Abstract

Refactoring – the technique to improve the internal structure of a program – has become a widely adopted practice among software engineers, but manual refactoring is tedious and error prone.

The Scala programming language is supported on all major Java development platforms, but most do not yet assist the programmer with automated refactoring tools.

This project provides an IDE independent library to create automated refactorings for Scala. A refactoring is essentially a transformation of the abstract syntax tree. The library makes writing such transformations as simple as possible: combinators can be used to build complex transformations from basic ones. Deriving the concrete source code changes from these converted trees is handled transparently by the library.

Several refactorings have been implemented on top of the library, along with the integration into the Scala IDE for Eclipse: Rename, Extract Local, Extract Method, Inline Local and Organize Imports.

Management Summary

In this thesis, we describe the development of a refactoring tool for the Scala programming language, conducted at the Institute for Software at the University of Applied Sciences Rapperswil. This master's thesis is a continuation of a previous term project by the same author.

Motivation

Refactoring means to improve the internal structure of a program while keeping its external behavior. Improving a program's internal structure can be achieved in various ways: the names that are used internally can be changed to better reflect their functionality, or the code can be reorganized to make the program easier to extend, read, comprehend, and test.

Refactoring does not have to be done with a specific tool, nor is it limited to a certain language or technology. Most integrated development environments support the developer with automated refactorings. Having such support reduces the time and therefore the hurdle to apply a refactoring; automation is also less error-prone than doing the same operations manually.

Scala is a modern programming language developed by Martin Odersky and his team at EPFL. Scala combines various aspects from object oriented and functional programming models. While it supports the developers with many powerful features, it is still fully compatible with code written in Java, allowing projects to mix Scala and Java.

Scala is an impressive language, but if it wants to become widely used in enterprises, it also needs to provide tools, including integrated development environments (IDEs). There already exist several Scala IDEs, but their refactoring support is still very limited.

Goals

The primary goal of this thesis is to support Scala IDEs with automated refactoring tools. The refactoring functionality is offered in the form of a library, so it can be integrated into and shared among different IDEs and other tools that want to refactor Scala code. To demonstrate the implemented refactorings, the library has to be integrated into the Eclipse based Scala IDE.

A second goal is to make the creation of new automated refactorings as simple as possible, to enable interested developers to implement their own refactorings.

Results

We have developed a library that builds on the Scala compiler and contains everything that is needed to create automated refactorings for Scala. The following refactorings have been implemented:

Rename for all the names that are used in the source code.

Extract Method to extract a selection of statements into a new method.

Extract Local to introduce a new local variable for an existing expression.

Inline Local to replace references to a local variable with its right hand side.

Organize Imports to clean up the imported dependencies of a source file.

These refactorings are all fully integrated into the Scala IDE for Eclipse, along with an online help that explains the usage of each refactoring.

To help new refactoring implementors getting started, this report documents not only the internals of the library but also the detailed implementation of the refactorings as well as how-tos and guides on how new refactorings can be written and integrated into IDEs or other tools.

The implemented refactorings are already part of the current development builds of the Scala IDE for Eclipse and have been presented at the first Scala conference – Scala Days 2010 [Sto10a].

Declaration of Authorship

I, Mirko Stocker, declare that this thesis and the work presented in it is my own, original work. All the sources I consulted and cited are clearly attributed. I have acknowledged all main sources of help.

Location, Date: .....

Signature: .....

Contents

1. Introduction **1**
1.1. Refactoring . . . 1
1.2. Scala . . . 1
1.3. Integrated Development Environments . . . 2
1.4. Thesis Goals . . . 2
1.5. Contents of This Report . . . 4
1.6. Target Audience . . . 5

2. Refactoring Library **7**
2.1. Overview . . . 8
2.2. Analysis . . . 11
2.2.1. Symbols . . . 11
2.2.2. Refactoring Index Interface . . . 12
2.2.3. Default Index Implementation . . . 12
2.2.4. Resolving References . . . 12
2.2.5. Tree Analysis . . . 14
2.2.6. Name Validation . . . 16
2.3. Transformation . . . 18
2.3.1. Transformations . . . 18
2.3.2. Combinators . . . 20
2.3.3. Traversal . . . 21
2.3.4. Creating Trees . . . 25
2.3.5. Tree Transformations . . . 26
2.4. Source Generation . . . 28
2.4.1. Modification Detection . . . 28
2.4.2. Code Generation . . . 29
2.4.3. Using the Source Generator . . . 35
2.4.4. Comparison With the Term Project . . . 36

3. Implemented Refactorings **39**
3.1. Rename . . . 40
3.1.1. Features . . . 40
3.1.2. Implementation Details . . . 42
3.1.3. Limitations . . . 42

3.2. Organize Imports . . . 43
3.2.1. Features . . . 43
3.2.2. Limitations . . . 45
3.3. Extract Local . . . 45
3.3.1. Features . . . 46
3.3.2. Implementation Details . . . 48
3.3.3. Limitations . . . 49
3.4. Inline Local . . . 50
3.4.1. Examples . . . 51
3.4.2. Implementation Details . . . 51
3.5. Extract Method . . . 53
3.5.1. Features . . . 53
3.5.2. Implementation Details . . . 54
3.5.3. Examples . . . 54
3.5.4. Limitations . . . 58

4. Tool Integration **61**
4.1. Dependencies . . . 61
4.2. Integrating the Library . . . 61
4.3. Scala IDE for Eclipse Integration . . . 65
4.3.1. Integrating with Eclipse LTK . . . 65
4.3.2. Interfacing with the Scala IDE . . . 67
4.3.3. A Concrete Example . . . 68
4.3.4. Adding New Refactorings . . . 69

5. Testing **71**
5.1. Compiling Test Code . . . 71
5.2. Creating a Project Layout . . . 72
5.3. Implementation . . . 73

6. Conclusion **75**
6.1. Accomplishments . . . 75
6.2. Future Work . . . 76
6.3. Acknowledgments . . . 76

A. Project Environment **79**
A.1. Tools . . . 79
A.2. Time Report . . . 79
A.3. Project Plan . . . 80

B. User Guide **83**
B.1. Rename . . . 83
B.1.1. Limitations . . . 84

B.2. Organize Imports 85
B.2.1. Limitations 85
B.3. Extract Local 86
B.4. Inline Local 87
B.5. Extract Method 87
B.5.1. Limitations 88

C. Developer How-To **89**
C.1. Introduction 89
C.2. The Example 89
C.3. Implementing It 90
C.4. The Result 94

D. Scala AST **95**
D.1. Base Classes and Traits 95
D.2. Concrete Trees 98
D.3. Other AST Constructs 111

E. Advanced Scala Features **115**
E.1. Path Dependent Types 115
E.2. Stackable Traits 116
E.3. Implicit Conversions 118
E.4. Self Type Annotation 119
E.5. Package Nesting 119

F. License **121**

Bibliography **123**

1. Introduction

The goal of this project is to provide Scala developers with automated refactoring tools. This master's thesis is a continuation of a foregoing term project (see [Sto09]) at the University of Applied Sciences Rapperswil, Switzerland.

In this chapter, we will briefly introduce the Refactoring technique and the Scala programming language, as well as explain the goals and motivation of this thesis.

1.1. Refactoring

Refactoring of programs is a well established practice among professional software developers. In his 1992 PhD thesis [Opd92], William Opdyke defined refactoring as

> a set of program restructuring operations (refactorings) that support the design, evolution and reuse of object-oriented application frameworks.

The breakthrough in industry started in 1999, when Martin Fowler and his colleagues published their popular book *Refactoring: Improving the Design of Existing Code* [Fow99], where refactoring is defined as

> the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

Today, refactoring has been absorbed by the programming mainstream, and is usually well integrated into the developer's work-flow and development environment. Developers use refactoring tools to keep their code maintainable by applying refactorings such as Rename to quickly change identifiers. In agile environments, where software is rapidly adapted to handle new requirements, performing refactorings regularly is essential to get reusable code and to keep up with the pace of change.

Refactoring as a technique does not mandate a tool nor depend on a specific programming language.

1.2. Scala

The Scala programming language [OSV08], developed by Martin Odersky and his team at EPFL, is a statically typed, compiled language that runs on the Java Virtual Machine (or on .NET alternatively [EPF08]) and excels with its unique combination

of object-oriented and functional programming concepts. Odersky also calls Scala a postfunctional language because it has been designed "to make functional constructs, imperative constructs, and objects all play well together" [Ode10].

One of Scala's strengths is its seamless interoperability with Java on the class level: Scala classes can extend Java classes and vice-versa. Scala also does not ship with a large standard library but uses existing Java classes where it is sensible.

Scala provides all of Java's object-oriented features but does away with the not really object oriented ones like primitive data types and static class members. Scala also provides code reuse via traits; a kind of interface that may contain implementations.

From functional programming, Scala has absorbed functions as first class values and embraces the idea of immutability with various language constructs. Scala even supports lazy evaluation through call-by-name parameters and the `lazy` modifier for values. A combination from both object-oriented and functional worlds can be seen in Scala's ability to use pattern matching to deconstruct objects while still preserving encapsulation.

These were just a few examples of how Scala differs from other languages such as Java. One last feature worth mentioning is that in Scala, building your own abstractions and control structures is easy, which is the reason why it has been named the "scalable language". For a short introduction and a tutorial, see [SH09] and [Ode09b].

1.3. Integrated Development Environments

Many programmers, particularly of mainstream languages such as Java and C#, use integrated development environments (IDE) to create their software. Notably the IDEs for the Java programming language excel with automated refactoring support; the screen-shots in Figure 1.1 on the following page show two examples. If Scala wants to cater to those programmers and become a viable alternative in enterprises, it needs to offer IDE support that is as comfortable to use and as mature as the existing Java tooling is.

Scala is supported on the three main Java development platforms Eclipse [Sab10], IntelliJ IDEA [ZP09], and NetBeans [Net09], but with the exception of IntelliJ IDEA – which offers a few refactorings – support for automated refactoring does not yet exist. Although a study by Emerson Murphy-Hill et al. among developers using Eclipse [MHPB09] indicates that many refactorings are not performed with the tool support but by hand, other automated refactorings like Rename, Move and Extract Method are used frequently.

1.4. Thesis Goals

The goal of this thesis is to support Scala IDEs with automated refactoring tools. It aims to provide a comprehensive catalog of refactorings and the necessary infrastructure

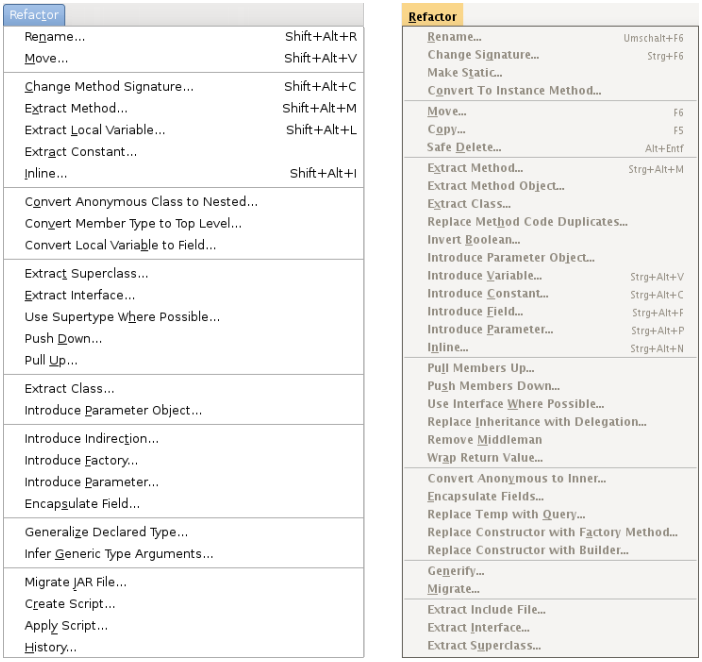


Figure 1.1.: Automated refactoring in Java IDEs

to create new refactorings. To maximize the number of IDEs and other tools that can profit from the project, it will provide an IDE independent refactoring library that only depends on the Scala compiler. IDEs can then seamlessly integrate this library by providing the user interface and interaction.

As many IDEs today are written in Java, integrating a Scala library is no problem. Also, because the majority of Scala IDEs are completely open source (NetBeans, Eclipse), having a single refactoring library allows their developers to cooperate on an implementation, not fragmenting the already scarce resources any further. As a showcase, this project provides the integration into the Eclipse based Scala IDE [Sab10].

Writing an automated refactoring is no trivial task, several things have to be taken care of: one has to analyze the source code, create an appropriate representation (e.g. abstract or concrete syntax tree) of the program, transform it and turn it back into plain source code.

The heart of a refactoring is the transformation or manipulation of the program representation; but often – from our experience with refactoring tools for languages like Ruby [CFS07], C++ [GZS07], and Groovy [KKKS08b] – the developer also has to provide the instructions how these manipulations affect the source code, or how the changes made to the AST are to be translated back into source code changes. This makes creating new refactorings unjustifiably more complex and is a high entry barrier for contributors. The Scala refactoring library tries to make creating new refactorings as simple as possible: code generation from the abstract syntax tree is completely transparent and needs almost no guidance from the refactoring writer.

Transformations of the program are based on the Scala compiler's own AST, and are written in a functional programming style that makes it possible to assemble complex transformations from simple ones using combinators.

To summarize, the Scala Refactoring project develops an IDE independent refactoring library that makes creating new refactorings as simple as possible.

1.5. Contents of This Report

This document is organized as follows: Chapter 2 on pages 7–37 explains the concepts and implementation of the refactoring library. The details of the implemented refactorings are described in Chapter 3 on pages 39–59. How these refactorings can be integrated into an IDE or other tool is the topic of Chapter 4 on pages 61–70. How the implemented refactorings are tested is explained in Chapter 5 on pages 71–74. Chapter 6 on pages 75–77 concludes this thesis with a review of the achievements and an outlook on further work.

The project environment is briefly explained in Appendix A on pages 79–82. The appendices also contain a user guide to the refactorings in Eclipse (Appendix B on pages 83–88), and a how-to introduction for developers that explains how a new refactoring can be created in Appendix C on pages 89–94. Developers that work with

Scala's AST might also be interested in Appendix D on pages 95–113, where the specific trees of the AST are described. Appendix E on pages 115–120 contains explanations of more advanced Scala features and is referenced where needed in this document. The source code of this thesis is released under the Scala license, which is printed in Appendix F on page 121

1.6. Target Audience

We assume that the reader knows the basic Scala concepts (if not, Scala by Example [Ode09b] is a good starting point) and is able to read Scala source code. Whenever more advanced or possibly confusing concepts are used, a reference to Appendix E will be provided.

Developers who want to use the library to transform Scala source code should start with Chapter 2 on page 7 on the library internals and Appendix D on pages 95–113 to learn more about Scala's AST.

To integrate the existing refactorings in a new tool, Chapter 4 on pages 61–70 shows how this can be done with a made up editor and how the integration into the Scala IDE for Eclipse looks like.

For those wishing to implement new refactorings, the how-to in Appendix C on pages 89–94 and Chapter 3 on pages 39–59 on the implemented refactorings can serve as a starting point. How the new refactoring can be tested is explained in Chapter 5 on pages 71–74.

Users who wish to provide accurate bug reports should take a look at Chapter 5 on pages 71–74 on testing to learn how a new test that points out a failure can be implemented.

