Automated Refactoring of Rust Programs

Algorithms and Implementations of Extract Method and Box Field

Per Ove Ringdal

University of Oslo

June 29, 2020

1. Introduction
2. Refactoring: Extract Method
   - Extract Block
   - Introduce Anonymous Closure
   - Close Over Variables
   - Convert Anonymous Closure to Function
   - Lift Function Declaration
3. Refactoring: Box Field
4. Experiments & Demo
5. Summary

Refactoring

What is a refactoring?

a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. [1]

Refactoring

What is a refactoring?

a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. [1]

Microrefactoring

. . . this approach allows a very fine-grained decomposition of the overall refactoring into a series of micro-refactorings that can be understood, implemented, and tested independently. [2]

Rust

- Announced in 2010
- Ownership model
- Hygienic macros

Rust

- Announced in 2010
- Ownership model
- Hygienic macros

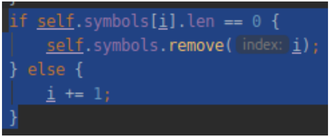
Ownership

memory is managed through a system of ownership with a set of rules that the compiler checks at compile time [3]

Refactoring Rust

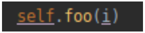
- Rust is a new language, little support in IDEs
- Data flow is changed with IntelliJs Extract Method in the example below

Before refactoring

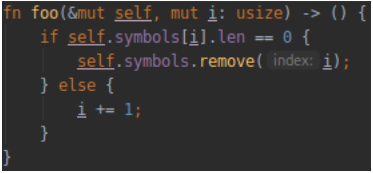


```
if self.symbols[i].len == 0 {
    self.symbols.remove( index: i);
} else {
    i += 1;
}
```

After refactoring



```
self.foo(i)
```



```
fn foo(&mut self, mut i: usize) -> () {
    if self.symbols[i].len == 0 {
        self.symbols.remove( index: i);
    } else {
        i += 1;
    }
}
```

Extract Method Composition

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 Composition

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

Extract Method Composition

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

Extract Block

Definition

Converts one or more `Statements` into a `Block`

Extract Block

Definition

Converts one or more `Statements` into a `Block`

Challenges

- Name Binding
  - `ItemDeclarations` should not occur inside (precond.)
  - `let`-declarations added before the new `Block`
- Ownership
  - Passing out value preserves the lifetime

Extract Block - Example

Before refactoring

```
fn bar() {
    let (mut i,j) = (0,1);
    i += 1;
    let sum = i + j;
    print!("{}", sum);
}
```

Extract Block - Example

Before refactoring

```
fn bar() {
  let (mut i,j) = (0,1);
  i += 1;
  let sum = i + j;
  print!("{}", sum);
}
```

After refactoring

```
fn bar() {
  let (mut i,j) = (0,1);
  let sum =
  {
    i += 1;
    let sum = i + j;
    sum
  };
  print!("{}", sum);
}
```

Introduce Anonymous Closure

Definition

Converts a `Block` to a `ClosureExpression`

Introduce Anonymous Closure - Example

Before refactoring

```
fn bar() {
  let (mut i,j) = (0,1);
  let sum =
  {
    i += 1;
    let sum = i + j;
    sum
  };
  print!("{}", sum);
}
```

Introduce Anonymous Closure - Example

Before refactoring

```
fn bar() {
  let (mut i,j) = (0,1);
  let sum =
  {
    i += 1;
    let sum = i + j;
    sum
  };
  print!("{}", sum);
}
```

After refactoring

```
fn bar() {
  let (mut i,j) = (0,1);
  let sum =
  (|| {
    i += 1;
    let sum = i + j;
    sum
  })();
  print!("{}", sum);
}
```

Introduce Anonymous Closure - Challenges

Definition

Converts a `Block` to a `ClosureExpression`

Introduce Anonymous Closure - Challenges

Definition

Converts a `Block` to a `ClosureExpression`

Challenges

- Control Flow
  - Cannot `break` or `continue` outside a closure.
  - A `return`-expression stops executing the current closure/function.

Introduce Anonymous Closure - Challenges

Definition

Converts a `Block` to a `ClosureExpression`

Challenges

- Control Flow
  - Cannot `break` or `continue` outside a closure.
  - A `return`-expression stops executing the current closure/function.

Solution

- Replace `break`, `continue` and `return`-expressions with `return`-expressions and handle them outside the closure.

