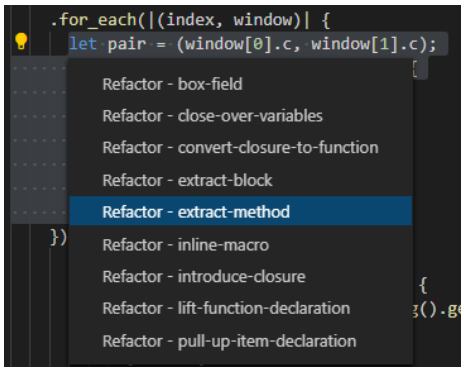

Figure 7.2: The implemented refactorings

```
.for_each(|(index, window)| {
  ({ Self::foo})(index, merges, &mut queue, window)
});
```

Figure 7.3: The new method invocation

We implemented a server and client communicating over the Language Server Protocol. The client was developed specifically for Visual Studio Code and it made the refactorings available for the user.

Here we show screenshots of an application of the Extract Method refactoring. First, in Figure 7.1, the selection is marked in the user interface. Then, in Figure 7.2, the available refactorings are shown in a context menu either by invoking the refactoring menu item, or by clicking the light bulb. After selecting "Refactor - extract-method", the client sends a `workspace/executeCommand` request to the server containing the name of the refactoring, "Extract Method", the name of the file, and the range of the selection within that file. The server then executes the refactoring CLI and based on the output, it then sends multiple `workspace/applyEdit` requests back to the server which contains the file modifications done by the refactoring. The new method invocation is shown in Figure 7.3 and the new function declaration is shown in Figure 7.4.

```
fn foo(index: usize,
  merges: &HashMap<(u32, u32), (u32, u32)>,
  queue: &mut BinaryHeap<Merge>,
  window: &[Symbol]) {
  let pair = (window[0].c, window[1].c);
  if let Some(m) = merges.get(&pair) {
    (*queue).push(Merge {
        pos: index,
        rank: m.0,
        new_id: m.1,
    });
}}
```

Figure 7.4: The new function declaration

Future work The refactorings could be made more precise, supporting the "?" expression which conditionally returns when evaluating `Result` and `Option` values, generic parameters and lifetime parameters on new functions.

For the candidate search for Extract Method, we used all subsequences of blocks. This candidate search could be improved using software metrics, such as the cyclomatic complexity to better identify refactorings that are useful, so that a refactoring is only suggested when it improves the quality of the source code.

With an improved candidate search, the refactoring could be automated, where a service listens to a source code repository and suggests good refactorings that it finds as pull requests, where the developers chooses to apply refactorings based on pull requests. The `CodeActions` that the language server sends to the client could benefit from a candidate search.

The Extract Method refactoring could be adapted to support concurrent programs. The `std::future::Future` API and the `async/await` syntax were made stable in versions 1.36 and 1.39 for Rust.

Bibliography

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Pearson Education, Inc, 2006.

[2] Kent Beck. *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[3] Luca Cardelli and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism". In: *Computing Surveys* 17.4 (1985), pp. 471–522.

[4] Alistair Cockburn and Jim Highsmith. "Agile Software Development: The People Factor". In: *Computer* 34.11 (Nov. 2001), pp. 131–133.

[5] The Rust Analyzer Developers. *Rust Analyzer*. URL: https://github.com/rust-analyzer/rust-analyzer.

[6] The Rust Project Developers. *Clippy*. URL: https://github.com/rust-lang/rust-clippy.

[7] The Rust Project Developers. *Rerast*. URL: https://github.com/rust-lang/rustfmt.

[8] The Rust Project Developers. *The Rust Programming Language*. URL: https://www.rust-lang.org (visited on 11/26/2018).

[9] The Rust Project Developers. *What is ownership?* URL: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

[10] Anna Maria Eilertsen. "Making Software Refactorings Safer". MA thesis. Dept. of Informatics, University of Bergen, 2016. URL: http://www.uib.no/fg/put/101340/safer-refactorings-assertions.

[11] Anna Maria Eilertsen, Anya Helene Bagge, and Volker Stolz. "Safer Refactorings". In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2016, pp. 517–531.

[12] Torbjörn Ekman et al. *Refactoring bugs, 2008*. URL: http://progtools.comlab.ox.ac.uk/refactoring/bugreports.

[13] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[14] William G. Griswold. “Program Restructuring as an Aid to Software Maintenance”. PhD thesis. University of Washington, 1991.

[15] William G. Griswold and William F. Opdyke. “The Birth of Refactoring: A Retrospective on the Nature of High-Impact Software Engineering Research”. In: *IEEE Software* 32.6 (2015), pp. 30–38.

[16] JetBrains. *IntelliJ Rust*. URL: https://intellij-rust.github.io/features/.

[17] Stephen C. Johnson. *Lint, a C Program Checker*. URL: https://wolfram.schneider.org/bsd/7thEdManVol2/lint/lint.pdf.

[18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018. URL: https://doc.rust-lang.org/book/index.html.

[19] Eugene Kohlbecker et al. “Hygienic Macro Expansion”. In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP ’86. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1986, pp. 151–161.

[20] Erlend Kristiansen. “Automated Composition of Refactorings”. MA thesis. Dept. of Informatics, University of Oslo, 2014. URL: https://www.duo.uio.no/handle/10852/42078.

[21] Erlend Kristiansen and Volker Stolz. “Search-based composed refactorings”. In: *27th Norsk Informatikkonferanse, NIK 2014, Høgskolen i Østfold, Fredrikstad, Norway, November 17-19, 2014*. Bibsys Open Journal Systems, Norway, 2014.

[22] David Lattimore, Dale Wijnand, and Kornel. *Rerast*. URL: https://github.com/google/rerast.

[23] Thomas J. McCabe. “A Complexity Measure”. In: *IEEE Trans. Softw. Eng.* 2.4 (July 1976), pp. 308–320.

[24] Flávio Medeiros et al. “A Catalogue of Refactorings to Remove Incomplete Annotations”. In: *J. UCS* 20.5 (2014), pp. 746–771.

[25] Microsoft. *Language Server Protocol*. URL: https://microsoft.github.io/language-server-protocol/.

[26] William F. Opdyke. “Refactoring Object-Oriented Frameworks”. PhD thesis. University of Illinois at Urbana-Champaign, 1992.

[27] SonarSource S.A. *Cognitive Complexity*. URL: https://blog.sonarsource.com/cognitive-complexity-because-testability-understandability.

[28] Max Schäfer et al. “Stepping Stones over the Refactoring Rubicon”. In: *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*. Genoa. Italy: Springer-Verlag, 2009, pp. 369–393.