2. Refactoring Library

The refactoring library is the heart of the Scala Refactoring project. It contains the means to analyze a Scala program, to modify it by transforming it, and to turn these modifications back into source code. When writing a refactoring, one usually has to take care of the following steps:

1. provide a user interface so that a specific refactoring can be discovered and invoked from the IDE.
2. analyze the program under refactoring to find out whether the refactoring is applicable and further to determine the parameters and constraints for the refactoring.
3. transform the program tree from its original form into a new – refactored – form according to the refactoring's configuration.
4. turn this new form back into source code, keeping as much of the original formatting in place as possible and to generate code for new parts of the program.
5. present the result of the refactoring to the user – typically in the form of a patch – and apply it to the source code.

From all these steps, the first and the last one are IDE-platform dependent and usually well supported (see Chapter 4 for details on Eclipse's refactoring support). For the remaining three, the refactoring library contains the necessary infrastructure to implement these steps.

The essence of a refactoring is a transformation that takes a program in some abstract form and changes its structure. To know what to transform, one has to analyze the program first. That we also have to turn a refactored program back into source code is a consequence of storing programs as plain text files, but not an essential part of a refactoring. Therefore, one of the design goals was to provide a generic implementation that can handle all kinds of changes without knowing exactly what the transformation changed.

In the remainder of this chapter, we shall first take a look at the architecture of the library and then describe each of the three main components in detail.

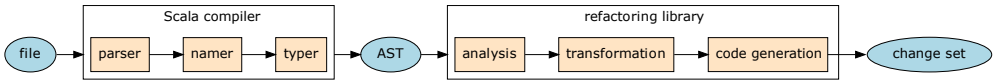


Figure 2.1.: The work-flow of the refactoring. The IDE uses the compiler to parse the source file and passes the resulting syntax tree to the refactoring tool. The result of a refactoring is a set of changes – a patch – that the IDE has to apply to the source files (adapted from [Sto09]).

2.1. Overview

Automated refactoring implementations typically do not work on the source code directly but – just as a compiler – do the majority of the work on the abstract syntax tree (AST) of the program. We do not create our own AST representation but reuse the Scala compiler's parser and type checker (as explained in Chapter 4, we also do not parse the code ourselves but get the AST from the IDE). This not only saved time during the development of the library, but also makes it easier to implement a new refactoring if one is already familiar with the Scala compiler's AST. Additionally, the Scala compiler also already provides some infrastructure to traverse and transform an AST.

Scala's AST is explained in more detail in Appendix D on page 95; a general knowledge of what an AST is should suffice to follow the explanations in this chapter. Useful to know is that all trees have a position information: either indicating a location from where the tree origins or a NoPosition, which denotes trees that do not have a corresponding source code location. This information is later used by the transformation and code generation phases.

As we have seen at the beginning of this chapter, a typical refactoring takes the current file's AST and the user's selection or caret position and first checks if the chosen refactoring is applicable – for example, whether the selected region of the source file corresponds to an AST element that can be handled by the chosen refactoring. If necessary, the refactoring queries additional configuration – for example, a new name – from the user. The AST is then transformed into its new form and handed over to the source generator to turn the AST back into source code (see Figure 2.1 for a visualization of this work-flow). As motivated above, generating the source code is already implemented generically and needs no further instructions from the refactoring implementer.

The architecture (see Figure 2.2 on the next page) and also the source code layout follow these three phases of the refactoring:

Analysis in package analysis contains the means to analyze the program and to build an index for the identifiers in the program. This will be explained further in Section 2.2 on page 11.

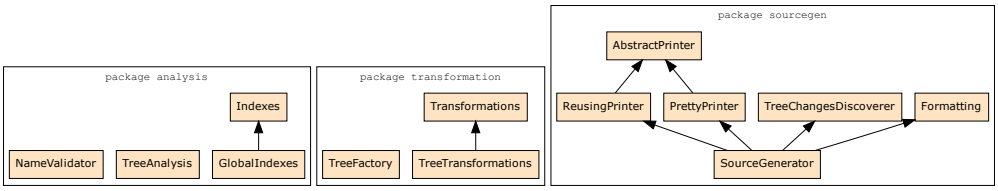


Figure 2.2.: An overview of the refactoring library architecture: the three main packages analysis, transformation, and sourcegen. Note that there exist more traits and classes in these packages – but for the sake of clarity, only the major ones are shown. The arrows stand for inheritance or mix-in composition.

Transformation in package transformation provides a framework to write, combine and apply transformations to trees, as well as factory methods to create new trees. How this works is described in Section 2.3 on page 18.

Source Generation in package sourcegen primarily contains the SourceGenerator that turns an AST back into change objects (i.e. patches) for the source code and is explained in Section 2.4 on page 28.

Classes that are shared between these three packages – for example, the Change class, custom exception classes and other utility traits – are all located in the common package. Two traits worth mentioning from common are Selections and PimpedTrees.

PimpedTrees contains several implicit conversions for the Scala compiler's trees that add useful functionality. For example, a namePosition method that returns the source code position of a tree's name. The trait also contains custom extractors and new tree subclasses that wrap things in trees that are not represented as such in the compiler – for example, modifiers:

```
case class ModifierTree(flag: Long) extends Tree {
...
}
object ModifierTree {
  def unapply(m: Modifiers) ...
}
```

This allows us to treat all these elements of the AST uniformly during source code generation.

The Selections trait contains an interface – Selection – and two implementations of the interface: TreeSelection and FileSelection. Once a selection has been created, it can be used to query the selected AST elements:

```
/**
 * Returns all selected trees that are not children of other selected trees.
 */
def selectedTopLevelTrees: List[Tree]

/**
 * Returns all symbols that are either used or defined in the selected trees and their children.
 */
def selectedSymbols: List[Symbol]

/**
 * Returns true if the given Tree is fully contained in the selection.
 */
def contains(t: Tree): Boolean

/**
 * Returns true if the given Tree fully contains this selection.
 */
def isContainedIn(t: Tree): Boolean

/**
 * Tries to find the selected SymTree: first it is checked if the selection fully contains a
 * SymTree, if true, the first selected is returned.
 * Otherwise, the result of findSelectedOfType[SymTree] is returned.
 */
def selectedSymbolTree: Option[SymTree]

/**
 * Finds a selected tree by its type. The tree does not have to be selected completely,
 * it is only checked whether this selection is contained in the tree.
 *
 * If multiple trees of the type are found, the last one (i.e. the deepest child) is returned.
 */
def findSelectedOfType[T](implicit m: Manifest[T]): Option[T]
```

This is used in most of the refactoring implementations to find selected trees or trees that surround the selection – for example, to find the enclosing class when extracting a method.

In the remainder of this chapter, the three library components analysis, transformation, and source generation will be explained in more detail.

2.2. Analysis

An important first step in each refactoring is to analyze the current program that is being refactored. For example, when doing a Rename Method refactoring, we need to resolve all references to the renamed method. A more complex example is Extract Method, where we need to perform data-flow analysis to determine the parameters and return values of the extracted method.

Our IDEs also analyze the program code in a similar way to make the life of the programmer easier: finding the declaration of a variable or listing all subtypes of a class are common operations.

2.2.1. Symbols

Our analyses heavily depend on the Scala compiler's AST and all the information it provides through the program's *symbols*. For example, each symbol has an owner that can be used to navigate the logical structure of the program. There are also almost one hundred isXY methods defined on the Symbol class that can be used to query information:

```
abstract class Symbol {
  ...
  def isAnonymousClass: Boolean
  def isConstructor: Boolean
  def isGetter: Boolean
  def isLocal: Boolean
  def isSubClass(that: Symbol): Boolean
  ...
}
```

All the trees that inherit from the SymTree trait provide a symbol instance. DefTrees usually introduce a new symbol and RefTrees reference a symbol introduced by a DefTree. The following illustration shows how symbols are related (not all symbols are colored – for example, the built in types have a symbol as well):

```
trait SuperClass {
  def strlen(str: String) = str.length
  def abstractMethod: Int
}

class SubClass extends SuperClass {
  def abstractMethod = 1 + strlen("1")
}
```

Note that the two abstractMethod symbols are not the same; but there are ways to find overridden and implemented methods in subclasses, as we shall see later.

While the trees can have a reference to a symbol, the converse is not true: symbols do not know about the trees they are related to. But for a refactoring which mainly works with trees, this information is crucial. This is why the refactoring library contains the means to build an index that relates symbols with corresponding trees.

2.2.2. Refactoring Index Interface

Indexing a complete project can be expensive, so ideally, the IDE would maintain the index and pass it to the refactoring library when needed, in the same way that the library does not compile the source files itself but gets the ASTs directly from the IDE.

The trait that needs to be implemented and that is used by the refactorings to query the index is shown in Figure 2.3 on the next page.

The library contains a default implementation of this trait that can be used if the IDE does not already maintain an index itself. This implementation is described in the following section.

2.2.3. Default Index Implementation

Building an index can be expensive: whenever a compilation unit in the program changes, references to the symbols from other compilation units can change, and also the other way around. Because of this, it is not wise to maintain one monolithic index that needs to be thrown away and recreated on every change in the program. The provided implementation avoids this by maintaining a simple data structure for each compilation unit and then combines these for queries:

CompilationUnitIndex One index per compilation unit that holds the references and declarations of just this part of the program. This structure can be rebuilt every time a compilation unit changes. Rebuilding it traverses the whole tree once and stores mappings from symbols to RefTrees and DefTrees.

GlobalIndex An implementation of the IndexLokup trait that ties together any number of these per compilation unit indices, but is completely stateless itself.

Whenever a compilation unit changes, just a single CompilationUnitIndex needs to be rebuilt and combined with the already existing ones into a new GlobalIndex.

2.2.4. Resolving References

Resolving the declaration tree of a symbol is an inexpensive lookup, but the reverse – finding all references – causes more work. In GlobalIndex, the process of finding all references is done in multiple steps: first, the symbol is *expanded* and second all references to these expanded symbols are collected.

```
trait IndexLookup {
  /**
   * Returns all defined symbols, i.e. symbols of DefTrees.
   */
  def allDefinedSymbols(): List[global.Symbol]

  /**
   * Returns all symbols that are part of the index, either referenced or defined. This also
   * includes symbols from the Scala library that are used in the compilation units.
   */
  def allSymbols(): List[global.Symbol]

  /**
   * For a given Symbol, tries to find the tree that declares it.
   */
  def declaration(s: global.Symbol): Option[global.DefTree]

  /**
   * For a given Symbol, returns all trees that directly reference the symbol. This does not
   * include parents of trees that reference a symbol, e.g. for a method call, the Select tree
   * is returned, but not its parent Apply tree.
   *
   * Only returns trees with a range position.
   */
  def references(s: global.Symbol): List[global.Tree]

  /**
   * For a given Symbol, returns all trees that reference or declare the Symbol.
   */
  def occurences(s: global.Symbol): List[global.Tree]

  /**
   * For the given Symbol – which can be a class or object – returns a list of all sub–
   * and super classes, in no particular order.
   */
  def completeClassHierarchy(s: global.Symbol): List[global.Symbol] =
    (s :: (allDefinedSymbols filter (_.ancestors contains s) flatMap (s => s :: s.ancestors))
      filter (_.pos != global.NoPosition) distinct)

}
```

Figure 2.3.: The index interface used by the library and the refactorings. Note that global is an instance of the compiler that is provided by an outer trait; see the Appendix E.1 on page 115 on path dependent types.

What do we mean by expanding a symbol? Consider the listing with the colored symbols we mentioned at the beginning of Section 2.2 where the implementing method defines a different symbol than the abstract declared method. Now when we want to find references, we need to collect all references to both symbols. The same is true for getters and setters: renaming a class parameter also needs to rename all usages of getters and setters. To do this, the index implementation uses so called SymbolExpanders to expand a symbol:

```
trait SymbolExpander {
  def expand(s: Symbol): List[Symbol] = List(s)
}
```

The SymbolExpander is used as a stackable trait (see Appendix E.2 on page 116) and is at the time of this writing implemented in the following variations:

ExpandGetterSetters connects getters, setters and the underlying field as well as constructor parameters.

SuperConstructorParameters resolves class parameters that are passed to a super constructor.

Companion to find the companion object or class for a symbol.

OverridesInClassHierarchy searches for a symbol in all sub- and super-classes. that might override or implement it.

The GlobalIndex uses all these traits, but it would also be possible for an implementation of the index to use only a subset of these to improve the performance.

Now when all references to a method need to be found, the initial symbol is run through all the symbol expanders until a fix point is reached. The graphic Figure 2.4 on the next page shows an example of the process works.

2.2.5. Tree Analysis

Besides the index to lookup references and declarations, some refactorings need more sophisticated analysis of the program. This section introduces the TreeAnalysis trait which contains these functionality.

Local Dependencies

The Extract Method refactoring extracts a selection of expressions into a new method. To do this, it needs to calculate all dependencies the selected expressions have to their enclosing scopes. Variables and functions that are not accessible from the new method location need to be passed as arguments, and program elements that are declared inside the extracted method and used outside of the selection need to be passed back.

In the following listing, the user wants to extract the selected expressions:

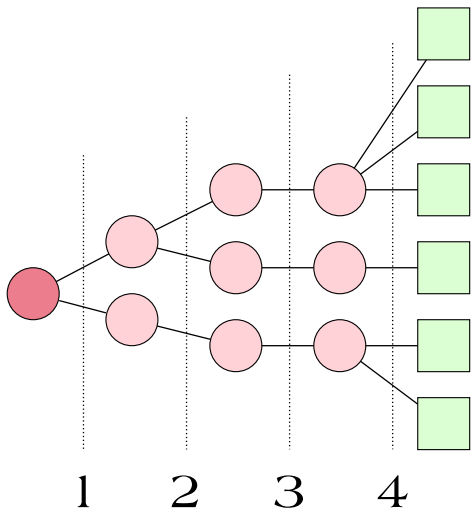


Figure 2.4.: An illustration how the symbol-expanding process works; circles represent symbols and squares trees. We start with a single symbol on the left – e.g. a class field – and in the first step, it is expanded to two symbols – for example because the class field has a getter method. We do another round of expansion and find yet another related symbol (the getter might be overridden in a subclass). The third expansion yields no new symbols, thus the fourth step concludes by collecting all references and declarations to these symbols.

```
def calculate {
  val sumList: Seq[Int] => Int = _ reduceLeft (_+_)
  val prodList: Seq[Int] => Int = _ reduceLeft (_*_)
  val values = 1 to 10 toList
  val sum = sumList(values)
  val product = prodList(values)

  println("The sum from 1 to 10 is "+ sum +".")
}
```

The refactoring has to create a method that takes the values and the sumList and prodList functions as arguments. Also, because the sum value is used in the originating method – but not product – it has to be returned from the new method.

The calculation of these inbound and outbound dependencies is done as follows:

Inbound: Starting with all symbols inside the selection, we filter all symbols that are declared in the current scope (e.g. the method we extract from) and remove all declarations that are defined inside the selection. This gives us all the inbound parameters.

Outbound: For each symbol that is defined inside the selection, check if it is used anywhere outside the selection.