Introduce Anonymous Closure - Control Flow Example

Before refactoring

```
fn foo() -> i32 {
 let sum =
  {
   let sum = i + j;
   if sum < 0 {
     return 0;
   }
   sum
  };
  return sum;
 }
}
```

Introduce Anonymous Closure - Control Flow Example

Before refactoring

```
fn foo() -> i32 {
 let sum =
  {
   let sum = i + j;
   if sum < 0 {
     return 0;
   }
   sum
  };
  return sum;
 }
}
```

After refactoring

```
fn foo() -> i32 {
 let sum =
 match (|| {
  let sum = i + j;
  if sum < 0 {
    return Rv::Return(0);
  }
  Rv::Expr(sum)
 })() {
  Rv::Expr(val) => val,
  Rv::Return(val) => return val
 };
 return sum;
}
```

Close Over Variables

Definition

Eliminates references to local variables declared outside a closure

Close Over Variables

Definition

Eliminates references to local variables declared outside a closure

Challenges

- Data Flow
  - Pass by reference / value
- Inference
  - `TupleIndexingExpression` and `FieldAccess` require type annotation when the variable is placed in the parameter list
  - Lifetimes aren't inferred when types are annotated in the parameter list

Close Over Variables - Example

Before refactoring

```
fn bar() {
  let (mut i,j) = (0,1);
  let sum =
  (|| {
    i += 1;
    let sum = i + j;
    sum
  })();
  print!("{}", sum);
}
```

Close Over Variables - Example

Before refactoring

```
fn bar() {
  let (mut i,j) = (0,1);
  let sum =
  (|| {
    i += 1;
    let sum = i + j;
    sum
  })();
  print!("{}", sum);
}
```

After refactoring

```
fn bar() {
  let (mut i,j) = (0,1);
  let sum =
  (|i: &mut i32, j: i32| {
    (*i) += 1;
    let sum = (*i) + j;
    sum
  })(&mut i, j);
  print!("{}", sum);
}
```

Convert Anonymous Closure to Function

Definition

Converts a `ClosureExpression` to a `FunctionDeclaration`

Convert Anonymous Closure to Function - Example

Before refactoring

```
fn bar() {
  let (mut i,j) = (0,1);
  let sum =
  (|i: &mut i32, j: i32| {
    (*i) += 1;
    let sum = (*i) + j;
    sum
  })(&mut i, j);
  print!("{}", sum);
}
```

Convert Anonymous Closure to Function - Example

Before refactoring

```
fn bar() {
  let (mut i,j) = (0,1);
  let sum =
  (|i: &mut i32, j: i32| {
    (*i) += 1;
    let sum = (*i) + j;
    sum
  })(&mut i, j);
  print!("{}", sum);
}
```

After refactoring

```
fn bar() {
  let (mut i,j) = (0,1);
  let sum =
  ({
    fn foo(i: &mut i32,
      j: i32) -> i32 {
      (*i) += 1;
      let sum = (*i) + j;
      sum
    }
    foo
  })(&mut i, j);
  print!("{}", sum);
}
```

Lift Function Declaration

Definition

Moves a local `FunctionDeclaration` upwards to the closest `impl`- or `mod`-`Block`

Lift Function Declaration

Definition

Moves a local `FunctionDeclaration` upwards to the closest `impl`- or `mod`-`Block`

Challenges

- Item Bindings
  - Item bindings in the `FunctionDeclaration` should be resolved to the target `mod`-`Block` or higher.
  - The new `FunctionDeclaration` should have a fresh identifier

Lift Function Declaration - Example

Before refactoring

```
impl Baz {
  fn bar() {
    let (mut i,j) = (0,1);
    let sum =
    ({
      fn foo(i: &mut i32,
        j: i32) -> i32 {
        (*i) += 1;
        let sum = (*i) + j;
        sum
      }
      foo
    })(&mut i, j);
    print!("{}", sum);
  }
```

Lift Function Declaration - Example

Before refactoring

```
impl Baz {
  fn bar() {
    let (mut i,j) = (0,1);
    let sum =
    ({
      fn foo(i: &mut i32,
        j: i32) -> i32 {
        (*i) += 1;
        let sum = (*i) + j;
        sum
      }
      foo
    })(&mut i, j);
    print!("{}", sum);
  }
```

After refactoring

```
impl Baz {
  fn bar() {
    let (mut i,j) = (0,1);
    let sum =
    ({ Self::foo
    })(&mut i, j);
    print!("{}", sum);
  }
  fn foo(i: &mut i32,
    j: i32) -> i32 {
    (*i) += 1;
    let sum = (*i) + j;
    sum
  }
}
```

Extract Method - Summary

Before refactoring

```
impl Baz {
  fn bar() {
    let (mut i,j) = (0,1);
    i += 1;
    let sum = i + j;
    print!("{}", sum);
  }
}
```

Extract Method - Summary

Before refactoring

```
impl Baz {
  fn bar() {
    let (mut i,j) = (0,1);
    i += 1;
    let sum = i + j;
    print!("{}", sum);
  }
}
```