The inboundLocalDependencies and outboundLocalDependencies methods implement these two calculations:

```
trait TreeAnalysis {

  self: Selections with Indexes =>

  val global: Global

  def inboundLocalDependencies(selection: Selection, currentOwner: global.Symbol):
    List[global.Symbol] = ...

  def outboundLocalDependencies(selection: Selection, currentOwner: global.Symbol):
    List[global.Symbol] = ...
}
```

2.2.6. Name Validation

When refactoring, one often has to introduce new names into the program that can potentially conflict with already existing names. The NameValidation trait contains

two methods: one to check whether a name is a valid identifier – based on the Scala compiler, and another method to check if a name will collide with an already existing name:

```
trait NameValidation {
  def isValidIdentifier(name: String): Boolean = ...
  def doesNameCollide(name: String, s: Symbol): List[Symbol] = ...
}
```

The doesNameCollide method takes a name and a symbol and returns a list of all symbols that collide with the given name in symbol's context.

2.3. Transformation

The core of every refactoring is a *transformation* that takes the current program in its abstract syntax tree form and transforms it into its refactored form. Such a transformation can be as simple as changing names – think of the Rename refactoring – or restructure large parts of the AST as in an Extract or Move refactoring.

Often, a larger refactoring comprises many smaller transformations. An illustrative example is the Extract Method refactoring, which can be assembled from three basic transformations:

Create Method to introduce a new (empty) method.

Copy Statements to copy the selected statements into the newly created method.

Replace Statements to replace the original statements that have been copied to the new method with a call to the new method.

The *replace* transformation itself is again a combination of two even more fundamental transformations: *insert* and *delete*. Once we have our Extract Method transformation, it can then again be combined with other transformations – for example into an Extract Class refactoring. It should be clear from this that the key to a reusable refactoring library lies in the composability of its transformations.

Conceptually, chaining simple transformations to build more powerful ones follows the Unix pipes philosophy. The design of this implementation was inspired by the Stratego program transformation tool-set [Str10] and the Kiama language processing library [Slo10]. Functional programming also uses the term *combinator* for functions that can be combined and yield new functions of the same kind. An example of this are parser combinators [MPO08], which are part of the Scala standard library.

In contrast to Unix pipes that operate on their input line by line, performing transformations on a tree data structure adds an additional dimension. When transforming trees, we are also concerned with questions on how we want to traverse the tree – i.e. pre-order or post-order – and to which children a transformation should be applied. The presented implementation handles all these concerns in a uniform way.

In the remainder of this section, we will develop the basics of the Scala refactoring's transformation combinators and show examples of their usage.

2.3.1. Transformations

A refactoring transformation is essentially a function that transforms a tree into another tree. But because most transformations do not apply to all kinds of possible trees, we model a transformation as a function of type `Tree ⇒ Option[Tree]`, making use of Scala's `Option` type to indicate the potential inability to transform. In the actual implementation, the transformations are implemented generically as a `Transformation[A, B]` that extend `A ⇒ Option[B]`:

```
abstract class Transformation[A, B] extends (A ⇒ Option[B]) {
  self ⇒

  def apply(in: A): Option[B]
...
}
```

The explicit self type annotation (see Appendix E.4 on page 119) will be used later in the implementation of the combinators. Note that all transformations are implemented polymorphically, but to make the explanations more clear, we will assume that they are used to transform trees.

Transformations can be created from partial functions using the transformation convenience function. As an example, we create a transformation that reverses the order of a class, trait, or object's member definitions and apply it to a given template instance.

```
def transformation[A, B](f: PartialFunction[A, B]) = new Transformation[A, B] {
  def apply(t: A): Option[B] = f lift t
}

val reverseTemplateMembers = transformation[Tree, Tree] {
  case t: Template ⇒ t copy (body = t.body.reverse)
}

val result: Option[Tree] = reverseTemplateMembers(template)
```

Now that we have a way to create single transformations, we need to be able to combine them. To do this in various ways, we introduce several combinators. We use a notational shortcut to denote transformations: $A \xrightarrow{t} [B]$ is a Transformation [A, B].

There also exist two basic transformations, one that always succeeds, returning its input unchanged, and one that always fails, independent of its input. Depending on the context, the alias id for succeed might be a better fit and is provided as well.

```
def succeed[A] = new Transformation[A, A] {
  def apply(a: A): Option[A] = Some(a)
}

def id[A] = success[A]

def fail[A] = new Transformation[A, A] {
  def apply(a: A): Option[A] = None
}
```

2.3.2. Combinators

There are several existing combinators already implemented in the library. On the right side of each paragraph, the symbolic or alphanumeric name and type of the transformation is shown.

Sequence $\&>: \ (A \xrightarrow{t} [B]) \Rightarrow (B \xrightarrow{t} [C]) \Rightarrow (A \xrightarrow{t} [C])$

Combines two transformations so that the second one is only applied when the first one succeeded. The result of the first transformation is passed into the second one. This is implemented as the andThen method – or alternatively with the &> operator – on Transformation, which takes the second transformation as a by-name parameter:

```
abstract class Transformation[A, B] extends (A ⇒ Option[B]) {
  self ⇒

  def apply(in: A): Option[B]

  def andThen[C](t: ⇒ Transformation[B, C]) = new Transformation[A, C] {
    def apply(a: A): Option[C] = {
      self(a) flatMap t
    }
  }
  def &>[C](t: ⇒ Transformation[B, C]) = andThen(t)
...
```

Alternative $|>: \ (A \xrightarrow{t} [B]) \Rightarrow (A \xrightarrow{t} [B]) \Rightarrow (A \xrightarrow{t} [B])$

Combines two transformations so that the second one is only applied in case the first one fails. The implementation is directly based on the underlying Option type in the orElse method on Transformation and also has an operator alias:

```
abstract class Transformation[A, B] extends (A ⇒ Option[B]) {
  self ⇒

  def apply(in: A): Option[B]

  def orElse(t: ⇒ Transformation[A, B]) = new Transformation[A, B] {
    def apply(a: A): Option[B] = {
      self(a) orElse t(a)
    }
  }
  def |>(t: ⇒ Transformation[A, B]) = orElse(t)
...
```

With these two combinators, we are already able to represent conditional transformations. For example, given a transformation isClass that acts as a predicate, and two transformations a and b that represent the two possible branches the transformation can take, we can combine them into a new transformation isClass &> a |> b that executes the a transformation if the isClass transformation succeeds or b if either isClass or a fails.

Note that due to Scala's precedence rules, the |> combinator has a lower precedence than &>.

Predicate predicate: $(A \xrightarrow{?} Boolean) \Rightarrow (A \xrightarrow{t} [A])$

As we have seen, transformations can be used as predicates. We often want to construct a predicate from a function that returns a boolean value. This can be done with the predicate function which create a transformation from a partial function.

```
def predicate[A](f: ⇒ PartialFunction[A, Boolean]) = new Transformation[A, A] {
  def apply(a: A): Option[A] = if (f.isDefinedAt(a) && f(a)) Some(a) else None
}
```

Not !: $(A \xrightarrow{t} [A]) \Rightarrow (A \xrightarrow{t} [A])$

A combinator that inverts a transformation. Given a transformation that succeeds, then not will fail. Should the given transformation fail, then not returns the original input unchanged. This behavior is useful for transformations that act as predicates; not can be implemented using the fail and id transformations as follows.

```
def not[A](t: ⇒ Transformation[A, A]) = t &> fail |> succeed
```

Now that we have several means to specify and combine our transformations, we also need a way to apply them to a whole AST, instead of just single tree nodes. For this, there exist several traversal strategies.

2.3.3. Traversal

Applying a transformation to a single tree element is not difficult, but once we want to traverse the whole AST, we need a way to apply a transformation to all children of a tree node and to construct a new tree from the result of the transformation operation. Note that traversal strategies are also just transformations that can again be combined.

All Children $allChildren : (A \xrightarrow{t} [B]) \Rightarrow (A \xrightarrow{t} [B])$

Takes a transformation and creates a new one that applies the given transformation to all children, returning a single tree. Because there is no generic way to get all

children and construct a new tree, we constrain the type parameter A to be convertible to $(A \Rightarrow B) \Rightarrow B$. This means that the user of the generic transformation has to pass us its children and create a new tree. When a child cannot be transformed, allChildren immediately aborts and returns None.

```
def allChildren[A <% (A ⇒ B) ⇒ B, B](t: ⇒ Transformation[A, B]) =
  new Transformation[A, B] {
    def apply(a: A): Option[B] = {
      Some(a(child ⇒ t(child) getOrElse (return None)))
    }
  }
```

X <% Y is called a *view bound* and demands that there exists an implicit conversion from type X to Y (see Appendix E.3 on page 118). This is less constrictive than X <: Y, where X has to be a subtype of Y. In our case, we can then treat a as if it were of type (A ⇒ B) ⇒ B. This allows us to apply the transformation to the children of a.

Matching Children $\quad matchingChildren : (A \xrightarrow{t} [A]) \Rightarrow (A \xrightarrow{t} [A])$

The allChildren traversal only succeeds when the transformation can be applied to all children. If children that cannot be transformed should simply be kept and passed to the new tree unchanged, we can use the matchingChildren transformation.

```
def matchingChildren[A <% (A ⇒ A) ⇒ A](t: Transformation[A, A]) = allChildren(t |> id[A])
```

Using the id transformation, we retain the original tree should the transformation not be applicable. A consequence of this is that the transformation needs to be done between the same types.

The next step after being able to apply a transformation to a tree or all of its children is to expand this to the AST as a whole. We can distinguish between two fundamental ways of transforming a tree: either in a pre-order or post-order fashion.

Pre-Order $\quad \downarrow : (A \xrightarrow{t} [A]) \Rightarrow (A \xrightarrow{t} [A])$

Pre-order application of a transformation applies the transformation to the parent first and then descends into its children. The consequence is that at the time a tree gets transformed, its children are still in their original, untransformed state.

```
def ↓        [A <% (A ⇒ A) ⇒ A](t: Transformation[A, A]) = t &> allChildren(↓(t))
def preorder [A <% (A ⇒ A) ⇒ A](t: Transformation[A, A]) = ↓(t)
def topdown[A <% (A ⇒ A) ⇒ A](t: Transformation[A, A]) = ↓(t)
```

Using a pre-order transformation has the benefit that trees are always in their original state when they are transformed, this can be used when the trees need to be

compared for equality. A disadvantage is that a transformation can diverge when it modifies a tree so that it again applies to one of its new children. For example, applying the following transformation to a tree results in a stack overflow when applied with pre-order traversal:

```
transformation[Tree, Tree] {
  case block @ Block(stats, _) => block copy (stats = block :: stats)
}
```

This will not happen when the transformation is applied using post-order traversal.

Post-Order $\uparrow: (A \xrightarrow{t} [A]) \Rightarrow (A \xrightarrow{t} [A])$

Bottom-up application first descends into the children of a tree and processes the parent after the children. Thus once a tree gets transformed, its children have already been transformed.

```
  def ↑         [A <% (A ⇒ A) ⇒ A](t: Transformation[A, A]) = allChildren(↑(t)) &> t
  def postorder [A <% (A ⇒ A) ⇒ A](t: Transformation[A, A]) = ↑(t)
  def bottomup  [A <% (A ⇒ A) ⇒ A](t: Transformation[A, A]) = ↑(t)
```

Combining all these transformations with combinators and traversal strategies allows us to describe transformations in a very concise way. Figure 2.5 on the following page illustrates the difference between the two traversal modes.

Examples

As a first example, let us write and apply a transformation that replaces all trees in the AST which do not have a range position with the EmptyTree.

```
val tree: Tree = ...

val emptyTree = transformation[Tree, Tree] {
  case t if t.pos.isRange => t
  case _ => EmptyTree
}

preorder(allChildren(emptyTree)) apply tree
```

Pre-order traversal already applies the transformation to all children, so we can simplify this to:

```
preorder(emptyTree) apply tree
```

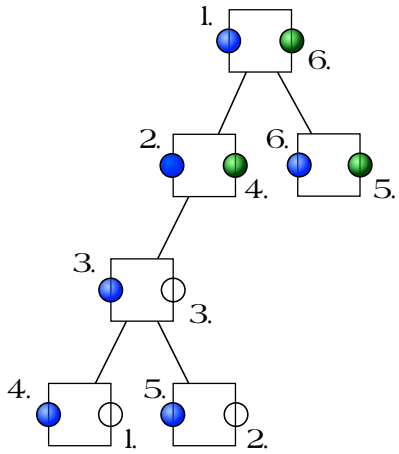


Figure 2.5.: An illustration of the pre- and post-order traversal strategies: the blue points show the order in which the tree gets transformed in pre-order traversal, and the green ones illustrate the post-order traversal. Instead of pre- and post-order, we can also think of the transformations being applied top-down or bottom-up, hence the ↓ and ↑ aliases.

Of course, this is not the only way to achieve this, here is a variation that separates the testing for the range position into a predicate and uses a simpler transformation to replace the tree. If the tree has a range position, it is not transformed (remember that the id transformation simply returns its argument unchanged). In case the predicate fails, the tree is replaced.

```
val hasRangePosition = predicate((t: Tree) => t.pos.isRange)

val emptyTree = transformation[Tree, Tree] {
  case _ => EmptyTree
}

preorder(hasRangePosition &> id[Tree] |> emptyTree) apply tree
```

Using the not combinator, we can swap the two actions:

```
preorder(not(hasRangePosition) &> emptyTree |> id[Tree]) apply tree
```

To get rid of the id transformation, we can use a different traversal strategy for the children:

```
preorder(matchingChildren(not(hasRangePosition) &> emptyTree)) apply tree
```

More examples can be found in Section 2.3.5 on the next page.

2.3.4. Creating Trees

Most refactorings do not just reuse existing trees but also have to create new ones. The Scala compiler already contains several facilities to create new trees: the trait scala.tools.nsc.ast.Trees contains many methods that create AST trees and there's even a DSL in scala.tools.nsc.ast.TreeDSL whose "goal is that the code generating code should look a lot like the code it generates" [Tre10].

An example from Trees shows how many methods there are to create method definitions (this code has obviously been written before Scala had default arguments):

```
def DefDef(sym: Symbol, mods: Modifiers, vparamss: List[List[ValDef]], rhs: Tree): DefDef

def DefDef(sym: Symbol, vparamss: List[List[ValDef]], rhs: Tree): DefDef

def DefDef(sym: Symbol, mods: Modifiers, rhs: Tree): DefDef

def DefDef(sym: Symbol, rhs: Tree): DefDef

def DefDef(sym: Symbol, rhs: List[List[Symbol]] => Tree): DefDef
```

Using the TreeDSL allows one to write very concise code. The following listing creates the AST for the code that checks whether tree is null.

```
IF (tree MEMBER_== NULL) THEN ... ELSE ...
```

Unfortunately, all these tree construction helpers are problematic for us: they can change the position of the trees, which we have to avoid when we want to retain the source code layout. For this reason, the refactorings do not make use of these facilities but simply create the trees from scratch. There are some helper methods in transformation.TreeFactory which take care of constructing trees that are needed by the currently implemented refactorings:

```
def mkRenamedSymTree(t: SymTree, name: String): SymTree

def mkValDef(name: String, rhs: Tree): ValDef

def mkCallDefDef(name: String, arguments: List[List[Symbol]],
  returns: List[Symbol]): Tree

def mkDefDef(mods: Modifiers, name: String,
  parameters: List[List[Symbol]], body: List[Tree]): DefDef

def mkBlock(trees: List[Tree]): Block
```

Now that we have seen how trees can be transformed and how new trees can be generated, we are ready for a larger example.

2.3.5. Tree Transformations

For the usage in the refactoring, the TreeTransformations trait implements the traversal for Scala's AST and provides some definitions that make writing transformations more concise:

```
def transform(f: PartialFunction[Tree, Tree]) = transformation(f)

def filter(f: ⇒ PartialFunction[Tree, Boolean]) = predicate(f)
```

Let us now take a look at a larger example: Extract Method. At the beginning of this section, we looked at the different transformations that occur during the refactoring: Insert a new method with the extracted statements and replace them with a call to this new method. This can be achieved with the following transformations:

```
val replaceBlockOfStatements = transform {
  case block @ BlockExtractor(stats) => {
    mkBlock(stats.replaceSequence(selectedTrees, callExtractedMethod))
  }
}

val replaceSingleExpression = transform {
    case t if t == selectedTree => callExtractedMethod
}

val replace = topdown {
  matchingChildren {
    if(extractSingleTree)
      replaceSingleExpression
    else
      replaceBlockOfStatements
  }
}

val insertExtractedMethod = transform {
  case tpl @ Template(_, _, body) =>
    tpl copy (body = body ::: extractedMethod :: Nil) setPos tpl.pos
}
```

A remark on the call to setPos tpl.pos in insertExtractedMethod: Because the structure of a tree is immutable, we cannot change a tree in-place, even though we often want to do this. The source regeneration uses the position information of the trees to determine

whether a tree's existing source code can be reused. So if we want a tree to appear modified in-place, we simply assign it the position of the original tree. Note that this does only work if the two trees are of the same type.

Next we need two filters that find the enclosing class' template and the method we extract from:

```
val findTemplate = filter {
  case Template(_, _, body) => body exists (_ == selectedMethod)
}

val findMethod = filter {
  case d: DefDef => d == selectedMethod
}
```

Now we can combine these to assemble a new transformation that performs the following steps:

1. Traverse the tree until the selected template is found, the one that contains selectedMethod.
2. Once we found the template, start the following two transformations:
   a) Find the method we extract from and apply the replace transformation on it.
   b) Insert the new method in the class template.

All these steps can be expressed with the following transformation:

```
val extractMethod = topdown {
  matchingChildren {
    findTemplate &>
    topdown {
      matchingChildren {
        findMethod &> replace
      }
    } &>
    insertExtractedMethod
  }
}
```

More concrete implementations of transformations can be found in Chapter 3 on page 39.

2.4. Source Generation

Once our abstract syntax tree has been transformed, we need to convert it back into its textual source code representation. This process comprises two main steps: the *detection of modifications* to minimize the amount of code that is regenerated and the actual *source generation*.

The first step is necessary because we – in contrast to many other refactoring implementations – do not keep track of modifications to the AST while they are happening but reconstruct this information afterwards. This allows us to keep the transformations simpler but consequently makes the code generation more complex. This trade off is worthwhile because we intend the library to be reused and the transformations to be implemented by developers who do not (need to) know the details of the source generation.

The AST after the refactoring may contain several kinds of modifications: trees can be moved around, deleted and new trees can be introduced. From the transformations we know that trees that are moved around keep their original position information, and newly created trees have a NoPosition attribute per default. This allows us to detect changes and can later be used during source generation to preserve the layout of already existing trees.

2.4.1. Modification Detection

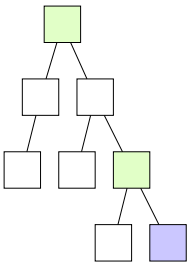
The primary goal of a fine-grained modification detection is to reduce the amount of trees that are regenerated. The source generation is invoked with a list of trees from various files that all can have an arbitrary number of changed children:

```
def createChanges(ts: List[Tree]): List[Change]
```

Modification detection performs the following three steps on the input trees:

1. Group all changed trees by their file.
2. Find the top-level changed trees for each file.
3. Detect the changes per top-level tree.

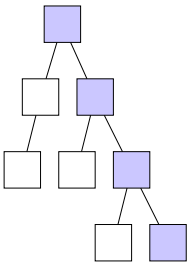
Top-level trees are trees that are ancestors of other changed trees. For example, the following graph shows an AST with some changed trees in green and blue:



The `createChanges` method is invoked with the two green trees, but the blue tree has also been modified by a (sub-) transformation.

Now if we were to generate two changes from the two green trees, we would get a problem when applying the changes because they overlap each other. The two changes would either overwrite each other or, in the case of Eclipse's Language Toolkit, yield an error. Therefore the second step of the modification detection is to find those trees that contain other changed trees. In the AST above, this would be the root node.

The third step then traverses these top-level trees and finds all changes as well as the trees that lie between changed trees, here marked in blue:



This set of trees is the minimal number of trees that need to be regenerated. Trees that are not contained in the set can be kept as they are to improve the performance. Figure 2.6 on the next page shows a larger example of the process.

Once we have identified all top-level tree changes, we start generating source code for them.

2.4.2. Code Generation

The AST does – by its very nature – not contain all the information that is necessary to fully reconstruct its original textual representation. Also, syntactic sugar of the programming language is typically not represented in the AST (see Section D.3 on page 111 for some examples); only the desugared representation is preserved. An example for this are Scala's `for` comprehensions. Because they are equivalent with function calls to `map`, `filter`, `flatMap`, and `foreach`, there is no need to create additional

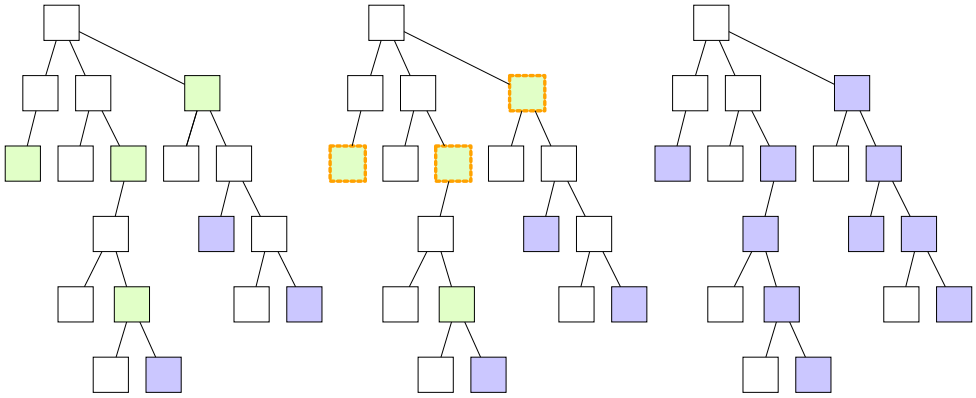


Figure 2.6.: An example of how the change set is built: the left AST shows in green the list of all trees that should be regenerated, but the blue trees have changed as well. In the middle graph, we see the trees that were identified as top-level trees. The rightmost AST shows all trees that need to be regenerated.

tree classes for them. This means that the two statements in the following listing have the same representation in the AST.

```
val v1 = List(1,2,3) map (i => i * 2)
val v2 = for(i <- List(1,2,3)) yield i * 2
```

Other things that are not mentioned explicitly in the AST are parenthesis, commas and many other tokens. In the context of source generation, we will call them layout elements, or just *layout*.

If we were only interested in a semantically equivalent program, we could simply pretty print the AST to generate the source code. A purely AST based pretty printer would unknowingly convert the user's for comprehensions from above into the map form. No user of a refactoring tool would accept this, and this is also not the only problem: because comments in the source code are generally considered whitespace by parsers, they are not represented in the AST and get lost during pretty printing (see [SZCF08] for a detailed treatment of this particular problem). It is clear that we need a more refined technique.

The original source code is always available to the refactoring tool, and with the position information on the trees, we have a means to look up the original source code for a tree.

Other refactoring tools (e.g. the C++ Development Tooling for Eclipse [GZS07] or the Ruby Development Tools [CFSS07]) have used various approaches to solve this

problem (see [Sto09] for details on these approaches). For some cases – for example, in a rename refactoring – it might even be acceptable to pretty print the code as long as only very small regions of the program change. But as a general solution, this approach is problematic. For example, with the Extract Method refactoring, where arbitrary large parts of the program are moved around. A tool can handle this situation by cutting-and-pasting the body of the extracted method. This is not feasible for us because we need a generic way to handle all kinds of unforeseeable changes to the source code.

Preserving Layout

Our approach is based on using two different kinds of source printers: one that pretty prints code and another one that reuses the existing code where possible. The *pretty printer* simply prints the code with a default layout and is used for trees that were introduced during the transformation. The *reusing printer* takes the existing layout with the help of the trees's position information and also makes sure all needed layout elements are present. How this is done will be explained in more detail later.

The source generation algorithm then alternates between these two printers during the code generation process.

Now we just need to know how we can reuse the existing layout. What we need is a way to decide how all these layout elements can be associated to their enclosing trees. If we take a look at the following listing, we can see several occurrences of whitespace and other layout, like the three comments and the braces.

```
package p //TODO
// myclass
class MyClass(a: Int /* the int */) {
}
```

Because no rules of the programming language dictate how the layout is associated with the other parts of the program, we have to guess how to divide it and associate it with its surrounding trees. Often this can be done by taking the types of the adjacent trees into consideration and then divide the layout according to some rules and regular expressions.

For example, one rule expresses that the layout between two enclosing value definitions is split by a comma, or by newline if there is no comma present. So when the values are part of an argument list, they will get comma-separated, and if they are definitions, the layout will be split at the end of the line, so that the first value will get all layout that follows on its line. Comments can be handled with the same rules as well: a comment on a preceding and otherwise empty line is associated with the following tree.

Let us take a look at a concrete example. Figure 2.7 on the next page shows the AST of the previous listing and how the layout elements have been associated with the left

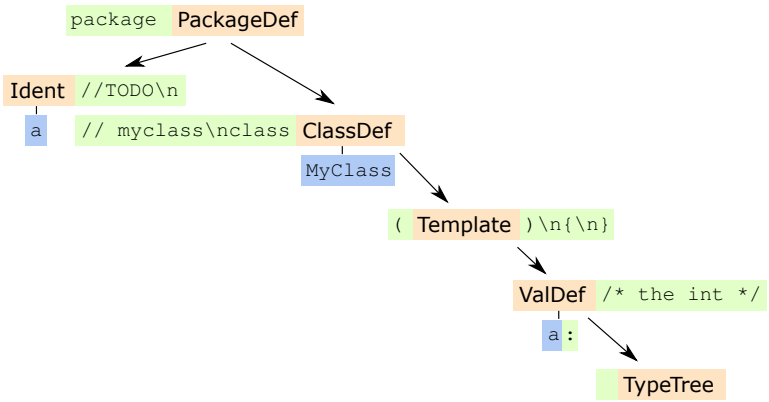


Figure 2.7.: An example of how layout can be associated with trees: the apricot colored boxes represent the trees and the green ones their associated layout. The blue parts are not real AST nodes but names; they are treated like trees in the source generation.

and right sides of a tree. Note that the class and package keywords are also considered layout, this is because they are not represented in the AST with their own tree and position information.

Once we have identified the layout that belongs to a tree, we can use it during the source generation. For example, it should be clear now that when we would delete the ValDef parameter in the above AST, then the comment would be removed along with it.

Another issue that concerns both the pretty and the reusing printer is the indentation of the code. When a new statement in a block of other statements is inserted, we want it to have the same indentation as its siblings. For this, the printers also keep track of the currently desired indentation as specified by the parent tree.

Whether we can reuse existing code or have to invoke the pretty printed needs to be decided for each tree in the AST. This gives us the following definition of the various source printers:

```
trait AbstractPrinter {
  def print(t: Tree, ind: Indentation): Fragment
}
```

```
trait PrettyPrinter extends AbstractPrinter {
  def print ...
}

trait ReusingPrinter extends AbstractPrinter {
  def print ...
}

trait SourceGenerator extends PrettyPrinter with ReusingPrinter {
  override def print(t: Tree, i: Indentation): Fragment = {
    if(t.hasExistingCode)
      super[ReusingPrinter].print(t, i)
    else if(t.hasNoCode)
      super[PrettyPrinter].print(t, i)
    else
      EmptyFragment
  }
  ...
}
```

Fragments and Layout

The result of a printing operation is not a plain string but an instance of Fragment. A fragment contains a leading, center, and trailing layout. A layout is simply a wrapper around a string or a part of the source file with some additional helper methods. For example, in Figure 2.7 on the preceding page, all the apricot and blue colored boxes are fragments and the green ones are instances of Layout.

The fragments and layouts are created in the printers. Printers pattern match on the current tree and recursively print the children of a tree. This is an excerpt from the pretty printer:

```
def print(t: Tree, ind: Indentation) = t match {
  case PackageDef(pid, stats) =>
    Layout("package ") ++
      printTree(pid, after = newline) ++
      printTrees(stats, separator = newline)
...
```

The ++ operation on the layout and fragments simply concatenate their operands, again yielding a fragment. So far, we could also have just used plain Strings and concatenate them with +, except that using strings is dangerous because every object can be concatenated with a String using the implicit toString method.

Reusing Layout

The printers also have to take care that all the necessary layout is printed when needed. This can become difficult when layout is reused. Imagine the following scenario: We create a new Block (a Block tree wraps a list of other statements) and insert several statements into it. The pretty printer separates each statement in a block with a newline, so the code to pretty print a block could look like this:

```
case Block(stats) =>
  Layout("{"+newline) ++
    printTree(stats, separator = newline) ++
    Layout(newline+"}")
```

This works fine as long as the statements are not reused trees that might already have a leading or trailing newline in their associated layout. If this is the case, we could get too many blank lines between our statements.

To solve this, the pretty printer could print the block's children one by one and then check if the newline is already present or needs to be inserted. This is tedious to do in every place where a layout element is inserted, so we need a more generic way to handle such cases, and this is where the Requisites come into play. Instead of specifying the layout directly, the printers simply declare that there needs to be a newline present in the surrounding layout:

```
case Block(stats) =>
  Requisite("{"+newline) ++
    printTree(stats, separator = Requisite(newline)) ++
    Requisite(newline+"}")
```

Now during the concatenation of fragments and layout objects with ++, it is checked whether a certain requisite is already satisfied. The Requisite's layout is only inserted when it is needed.

This leads us to the following three interfaces (the ++ operators and some other methods have been omitted) that are used to represent the source code in the printers:

```
trait Layout {
  def asText: String
}

trait Requisite {
  def isRequired(l: Layout, r: Layout): Boolean
  def apply(l: Layout, r: Layout): Layout
}
```

```
trait Fragment {
  def leading: Layout
  def center: Layout
  def trailing: Layout

  def pre: Requisite
  def post: Requisite

  def asText: String
}
```

Using implicit conversions (see Appendix E.3 on page 118), short aliases for the print methods and Scala's named and default arguments, this allows us to write the code for the two printers in a very concise way. Pattern matching gives us the ability to easily handle special cases and variations, as can be seen from the Bind matches below:

```
trait PrettyPrinter {

  def print ...

    case Alternative(trees) =>
      p(trees, separator = " | ")

    case Star(elem) =>
      p(elem) ++ Layout("*")

    case Bind(name, body: Typed) =>
      Layout(name.toString) ++ p(body, before = ": ")

    case Bind(name, body: Bind) =>
      Layout(name.toString) ++ p(body, before = " @ \\(", after = "\\)")

    case Bind(name, body) =>
      Layout(name.toString) ++ p(body, before = " @ ")
    ...
}
```

2.4.3. Using the Source Generator

For users of the code generation, there are several methods to transform a tree back into source code. The createChanges method of the SourceGenerator trait creates the change objects from a list of trees by first narrowing down the changed trees and then generating the code for them:

```
def createChanges(ts: List[Tree]): List[Change]
```

The result is a list of change objects that describe which parts in a file are to be replaced:

```
case class Change(file: AbstractFile, from: Int, to: Int, text: String)
```

This is the preferred method for IDEs that operate with change objects. The Change object contains a useful function that applies a list of changes to a source code string:

```
def applyChanges(ch: List[Change], source: String): String
```

Alternatively, if one just wants to generate the source code from a tree, the create-Fragment method can also be invoked directly, yielding a fragment for the lop-level tree.

```
def createFragment(t: Tree): Fragment
```

The createFragment method also minimizes the trees that are regenerated using the technique explained at the beginning of this section.