After refactoring

```
impl Baz {
  fn bar() {
    let (mut i,j) = (0,1);
    let sum =
      ({Self::foo})(&mut i, j);
    print!("{}", sum);
  }
  fn foo(i: &mut i32,
    j: i32) -> i32 {
    (*i) += 1;
    let sum = (*i) + j;
    sum
  }
}
```

Box Field

- Based on a commit at the Rust Language repository

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

Box Field

- Based on a commit at the Rust Language repository
- Similar to Extract Class with one field and an existing target class

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

Box Field

- Based on a commit at the Rust Language repository
- Similar to Extract Class with one field and an existing target class
- It does not improve structure, but it may improve performance

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

Box Field

- Based on a commit at the Rust Language repository
- Similar to Extract Class with one field and an existing target class
- It does not improve structure, but it may improve performance
- Reduced instruction count by 2.6% [1]

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

Box Field

Definition

Adds the `Box` type to a field of a `struct`

Box Field

Definition

Adds the `Box` type to a field of a `struct`

Preconditions

- The `struct` should not have the `Copy` trait
- The field should not already be of type `Box`

Box Field

Definition

Adds the `Box` type to a field of a `struct`

Preconditions

- The `struct` should not have the `Copy` trait
- The field should not already be of type `Box`

Challenges

- Update any occurrences of the field to reflect the new layout
  - `StructExpressions`
  - `FieldAccessExpressions`
  - `StructPatterns`
- Builtin `#[derive]` macros are frequently used

Box Field - StructExpr and FieldAccess Example

Before refactoring

```
struct Foo {
  field: i32
}
fn bar () {
  let mut foo = Foo {
      field: 0
  };
  foo.field += 1;
}
```

Box Field - StructExpr and FieldAccess Example

Before refactoring

```
struct Foo {
  field: i32
}
fn bar () {
  let mut foo = Foo {
      field: 0
  };
  foo.field += 1;
}
```

After refactoring

```
struct Foo {
  field: Box<i32>
}
fn bar () {
  let mut foo = Foo {
    field: Box::new(0)
  };
  (*foo.field) += 1;
}
```

Box Field - Patterns Example

Before refactoring

```
struct Foo {
  field: i32
}
fn bar () {
  match foo {
    Foo { field } => {
      print!("{}",
        field);
    }
  }
}
```

Box Field - Patterns Example

Before refactoring

```
struct Foo {
  field: i32
}
fn bar () {
  match foo {
    Foo { field } => {
      print!("{}",
        field);
    }
  }
}
```

After refactoring

```
struct Foo {
  field: Box<i32>
}
fn bar () {
  match foo {
    Foo { field } => {
      print!("{}",
        (*field));
    }
  }
}
```

Experiments

- Implemented refactorings using the `rustc` library, and a CLI to invoke them
- Developed a tool that finds all candidates, attempts refactorings one by one, and runs unit tests after.
- Ran the experiments on two projects (RustyXML[2] and tokenizers[3])
- Candidates for Extract Method are all subsequences of `Blocks`.

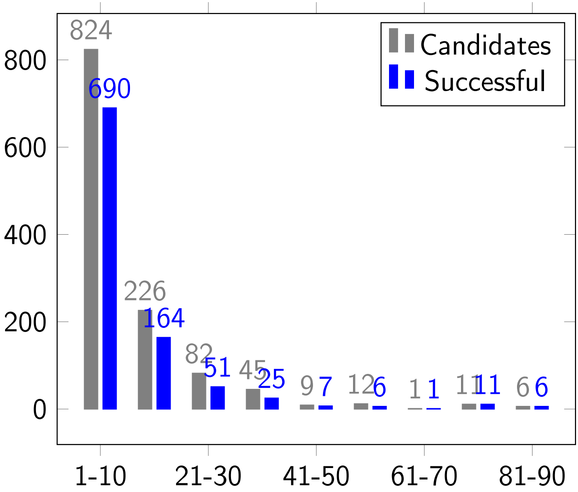
[2] https://github.com/Florob/RustyXML

[3] https://github.com/huggingface/tokenizers

Extract Method - Result

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

Extract Method - Result Grouped by Number of Lines



Box Field - Result

The candidates for Box Field are all fields of `struct` declared in the package.

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

Demo

- A client and server was developed that communicated over the Language Server Protocol.
- The client was for Visual Studio Code.

Demo

Summary

- Adapted the microrefactorings in Extract Method, with new and modified steps for Rust
- Developed Box Field, a specialization of Extract Class
- Experiments
    - Extract Method: 79% success
    - Box Field: 81% success
- Client and server communicating over LSP

Future Work

- More precise
  - Error Propagation “?”
  - Generic Parameters
  - Liftetime Parameters
- Improved candidate search - Should improve quality
- Automated refactoring
- Concurrent programs (Futures and async/await)

References

- Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- 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.
- The Rust Project Developers. *What is ownership?* URL: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.