We have now completed our tour through the library's internals. The next section compares the current implementation of the code generation with the previous one from the term project, but can be safely skipped. The next chapter will then explain how the implemented refactorings use the library.

2.4.4. Comparison With the Term Project

Most parts of the source generation package have been re-written during the thesis because the previous version had some serious issues. To recapitulate, the earlier version built its own abstraction from the Scala AST to make the code generation easier to implement. This abstraction was built for the original AST and the modified AST; the idea was that source generation and detection of changes could then be implemented very generically. There were a few problems with this approach:

- Even though the generic approach worked well for simple cases, as soon as the code generation needed special handling for certain AST constructs (see the PrettyPrinter excerpt in Section 2.4.2 on the preceding page), they were much harder to implement because we were working with our own abstraction. So we had to include more and more information into this abstraction, making the once simple constructs very complex.

- Using the pimp-my-library pattern (see E.3 on page 118), new functionality can comfortably be added to existing code. In the current version, several implicit conversions (see the common.PimpedTrees trait) add useful features to the AST

classes. Using this approach, we still have the original AST classes but can adapt them to our needs.

- Building our own abstraction had a notable negative effect on the performance, up to the point where we had to start using memoization to speed up code generation. The current approach seems so far to be performant enough, without any performance optimizations.

The experience of implementing the refactorings with the current code generator has shown that it is much easier to adapt to new situations, and special cases can be handled concisely and at the point where they are needed.

3. Implemented Refactorings

The previous chapter explained the internals of the Scala Refactoring library; in this chapter, we shall take a look at the refactorings that have so far been implemented on top of it.

The three components of the refactoring library – analysis, transformation, and source generation – can be used independently from each other, but they also have dependencies expressed through self type annotations (see E.4 on page 119).

The Refactoring trait combines the library with their dependencies and can be used as an entry point by library users.

```
trait Refactoring extends
  Selections with
  TreeTransformations with
  SilentTracing with
  SourceGenerator with
  PimpedTrees {
  ...
}
```

Performing a refactoring is not a single-step process: when the user invokes a refactoring, the first step is to check whether the refactoring can be applied – for example, to perform a renaming, a name has to be selected. We call this the *prepare* step. This step usually has a result, which is used in a configuration dialog to parameterize the refactoring. In our renaming example, this is the new name. Using the information from the preparation step and the configured parameters, the refactoring can then be *performed*. This yields either a list of changes to be applied or it can also fail. See Figure 4.1 on page 62 for a visualization.

These steps are represented by the abstract class MultiStageRefactoring, which is subclassed by all concrete refactoring implementations:

```
abstract class MultiStageRefactoring extends Refactoring {

  type PreparationResult

  case class PreparationError(cause: String)

  def prepare(s: Selection): Either[PreparationError, PreparationResult]

  type RefactoringParameters

  case class RefactoringError(cause: String)

  def perform(selection: Selection, prepared: PreparationResult, params: RefactoringParameters)
    : Either[RefactoringError, List[Change]]
}
```

The reason why the selection and the preparation results need to be passed to perform is to keep it stateless. This makes it much easier for an IDE to let the user go backwards and forwards in its wizard, testing different configurations.

The remainder of this chapter introduces each refactoring and explains the current implementation for the Eclipse Scala IDE with examples.

3.1. Rename

Renaming is one of the most used refactorings among Eclipse using Java programmers (see [MHPB09], [MKF06]). Choosing good names is a very basic and yet important task for a programmer if he wants to write readable code. During the evolution of a program, the roles of the classes, methods and variables change. Having an automated refactoring for renaming considerably reduces the cost of keeping these names in sync with their functionality.

3.1.1. Features

This implementation supports renaming of all identifiers that occur in the program – for example, local values and variables, method definitions and parameters, class fields, variable bindings in pattern matches, classes, objects, traits, packages, and types parameters.

The IDE implementation distinguishes between two different modes: inline renaming as shown in Figure 3.1 on the following page and the traditional dialog based implementation in Figure 3.2 on the next page. Inline renaming is implemented using Eclipse's linked mode user interface [Lin10].

Inline renaming is automatically chosen if the identifier that is renamed has only a local scope – for example, a local variable. All names that can potentially be accessed

```
object RenameDemo {

  def helloWorld {

    val msg = List("Hello", "World")

    println(msg mkString " ")
  }
}
```

Figure 3.1.: The Rename refactoring in the inline mode: the selected name along with all references can be renamed without the need of a wizard and without previewing the changes.

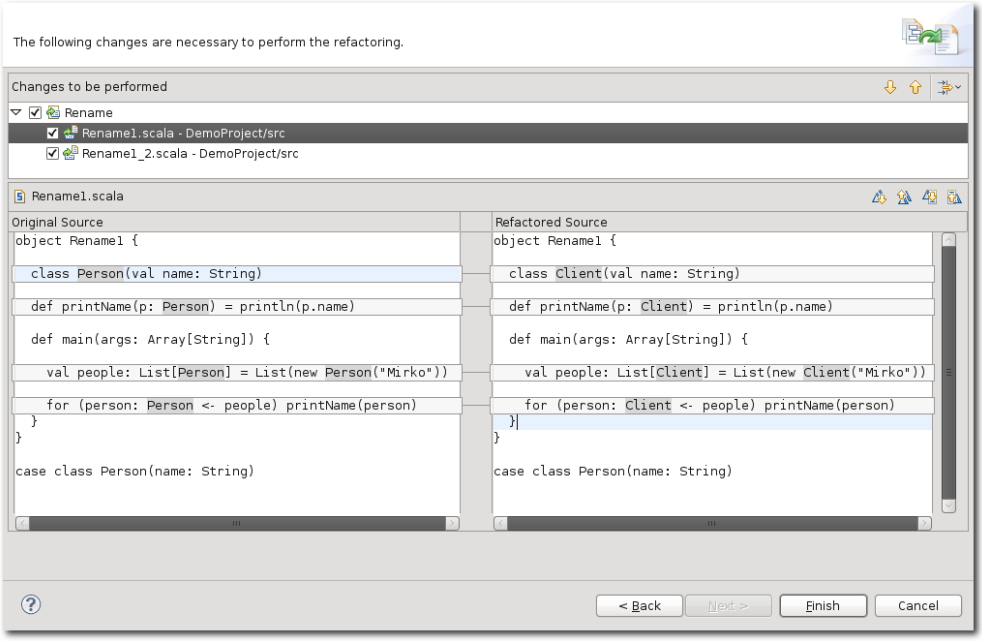


Figure 3.2.: A classical Rename refactoring: All occurrences of the selected name are changed across all files in the project.

from other compilation units in the program are renamed with the wizard and show a preview of the changes.

3.1.2. Implementation Details

From the refactoring developer's point of view, the Rename refactoring is quite different from other refactorings. Because renaming does not change the shape of the AST at all, the transformations and source generation steps are trivial – or not even needed. On the other hand, having an accurate index is crucial. The inline rename refactoring uses the index to find the locations of the names and uses neither the source generator nor tree transformations.

The implementation of the non-inline mode looks as follows:

```
val occurences = index.occurences(selectedTree.symbol)

val isInTheIndex = filter {
  case t: Tree ⇒ occurences contains t
}

val renameTree = transform {
  case t: ImportSelectorTree ⇒
    mkRenamedImportTree(t, newName)
  case s: SymTree ⇒
    mkRenamedSymTree(s, newName)
  case t: TypeTree ⇒
    mkRenamedTypeTree(t, newName, selectedTree.symbol)
}

val rename = topdown(isInTheIndex &> renameTree |> id)

val renamedTrees = occurences flatMap (rename(_))
```

The renameTree transformation handles different kinds of trees but delegates to the TreeFactory to create the renamed trees. The rename transformation traverses the trees and renames the trees that are in the index, or keeps the original trees otherwise. This transformation is then applied to all trees returned by the index.

Why do we have to traverse the trees, would it not suffice to call occurrences flatMap (renameTree(_)) directly? No, this will not work for recursive method calls, where the method definition also has a child tree that has to be renamed.

3.1.3. Limitations

There is currently one limitation with the Rename refactoring: named parameters will not be renamed because they are not represented in the AST.

3.2. Organize Imports

It can be debated whether Organize Imports really deserves the label Refactoring, because it does not change the structure of your code; but neither does the Rename refactoring. But Organize Imports is definitely useful, therefore we chose to include it in our refactorings.

During the lifetime of a compilation unit, external dependencies can change and new import statements are added and old ones are removed. Organize imports reorders and simplifies these statements.

3.2.1. Features

Organize Imports does not need a configuration; the current implementation performs these three steps:

Sort the statements alphabetically by their full name.

```
import java.lang.{String, Object}
import java.io.File
import collection.mutable.ListBuffer
```

```
import collection.mutable.ListBuffer
import java.io.File
import java.lang.{String, Object}
```

Collapse multiple distinct imports from the same package into a single statement:

```
import java.lang.String
import java.lang.Object
```

```
import java.lang.{Object, String}
```

Simplify the imports: when a wildcard imports the whole package content, individual imports from that package are removed, unless they contain renames:

```
import java.io._
import java.lang._
import java.io.FileSet
import java.lang.{String ⇒ S}
```

```
import java.io._
import java.lang.{String ⇒ S, _}
```

Figure 3.3 on the following page shows a screenshot of the refactoring in action.

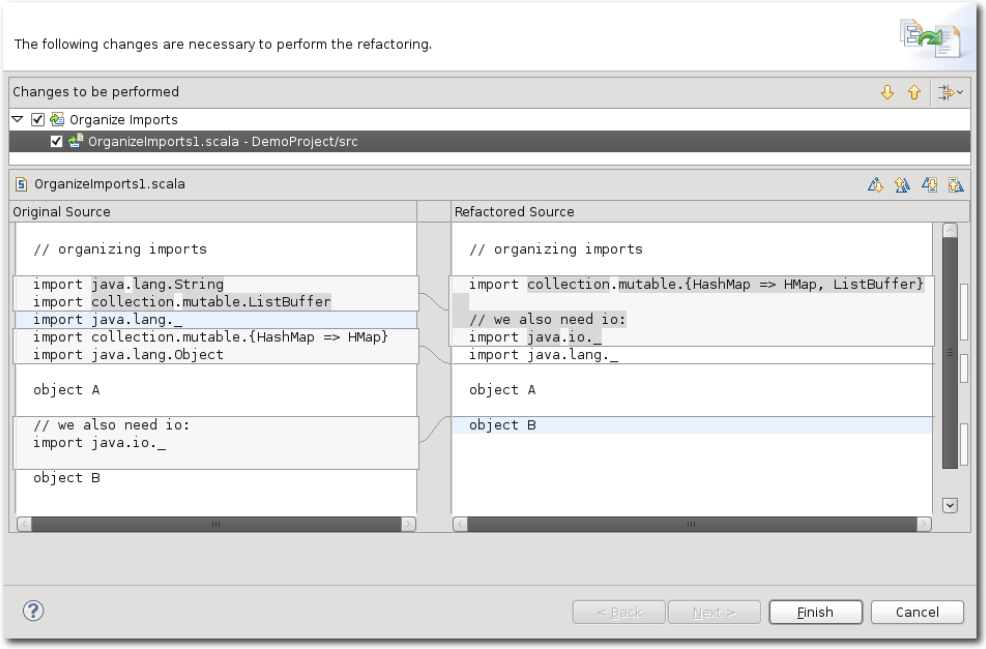


Figure 3.3.: The Organize Imports refactoring: we can see that the imports that were scattered all over the file are now all at the top in alphabetic order. All superfluous statements are getting removed, and imports from the same package are collapsed.

3.2.2. Limitations

The current implementation has several limitations compared to its Java counterpart. The refactoring does not do any dependency analysis, imports that are missing are not added, and unneeded imports are not being removed by Organize Imports. And there are more features that could be added in future versions:

Save Action In Eclipse, actions can be performed automatically when a file is saved. Enabling Organize Imports to automatically organize the imports might be useful.

Introduce Import In Scala, just as in Java, members from other packages do not have to be imported, they can also be used with their fully qualified name. Organize Imports could be extended to replace these fully qualified names with an import statement.

Expand Wildcards Once the refactoring does analyze the actually needed dependencies of the compilation unit, the refactoring might also replace all wildcard imports with just the necessary imports. This would also match the JDT's current behavior.

Shorten Import Paths In contrast to Java, packages in Scala can be nested (see Appendix E.5 on page 119). Organize Imports could take advantage of this and shorten the imported names. For example, the following import on the left could be simplified to the one on the right:

```
package scala.tools.refactoring
package common

import scala.tools.refactoring.analysis.Index
```

```
package scala.tools.refactoring
package common

import analysis.Index
```

3.3. Extract Local

Extract Local Variable, also known as *Introduce Explaining Variable*, should according to Fowler [Fow99] be used whenever "you have a complicated expression"; and the proposed fix is to

> put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.

In Scala, another reason why one would want to introduce new local variables is because existing Java debuggers are easier to use when one can step over single lines and examine the resulting values.

```
object ExtractLocal1 {

  def main(args: Array[String]) {

    args toList match {
      case x :: Nil =>
        val x = "one argument"
        println(x)
      case _ =>
        println("more than one argument")
    }
  }
}
```

Figure 3.4.: The Extract Local refactoring also uses the linked mode, making extracting a local variable much faster than with a wizard.

3.3.1. Features

From a selected expression, the Extract Local refactoring will create a new value in the enclosing scope and replace the selected expression with a reference to that value. Just as the rename refactoring in a local scope, Extract Local also uses Eclipse's linked mode to avoid distracting the user with dialogs (see Figure 3.4 for a screenshot).

The following listings show a few examples of the refactoring, on the left is the original code with the selection in gray, and on the right is the refactored code (line breaks were added by the author).

```
def main(args: Array[String]) {

  println("Detecting OS..")
  val props = System.getProperties



  if( props.get("os.name") == "Linux" ) {
    println("We're on Linux!")
  } else
    println("We're not on Linux!")
}
```

```
def main(args: Array[String]) {

  println("Detecting OS..")
  val props = System.getProperties
  val isLinux =
    props.get("os.name") == "Linux"

  if( isLinux ) {
    println("We're on Linux!")
  } else
    println("We're not on Linux!")
}
```

```
if(props.get("os.name") == "Linux") {

  println( "We're on Linux!" )
} else
  println("We're not on Linux!")
```

```
if(props.get("os.name") == "Linux") {
  val msg = "We're on Linux!"
  println( msg )
} else
  println("We're not on Linux!")
```

A more interesting examples shows what happens if there are no curly braces around the scope:

```
if(props.get("os.name") == "Linux") {
  println("We're on Linux!")
} else

  println( "We're not on Linux!" )
```

```
if(props.get("os.name") == "Linux") {
  println("We're on Linux!")
} else {
  val msg = "We're not on Linux!"
  println( msg )
}
```

We can extract all kinds of expressions – for example, a part of a chain of expressions:

```
val l = List(1,2,3)

l filter (_ % 2 == 0) mkString ", "
```

```
val l = List(1,2,3)
val filtered = l filter (_ % 2 == 0)
filtered mkString ", "
```

In the examples so far, we have only extracted expressions that resulted in a non-function value. Extract Local also lets you extract a method, which is turned into a partially applied function:

```
val l = List(1,2,3)
l filter (_ % 2 == 0) mkString ", "
```

```
val l = List(1,2,3)
val filterList = l filter _
filterList (_ % 2 == 0) mkString ", "
```

In the last example, we show how the extraction behaves inside single-expression functions:

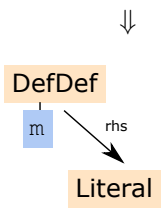
```
val l = List(1,2,3)
l filter (i ⇒ i % 2 == 0) mkString ", "
```

```
val l = List(1,2,3)
l filter (i ⇒ {
  val x = i % 2
  x == 0
}) mkString ", "
```

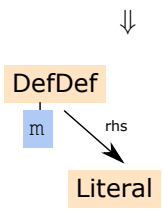
3.3.2. Implementation Details

On the first glance, extracting a local variable seems to be trivial, but when braces are missing, the source generation has to work hard to create them where necessary. An additional difficulty coming from Scala's AST is that Block trees around a scope are only created when there are multiple statements present. To illustrate this, the following three listings show their respective AST.

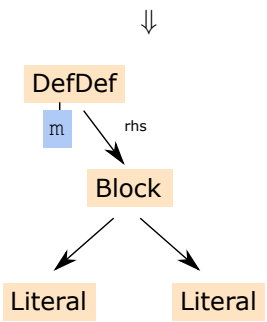
```
def m() = 42
```




```
def m() = {
  42
}
```




```
def m() = {
  42
  42
}
```



We can see that the AST in the middle looks just like the first one, even though the literal is surrounded with curly braces. Adding a second statement obviously forces the parser to surround them with a Block. When we extract a local variable, the refactoring generates a surrounding Block tree if needed, and the source generators have then to figure out whether they need to print new curly braces.

The Extract Local transformation is implemented as follows:

```
val findInsertionPoint = predicate((t: Tree) ⇒ t == insertionPoint)

def replaceTree(from: Tree, to: Tree) =
  topdown(matchingChildren(predicate((t: Tree) ⇒ t == from) &> constant(to)))

val insertNewVal = transform {

  case t @ CaseDef(_, _, NoBlock(body)) ⇒
    t copy (body = mkBlock(newVal :: body :: Nil)) replaces t

  case t @ Try(NoBlock(block), _, _) ⇒
    t copy (block = mkBlock(newVal :: block :: Nil)) replaces t

  case t @ DefDef(_, _, _, _, _, NoBlock(rhs)) ⇒
    t copy (rhs = mkBlock(newVal :: rhs :: Nil)) replaces t

  ...
}

val extractLocal =
  topdown(
    matchingChildren(
      findInsertionPoint &>
      replaceTree(selectedExpression, extractedValueReference) &>
      insertNewVal))
```

The findInsertionPoint transformation acts as a simple predicate to find the insertion point in the AST. Next, replaceTree creates a transformation that replaces two trees by top-down traversal. The insertNewVal transformation then takes care of inserting the value, creating the necessary surrounding Block trees. Finally, the transformations are combined and applied using a top-down traversal strategy.

3.3.3. Limitations

Curly braces are not always placed ideally – for example, the refactoring generates code like

```
(i ⇒ {
  ...
})
```

when it could just generate the code in the simpler form:

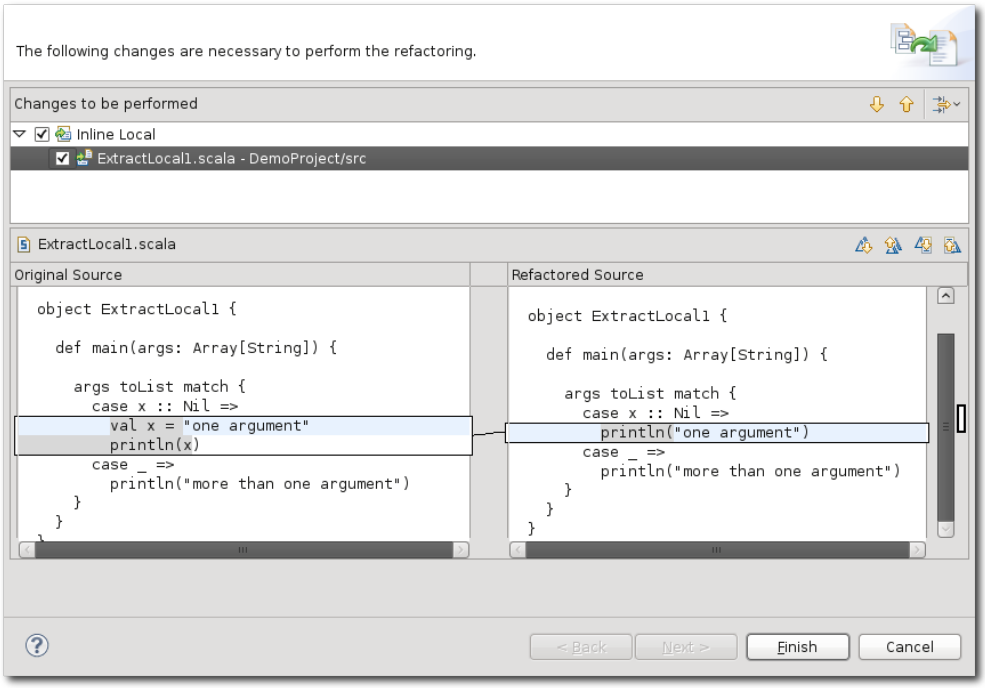


Figure 3.5.: The Inline Local refactoring lets us undo the extracted local refactoring from Figure 3.4 on page 46.

{i ⇒
  ...
}

3.4. Inline Local

The Inline Local – also known as Inline Temp – refactoring is the dual to Extract Local. It can be used to eliminate a local values by replacing all references to the local value by its right hand side.

Restricting the refactoring to vals only makes the refactoring easier to implement than its Java counterpart, where local variables can be reassigned. Still, inlining a local value can change the semantics of a program if the computation of the value has side-effects.

Figure 3.5 shows a screenshot of the refactoring in the Scala IDE for Eclipse.

3.4.1. Examples

Inlining a local value is in most cases trivial, but there are a few cases where it gets more complicated. Scala allows the programmer to omit the "." when calling methods in certain cases:

```
scala> Console println("Hello World")
Hello World

scala> List(1,2,3) filter (_ > 1)
res1: List[Int] = List(2, 3)

scala> 42 toString
res2: java.lang.String = 42
```

Things get more complicated when such calls are chained:

```
scala> List(1,2,3) filter (_ > 1) partition (_ % 2 == 0)
res3: (List[Int], List[Int]) = (List(2),List(3))

scala> 42 toString + " is the answer"
<console>:6: error: too many arguments for method toString: ()java.lang.String

scala> (42 toString) + " is the answer"
res5: java.lang.String = 42 is the answer
```

This means that when a value is inlined, it might become necessary to add parentheses around the inlined expression, as the following examples shows:

```
class Extr2 {
  def m {
    val five = 5 toString ;
    println(five)
    five + " is the answer"
  }
}
```

```
class Extr2 {
  def m {

    println(5 toString)
    (5 toString) + " is the answer"
  }
}
```

This is not done by the Inline Local implementation but by the source generator; another benefit that comes from separating the source generation from the refactorings.

3.4.2. Implementation Details

The Inline Local refactoring implementation is straight forward and is assembled from two transformations: one to remove the value and another one that replaces the

references to the value with its original right hand side.

The implementation for the first transformation looks as follows:

```
val removeSelectedValue = {

  def replaceSelectedValue(ts: List[Tree]) = {
    ts replaceSequence (List(selectedValue), Nil)
  }

  transform {
    case tpl @ Template(_, _, stats) if stats contains selectedValue =>
      tpl copy(body = replaceSelectedValue(stats)) replaces tpl
    case block @ BlockExtractor(stats) if stats contains selectedValue =>
      mkBlock(replaceSelectedValue(stats)) replaces block
  }
}
```

The selected value can either be contained in a block, or directly in a class template's body. To replace the reference, we first have to find out with what we want to replace it. If the value is bound to a method as follows:

```
val inlineThis = someList filter _
```

We need the _ to be removed, this is why we cannot just take the value's right hand side. The replace transformation then simply replaces all trees that reference the value:

```
val replaceReferenceWithRhs = {

  val references = index references selectedValue.symbol

  val replacement = selectedValue.rhs match {
    // inlining 'list.filter _' should not include the '_'
    case Function(vparams, Apply(fun, args)) if vparams forall (_.symbol.isSynthetic) => fun
    case t => t
  }

  transform {
    case t if references contains t => replacement
  }
}
```

Combining these two transformations leads to the Inline Local refactoring:

```
val inlineLocal =
  topdown(
    matchingChildren(
      removeSelectedValue &>
      topdown(
        matchingChildren(
          replaceReferenceWithRhs))))
```

3.5. Extract Method

Also among the most used refactorings by Java programmers is Extract Method. Extract Method is another key refactoring in making code more readable: if there is some code that can be grouped together, turn it into a method, and give it a meaningful name. The refactoring takes care of passing and returning the necessary parameters.

Martin Fowler once called Extract Method the refactoring's Rubicon [Fow01]:

> if you can do Extract Method, it probably means you can go on more refactorings [because it] requires some serious work. You have to analyze the method, find any temporary variables, then figure out what to do with them.

3.5.1. Features

There exist several variations of the refactoring depending on how the selected code interacts with surrounding local variables. In the case where no local variables are used, the refactoring is trivial, we can just move the code into its own method and insert a call to it from its origin.

When local variables are used, they need to be passed into the extracted method; the problematic case arises when local variables are re-assigned or declared inside and used outside of the extracted code. In this case, the respective variable has to be returned from the created method and the call to the created method becomes an assignment to the variable.

In Java, this scheme works as long as no more than one variable requires such special treatment. In Scala, we are also restricted by a single return value, but in contrast to Java, Scala has tuples and syntactic sugar for tuple creation and deconstruction, as shown in Figure 3.6 on the following page. This allows us to perform the refactoring in Scala where it would not (easily) be possible with similar code in Java.

Scala has other features like first class functions that allow variations of the refactoring, as described in [Sto09]. One more thing to mention is the choice of method placement: Scala allows methods to be defined inside other methods, which could also be an option for an Extract Method refactoring implementation.

```
def parse(source: String): (Int, String) = {
  ...
  (intResult, restSource)
}

val (parsedInt, restSource) = parse("5$")
```

Figure 3.6.: An example of Scala tuples. The function parseInt has the type String ⇒ (Int, String), which is syntactic sugar for String ⇒ Tuple2[Int, String]. Line 3 shows how such a tuple can be returned; and the last line how it is deconstructed into the two variables parsedInt and restSource.

3.5.2. Implementation Details

Extracting a method is done in several smaller steps:

Create the Method we want to extract. This includes determining all the inbound and outbound dependencies to construct the method signature.

Replace Extracted Statements with a call to the newly created method. In case the method returns values, we have to assign them to new local values.

Insert the Method somewhere in the surrounding class body.

The transformations that are used for Extract Method have already been described in Section 2.3.5 on page 26. A screenshot of the refactoring can be seen in Figure 3.7 on the following page.

3.5.3. Examples

To start, let us extract a statement that uses local variables and defines a variable that is used later in the program:

```
def main(args: Array[String]) {
  println("Hello World!")
  //create a message:
   val msg = args mkString ", "
  println(msg)
}
```

Applying the Extract Method refactoring results in a new method that takes an array as parameter and returns the created string.

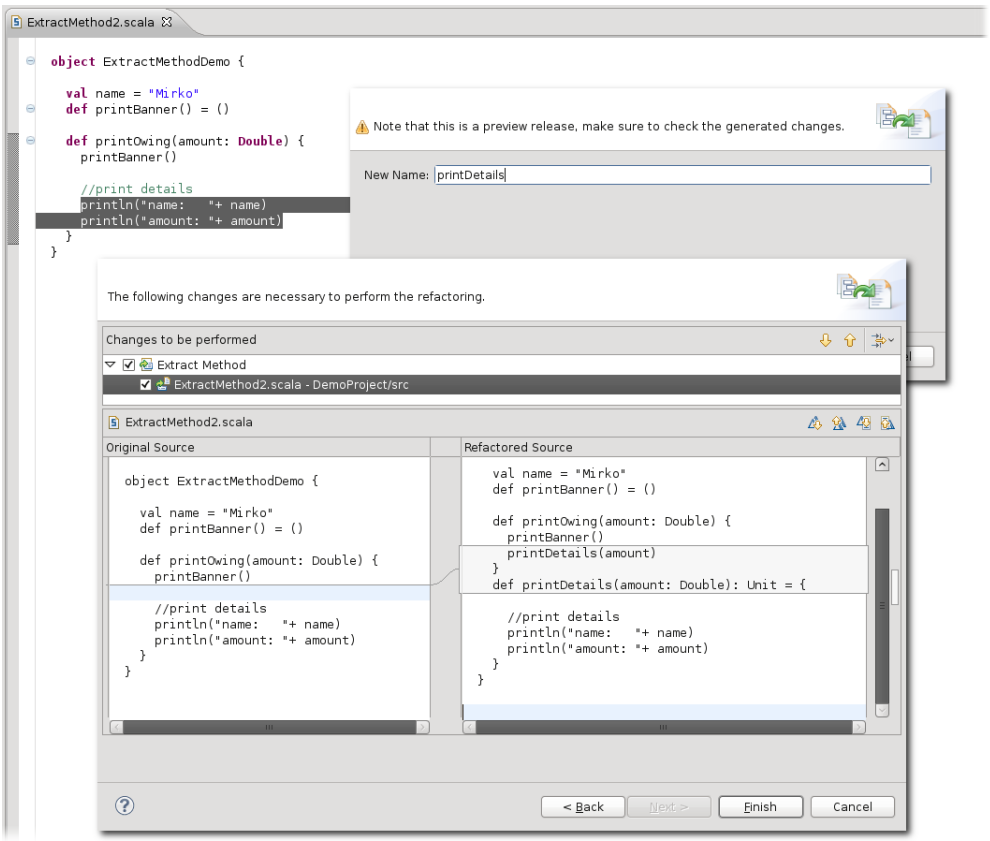


Figure 3.7.: The Extract Method refactoring: starting with a text selection, the user has to provide a name for the extracted method. The proposed changes are displayed to the user and can then be applied.

```
def main(args: Array[String]) {
  println("Hello World!")
  val msg = makeString(args)
  println(msg)
}
private def makeString(args: Array[String]): String = {
  //create a message:
  val msg = args mkString ", "
  msg
}
```

Returning Multiple Values

Using tuples to return values, we can return multiple values from an extracted method. The next two listings show an example:

```
def main(args: Array[String]) {
  val start = 0
  val end = 10
  val sum = start to end reduceLeft ((x, y) => x + y)
  println("The sum from %d to %d is %d".format(start, end, sum))
}
```

becomes

```
def main(args: Array[String]) {
  val start = 0
  val (end, sum) = calculateSum(start)
  println("The sum from %d to %d is %d".format(start, end, sum))
}
private def calculateSum(start: Int): (Int, Int) = {
  val end = 10
  val sum = start to end reduceLeft ((x, y) => x + y)
  (end, sum)
}
```

Higher Order Functions

Extract Method can also create a higher order function, as shown below:

```
def main() {

  val sumList: Seq[Int] => Int = _ reduceLeft (_+_)
  val prodList: Seq[Int] => Int = _ reduceLeft (_*_)

  val values = 1 to 10 toList

  val sum = sumList(values) // the sum
  val product = prodList(values) // the product

  println("The sum from 1 to 10 is "+ sum +"; the product is "+ product)
}
```

We can now extract the calculation of the sum and the product values. Both values are returned because they are used later in the print statement (line breaks were added manually):

```
def main() {

  val sumList: Seq[Int] => Int = _ reduceLeft (_+_)
  val prodList: Seq[Int] => Int = _ reduceLeft (_*_)

  val values = 1 to 10 toList
  val (sum, product) = sumAndProd(sumList, prodList, values)

  println("The sum from 1 to 10 is "+ sum +"; the product is "+ product)
}
private def sumAndProd(sumList: (Seq[Int]) => Int,
                       prodList: (Seq[Int]) => Int,
                       values: List[Int]): (Int, Int) = {
  val sum = sumList(values) // the sum
  val product = prodList(values) // the product
  (sum, product)
}
```

Arbitrary Expressions

In our examples so far, we have only extracted statements from blocks. But we can also extract single expressions from within other expressions. The following example extracts the condition of an if-expression:

```
def main(args: Array[String]) {

  println("Detecting OS..")

  if( System.getProperties.get("os.name") == "Linux" ) {
    println("We're on Linux!")
  }
}
```

```
def main(args: Array[String]) {

  println("Detecting OS..")

  if(isLinux) {
    println("We're on Linux!")
  }
}
private def isLinux: Boolean = {
  System.getProperties.get("os.name") == "Linux"
}
```

3.5.4. Limitations

Compared to Eclipse's Extract Method for Java (see Figure 3.8 on the next page), our version offers far less features – for example, one cannot reorder the parameters, nor rename them. Allowing the user to choose where the extracted method should be placed also has not been implemented yet, and the visibility of the extracted method is always set to private.

Also, the generated code is not always as simple as it could be. Consider the following example:

```
private def makeString(args: Array[String]): String = {
  //create a message:
  val msg = args mkString ", "
  msg
}
```

The local value msg could be inlined to get this simplified extracted method:

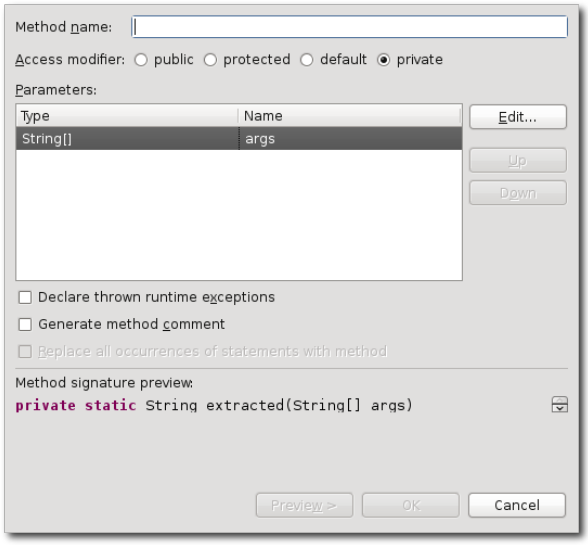


Figure 3.8.: Eclipse JDT's Extract Method Refactoring can be highly customized.

```
private def makeString(args: Array[String]): String = {
  //create a message:
  args mkString ", "

}
```

DefTree SymTree <: Tree

The DefTree class is extended by all trees that define or introduce a new entity into the program. Each DefTree also has a name and introduces a symbol.

RefTree SymTree <: Tree

RefTrees represent references to DefTrees. They also have a name and their symbol is the same as their corresponding DefTree's – i.e. it can be compared using ==.

Symbol Symbols provide another view on the program. Symbols are introduced by DefTrees and referenced by RefTrees. Symbols provide a lot more information about the program than the Trees – there are several dozen isXy methods defined on symbol to query information. In contrast to Trees, Symbols are much more connected to each other: this allows us to resolve class hierarchies or find the enclosing method or class for a symbol. Martin Odersky recorded three videos [Ode09a] that walk through the compiler, explaining many Symbol-related concepts.

Position The pos attribute of the trees is very important for us during refactoring. Unless the compiler runs in IDE mode, it generates OffsetPositions, which have only a single offset – the point – that indicates where a tree comes from; this is sufficient for non-interactive usage. When running in interactive – IDE – mode, the compiler generates RangePositions to represent the source range a tree originally comes from and OffsetPositions for those trees that were compiler-generated. Ranges have a start-, a point- and an end offset, where the end offset points to the position after the last character.

Range positions satisfy the following invariants, as specified in the Position.scala source's documentation:

1. A tree with an offset position never contains a child with a range position.
2. If the child of a tree with a range position also has a range position, then the child's range is contained in the parent's range.
3. Opaque range positions of children of the same node are non-overlapping (this means their overlap is at most a single point).

Due to the first invariant, compiler generated trees with an OffsetPosition cannot have children with original source code. Sometimes, it is still necessary for the compiler to generate trees that have children with RangePositions; in these cases, a Transparent position instead of an OffsetPosition is used.

There's also a NoPosition object that is assign to trees that have no origin in a source file. In the refactorings, we use this to indicate newly generated trees.

D.2. Concrete Trees

EmptyTree TermTree <: Tree

A Tree object that can stand in for most other trees, it has no position, no type and no symbol. For ValDefs, the equivalent is the emptyValDef object.

PackageDef MemberDef <: DefTree <: SymTree <: Tree
pid: RefTree, *stats*: List[Tree]

Describes a package clause with a package identifier and a list of statements. The package identifier is either an instance of Ident for a package like package a or an instance of Select for a package name like package a.b. The compilation unit root is always a package, even if there is no explicit package declaration present. In this case, the identifier is simply <empty>.

According to the Scala Language Specification, the two different notations are equal:

```
package a
package b.c
```

```
package a {
  package b.c {
  }
}
```

If there exists a top level package definition, its position does not necessarily enclose the whole source file, everything that lies before the package keyword or after the last statement in the package is not contained in the position. In a package that contains no explicit package declaration and only one statement, the package definition has the same start and end position as the statement, but a different point, which makes them distinguishable.

ClassDef ImplDef <: MemberDef <: DefTree <: SymTree <: Tree
mods: Modifiers, *name*: Name, *tparams*: List[TypeDef], *impl*: Template

The definition for all kinds of classes and traits (objects are defined in ModuleDef). The definition contains all modifiers, the name and the type parameters. The class' constructor arguments, super classes and its body are all defined in the impl Template.

Modifiers are a set of abstract, final, sealed, private, protected, trait, case. Note that the class keyword is not contained in the modifiers. If the class is anonymous (this can be queried with isAnonymousClass on the class' symbol), the name is of the form $anon.

ModuleDef ImplDef <: MemberDef <: DefTree <: SymTree <: Tree
mods: Modifiers, *name*: Name, *impl*: Template

We can see that multiple statements in the body create an additional block that wraps the statements. The do ... while loops are represented in a similar way:

```
do println("loop") while (true)
```

```
doWhile$1(){
  println("loop")
  if (true)
    doWhile$1()
}
```

Thanks to pattern matching, extracting the relevant parts is easy:

```
case LabelDef(_, _, If(cond, body, _)) // while with single expression
case LabelDef(_, _, If(cond, Block((body: Block) :: Nil, _) // while
case LabelDef(_, _, Block(body, If(cond, _, _))) // do While
```

Import

SymTree <: Tree

expr: Tree, *selectors*: List[ImportSelector]

An import statement imports one or many names – the selectors – from a package or object expr. An ImportSelector has two name-position pairs, the first one stands for the imported name and the second one is an optional renaming. Wildcard imports are also represented with an ImportSelector.

Import trees can also be comma separated, in this case, only the first import includes the import keyword in its position.

Block

TermTree <: Tree

stats: List[Tree], *expr*: Tree

A Block encloses a list of statements in { ... } and returns the value of its expr child. Block trees are only generated when needed – for example, the right hand side of a DefDef with a single expression is not a Block but the expression itself, even when the expression is enclosed in { ... }.

The expr is usually the last line of a block, with regards to their positions, but this is not always the case. For example, when creating an anonymous class, the class is introduced with a compiler generated name and then instantiated:

```
val a = new {
}
```

```
val a: java.lang.Object = {
  final class $anon extends scala.AnyRef {
    ...
  }
  new $anon()
}
```

CaseDef Tree

pat: Tree, *guard*: Tree, *body*: Tree

The body of a Match tree contains a number of CaseDefs trees. The guard can be an empty tree if it is not present. Note that even though the if keyword is used, the tree is not an If tree.

Patterns can be of different form, the catch-all _ is simply an Ident tree, whereas extractors are represented through the UnApply trees. Patterns that use an @ binding or are restricted by type with : are Bind trees. The body can again be an arbitrary tree.

Alternative TermTree <: Tree

trees: List[Tree]

Alternative trees are used in case definitions to match on alternative clauses, they are separated by |.

Star TermTree <: Tree

elem: Tree

Patterns can choose to match the whole remainder of an extracted sequence using the _* pattern, as in the following example:

```
"abcde".toList match {
  case Seq(car, cadr, _*) => car
}
```

The wildcard-star is represented by the Star tree.

Bind DefTree <: SymTree <: Tree

name: Name, *body*: Tree

The Bind tree binds a name to an expression and is used in the patterns of CaseDef. We can see from some examples that several seemingly different syntax variations are all represented in a uniform way in the AST:

```
list match {
  case i => ...
  case i: Int => ...
  case a @ i: Int => ...
}
```

```
list match {
  case (i @ _) => ...
  case (i @ (_: Int)) => ...
  case (a @ (i @ (_: Int))) => ...
}
```

UnApply

TermTree <: Tree

fun: Tree, *args*: List[Tree]

When an extractor object is used in the pattern of a case definition, an UnApply tree is used. The arguments of UnApply can then be more patterns.

```
case Ex(i) => i

case a @ Ex(i) => i

case a @ Ex(i: Int) => i
```

```
case Ex.unapply(<unapply-selector>)
  <unapply> ((i @ _)) => i
case (a @ Ex.unapply(<unapply-selector>)
  <unapply> ((i @ _))) => i
case (a @ Ex.unapply(<unapply-selector>)
  <unapply> ((i @ (_: Int)))) => i
```

Function

TermTree <: Tree ∧ SymTree <: Tree

vparams: List[ValDef], body: Tree

The Function tree contains a single list of parameters and a body for the implementation. The following listing shows various usages of the Function tree and how their trees look like.

```
list foreach println

list foreach (println _)

list foreach (i => println(i))
list foreach ((i: Int) => println(i))
list foreach {
  case i => println(i)
}
```

```
list foreach ({
  ((x: Any) => println(x))
})
list foreach ({
  ((x: Any) => println(x))
})
list foreach (((i: Int) => println(i)))
list foreach (((i: Int) => println(i)))
list foreach (((x0$1: Int) => x0$1 match {
  case (i @ _) => println(i)
}))
```

In the first two examples, the functions are encapsulated in an additional Block

– hence the curly braces. When the function parameter does not have a name, the compiler generates one and marks it with the SYNTHETIC flag. In the last example, we see that the pattern matching on the parameter is made explicit in the AST.

Assign TermTree <: Tree

lhs: Tree, *rhs*: Tree

Assign trees are not used for all = calls, only for non-initial assignments to variables. Calls to setter methods are not represented by Assign trees but are regular method calls.

If TermTree <: Tree

cond: Tree, *thenp*: Tree, *elsep*: Tree

An If expression consists of three parts: the condition, the then part and the else part. If the else part is omitted, the literal () of type Unit is generated and the type of the conditional is set to an upper bound of Unit and the type of the then expression, usually Any.

else if terms are implemented using nested if conditionals. We can see this in the following listing.

```
if (a)
  b
else if (c)
  d
else
  e
```

```
if (a)
  b
else
  if (c)
    d
  else
    e
```

Note that the if used in pattern matching guards is not an If tree but a designated member of the CaseDef tree.

Match TermTree <: Tree

selector: Tree, *cases*: List[CaseDef]

A match tree is used to represent a pattern match, with the selector being the tree that is matched against. When a pattern matching expression is used as the body of a function, the selector is a synthetic value:

```
list foreach {
  case i => println(i)
}
```

```
list foreach (((x0$1: Int) => x0$1 match {
  case (i @ _) => println(i)
}))
```

Return TermTree <: Tree ∧ SymTree <: Tree

expr: Tree

The Return tree contains an expression that constitutes the return value. For return statements without an expression, the compiler generates a () literal.

Try TermTree <: Tree

block: Tree, *catches*: List[CaseDef], *finalizer*: Tree

The Try tree represents try ... catch expressions. Both the catches and the finalizer are optional.

Throw TermTree <: Tree

expr: Tree

The Throw tree stands for the throw keyword and its expression.

New TermTree <: Tree

tpt: Tree

The New tree represents new statements, the tpt member is the type that is being instantiated.

Typed TermTree <: Tree

expr: Tree, *tpt*: Tree

The Typed tree is used whenever an expression is annotated with a type. For example, in the following listing, the second and third occurrences of Int are Typed trees:

```
val a: Int = 42: Int
println(a: Int)
```

Typed trees are also used in pattern matching when the match checks the type of the underlying object.

TypeApply

GenericApply <: TermTree <: Tree

fun: Tree, *args*: List[Tree]

TypeApply trees are used whenever a type is applied to a generic method. For example, in the following listing, both expressions on the left are represented by the same AST on the right.

List(1,2,3)
 	
List[Int](1,2,3)
	Apply(
	  TypeApply(
	  Select(. . . , "apply"),
	  List(Int)),
	  List(1,2,3))

Apply

GenericApply <: TermTree <: Tree

fun: Tree, *args*: List[Tree]

Function application is represented with Apply trees. The fun is often a Select tree that specifies the function name and args are the actual parameters.

Super

TermTree <: Tree ∧ SymTree <: Tree

qual: Name, *mix*: Name

The Super tree represents a super call, with optional qualifier and super class specifier:

```
trait A {
  def x = 42
}
trait B extends A {
  override def x = 43
}
class C extends A with B {
  println(super[A].x)
}
```

This

TermTree <: Tree ∧ SymTree <: Tree

qual: Name

The This tree represents the this reference, with an optional qualifier:

```
class Outer {
  class Inner {
    val outer = Outer.this
  }
}
```

Select RefTree <: Symtree <: Tree

qualifier: Tree, *name*: Name

The Select tree occurs on places that select a name from a qualifier, e.g. in method calls. Note that the typer fully qualifies references as illustrated in the following listing.

```
class A {
  val a = ...
  val b = a
}
```

```
class A {
  val a = ...
  val b = A.this.a
}
```

As usual, these generated trees then have an OffsetPosition.

Ident RefTree <: Symtree <: Tree

name: Name

Holds a Name, which can be generated (check with symbol.isSynthetic) by the compiler. Note that the name is in its alphabetic form; the real name can be found via the tree's symbol.

Literal TermTree <: Tree

value: Constant

All literals are represented by Literal trees. All possible kinds of constants are listed in the Constant trait.

TypeTree AbsTypeTree <: TypTree <: Tree

original: Tree

From the Scala compiler's documentation [Abs10]:

> A synthetic term holding an arbitrary type. Not to be confused with with TypTree, the trait for trees that are only used for type trees. TypeTrees are inserted in several places, but most notably in RefCheck, where the arbitrary type trees are all replaced by TypeTrees.

The original type tree is still accessible via the TypeTree's original member. Note that the standard tree Traverser and Transformer visitors do not traverse into the original subtree.

SingletonTypeTree TypTree <: Tree

ref: Tree

Whenever the .type expression is used, the tree is represented by a SingletonTypeTree tree.

SelectFromTypeTree TypTree <: Tree ∧ RefTree <: SymTree <: Tree

qualifier: Tree, *name*: Name

Type selection of the form qualifier#name is represented with the SelectFromTypeTree tree.

CompoundTypeTree TypTree <: Tree

templ: Template

An intersection type is represented by a CompoundTypeTree. Note that the tree contains a Template, this is because the compound type can have an optional refinement:

```
trait A
trait B
... A with B {
  ...
}
```

AppliedTypeTree TypTree <: Tree

tpt: Tree, *args*: List[Tree]

When a type is applied to a polymorphic function, a TypeApply tree is used. When a type is applied to an other type, an AppliedTypeTree is used.

TypeBoundsTree TypTree <: Tree

lo: Tree, *hi*: Tree

Whenever a type is constrained to lower or upper bounds, TypeBoundsTree represents these bounds. If one of the bounds is omitted, the compiler inserts Nothing respectively Any for the missing lower or upper bound. This is illustrated in the following example:

```
type B >: Nothing <: AnyRef
type C >: String
type D <: AnyRef
```

```
type B >: Nothing <: AnyRef
type C >: String <: Any
type D >: Nothing <: AnyRef
```

ExistentialTypeTree TypTree <: Tree

tpt: Tree, *whereClauses*: List[Tree]

Existential types are represented with a ExistentialTypeTree. In Scala, there exist two notations for existentials:

```
List[_]
List[T] forSome { type T }
```

Both are represented the same way in the AST, with their full notation:

```
List[_$1] forSome {
  <synthetic> type _$1 >: _root_.scala.Nothing <: _root_.scala.Any
}
List[T] forSome {
  type T >: _root_.scala.Nothing <: _root_.scala.Any
}
```

Note that the existential type tree is stored in another TypeTree's orig member, which is not traversed and transformed by the Scala compiler's Traverser and Transformer classes.

D.3. Other AST Constructs

For Comprehensions

For or sequence comprehensions are a purely syntactic construct; in the AST, they are represented with foreach, withFilter, map, and flatMap calls. The following listing shows some examples.

```
val xs = 1 to 10 toList
val ys = 1 to 10 toList

val coordinates1: List[(Int, Int)] = for(x ← xs; y ← ys) yield (x → y)
// is equal to
val coordinates2: List[(Int, Int)] = xs.flatMap(x ⇒ ys.map(y ⇒ (x → y)))

for(x ← xs if x % 2 == 0) println(x)
// is equal to
xs.withFilter(x ⇒ x % 2 == 0).foreach(x ⇒ println(x))
```

Multiple Assignment

Scala's multiple or parallel assignment syntax is just an abbreviation for a more complex pattern match expression. The following listing shows the desugared form for the call val (a, b) = getPair():

```
def getPair() = (1, 2)

val x$1 = getPair() match {
  case (a, b) ⇒ (a, b)
}

val a = x$1._1
val b = x$1._2
```

The same transformation is also performed when extractors are involved in the assignment:

```
val MyRegex = """(\w)(.*)""".r
val MyRegex(firstGroup, secondGroup) = "Hello"
```

becomes:

```
val MyRegex = """(\w)(.*)""".r
val x$1 = "Hello" match {
  case MyRegex(firstGroup, secondGroup) ⇒ (firstGroup, secondGroup)
}

val firstGroup = x$1._1
val secondGroup = x$1._2
```

Named Arguments

Named arguments are desugared into a series of local values that are then passed in the right order to the method. That is, the following code:

```
def p(first: String, second: Int) = ()

p(second = 42, first = "-")
```

is represented as:

```
def p(first: String, second: Int): Unit = ()
{
  val x$1 = 42
  val x$2 = "-"
  Account.this.p(x$2, x$1)
}
```

This is described in the first Scala Improvement Document [Ryt09].

E. Advanced Scala Features

This documentation does not contain an introduction to the Scala language, so an understanding of the basic concepts is assumed. This appendix explains some of the more advanced features and patterns of Scala that are used during the explanations in this thesis.

E.1. Path Dependent Types

Path dependent types are best explained with an example (taken from Programming Scala [OSV08]). We have an Animal class with an abstract type member called SuitableFood which is then defined in the subclasses to a suitable type.

```
abstract class Food

abstract class Animal {
  type SuitableFood <: Food
  def eat(food: SuitableFood)
}

class Grass extends Food
class Cow extends Animal {
  type SuitableFood = Grass
  def eat(food: Grass) {}
}

class DogFood extends Food
class Dog extends Animal {
  type SuitableFood = DogFood
  def eat(food: DogFood) {}
}
```

This now prevents us from feeding the wrong kind of food to our animals:

```
scala> val bessy = new Cow
bessy: Cow = Cow@3fb01949

scala> val lassie = new Dog
lassie: Dog = Dog@46c9220

scala> lassie eat (new bessy.SuitableFood)
<console>:14: error: type mismatch;
 found : Grass
 required: DogFood
        lassie eat (new bessy.SuitableFood)
```

In the context of the Scala compiler, all instances of Tree are dependent on the compiler – that is, impossible to mix trees from different compiler instances.

```
trait Trees {
  ...
  abstract class Tree extends Product {
  ...
}
```

The Scala Refactoring library follows this design, all functionality that operates on compiler dependent types is in traits that have a compiler instance as an abstract member, like for example in the AbstractPrinter or the Indexes:

```
trait AbstractPrinter {
  val global: scala.tools.nsc.interactive.Global
  import global._
  ...
}

trait Indexes {
  val global: scala.tools.nsc.interactive.Global
  ...
}
```

The user of the refactoring library then has to provide this abstract member and all implemented traits share the same instance. In the automated tests, this is done by the CompilerProvider trait.

E.2. Stackable Traits

Stackable traits are related to the decorator design pattern, except that they do not decorate objects at run-time but traits at compile-time. Let us take a look at an example.

Assume that we have a trait that allows us to log events and an implementation that logs to the standard output:

```
trait Logging {
  def log(severity: Int, msg: String): Unit
}

class ConsoleLogger extends Logging {
  def log(severity: Int, msg: String) = {
    println("%d: %s" format (severity, msg))
  }
}
```

Now we want to have a logger that only logs events of a certain severity, or one that filters the messages. We could subclass ConsoleLogger, but there are potentially many concrete loggers, and we want the user of the logger to be able to combine these features as he likes. This is where stackable traits are useful. Stackable traits use the abstract override modifier to override an abstract method and are allowed to call super in the implementation, even though the super method is not implemented.

```
trait LogOnlyErrors extends Logging {
  abstract override def log(severity: Int, msg: String) {
    if(severity >= 3)
      super.log(severity, msg)
  }
}
```

When we instantiate a new ConsoleLogger with LogOnlyErrors, the abstract override method in LogOnlyErrors overrides the log method in ConsoleLogger. We can also create more such stackable traits and combine them.

```
trait TreatAllAsErrors extends Logging {
  abstract override def log(severity: Int, msg: String) {
    super.log(3, msg)
  }
}
```

Because of Scala's trait linearization, the order of the stackable traits is significant and allows further combinations, as shown below.

```
scala> val logger = new ConsoleLogger with TreatAllAsErrors with LogOnlyErrors
logger: ConsoleLogger with TreatAllAsErrors with LogOnlyErrors = $anon$1@8aee908

scala> logger.log(1, "Something insignificant happened.")

scala> logger.log(4, "A critical error, severity 4!")
```

```
3: A critical error, severity 4!

scala> val logger2 = new ConsoleLogger with LogOnlyErrors with TreatAllAsErrors
logger2: ConsoleLogger with LogOnlyErrors with TreatAllAsErrors = $anon$1@2a788315

scala> logger2.log(1, "Something insignificant happened.")
3: Something insignificant happened.

scala> logger2.log(1, "A critical error, severity 4!")
3: A critical error, severity 4!
```

E.3. Implicit Conversions

Implicit conversions (also known as the *pimp my library* pattern) can be used to (seemingly) add new methods to an existing class. Assume that we are working with currencies and have a class to represent Swiss francs:

```
class SwissFrancs(private val amount: Int) {
  def + (other: SwissFrancs) = new SwissFrancs(amount + other.amount)
  override def toString = "CHF "+ amount
}
```

Thanks to Scala's support for methods with operator names, we can add instances of Swiss francs using +, but we still have to construct them verbosely. With an implicit conversion, we can add a francs method to Int that makes for a very readable syntax:

```
implicit def intToSwissFrancs(i: Int) = new Object { def francs = new SwissFrancs(i) }

scala> 5.francs + 10.francs
res1: SwissFrancs = CHF 15
```

This is also how Scala enriches Java's built-in types or why we can form tuples from any two objects using →:

```
class ArrowAssoc[A](x: A) {
  ...
  def → [B](y: B): Tuple2[A, B] = Tuple2(x, y)
}

implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] = new ArrowAssoc(x)

scala> "answer" → 42
res2: (java.lang.String, Int) = (answer,42)
```

E.4. Self Type Annotation

Scala allows the programmer to specify an alias for this inside the current class. This is often useful to access the outer instance from an inner class where this is already bound to the inner class.

```
class OuterClass(val name: String) {
  outerclass =>

  class Inner {
    println("I'm the inner class of "+ outerclass.name)
  }
}
```

The self type annotation allows us to annotate this self type with additional types and are a way to describe dependencies the class or trait has. For example, if we have a class that uses some kind of service interface, we can specify the service interface with a self type annotation:

```
trait Service {
  def callWebservice ...
}

class ServiceUser {
  self: Service =>

  callWebservice ...
}

val myService = new ServiceUser with SomeServiceImplementation
```

The user of ServiceUser then has to instantiate it with a suitable implementation of Service to make the program compile. In most cases, one could also just let ServiceUser inherit from Service, but using a self type annotation is conceptually clearer than inheritance.

E.5. Package Nesting

It is a common misconception that Java supports nested packages; they are only nested in the file system, but the language itself has no notation of nested packages. In Scala on the other hand, packages can be nested. The following two listings represent different compilation units:

```
package com.mycompany.project
package pd

class Student
```

```
package com.mycompany.project
package ui

import pd.Student
```

Note how the import statement does not have to specify the fully qualified name but can simply import pd.Student because both compilation units are in the com.mycompany.project package.

F. License

The project is licensed under the Scala license:


```
Copyright (c) 2002-2010 EPFL, Lausanne, unless otherwise specified.
All rights reserved.

This software was developed by the Programming Methods Laboratory of the
Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland.

Permission to use, copy, modify, and distribute this software in source
or binary form for any purpose with or without fee is hereby granted,
provided that the following conditions are met:

   1. Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

   2. Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.

   3. Neither the name of the EPFL nor the names of its contributors
      may be used to endorse or promote products derived from this
      software without specific prior written permission.


THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ‘‘AS IS’’ AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

Bibliography

[Abs10] AbsTypeTree.scala. http://www.scala-lang.org/archives/downloads/distrib/files/nightly/docs/library/scala/reflect/generic/Trees$AbsTypeTree.html, Archived at http://www.webcitation.org/5rEs36a6u, 2010.

[Can10] Aemon Cannon. ENSIME – the ENhanced Scala Interaction Mode for Emacs. http://wiki.github.com/aemoncannon/ensime/, Archived at http://www.webcitation.org/5r2go6zZZ, 2010.

[CFS07] Thomas Corbat, Lukas Felber, and Mirko Stocker. Refactoring support for the eclipse ruby development tools. Technical report, Institute for Software, HSR – University of Applied Sciences Rapperswil, 2007.

[CFSS07] Thomas Corbat, Lukas Felber, Mirko Stocker, and Peter Sommerlad. Ruby refactoring plug-in for eclipse. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 779–780, New York, NY, USA, 2007. ACM.

[Emb10] Richard Emberson. Using the refactoring library to generate source code. Private conversation, 2010.

[EPF08] EPFL. Scala on Microsoft .NET. http://www.scala-lang.org/node/168, Archived at http://www.webcitation.org/5meZVGY6N, 2008.

[Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Fow01] Martin Fowler. Crossing refactoring's rubicon. http://martinfowler.com/articles/refactoringRubicon.html, Archived at http://www.webcitation.org/5mjVmnOci, 2001.

[Fre06] Leif Frenzel. The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. http://www.eclipse.org/articles/Article-LTK/ltk.html, Archived at http://www.webcitation.org/5qtSJlLlY, 2006.

[GZS07] Emanuel Graf, Guido Zgraggen, and Peter Sommerlad. Refactoring support for the c++ development tooling. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 781–782, New York, NY, USA, 2007. ACM.

[KKKS08a] Martin Kempf, Reto Kleeb, Michael Klenk, and Peter Sommerlad. Cross language refactoring for eclipse plug-ins. In *WRT '08: Proceedings of the 2nd Workshop on Refactoring Tools*, pages 1–4, New York, NY, USA, 2008. ACM.

[KKKS08b] Michael Klenk, Reto Kleeb, Martin Kempf, and Peter Sommerlad. Refactoring support for the groovy-eclipse plug-in. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 727–728, New York, NY, USA, 2008. ACM.

[Lin10] Eclipse Linked Mode UI. http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/jface/text/link/package-summary.html, Archived at http://www.webcitation.org/5qtS88E9U, 2010.

[MHPB09] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[MKF06] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23:76–83, 2006.

[MPO08] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in scala. Technical report, Department of Computer Science, Katholieke Universiteit Leuven, 2008.

[Net09] Scala Plugins for NetBeans. http://wiki.netbeans.org/Scala#Scala_Plugins_for_NetBeans, Archived at http://www.webcitation.org/5msestVuy, 2009.

[Ode09a] Martin Odersky. Scala compiler internals. http://www.scala-lang.org/node/598, Archived at http://www.webcitation.org/5meXOPQBx, 2008-2009.

[Ode09b] Martin Odersky. Scala by example. Technical report, EPFL, 2009.

[Ode10] Martin Odersky. A Postfunctional Language. http://www.scala-lang.org/node/4960, Archived at http://www.webcitation.org/5qtSAbR3z, 2010.

[OO09] Martin Odersky and Others. The Scala Language Specification. 2009.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 2008.

[Ryt09] Lukas Rytz. Sid1 – named and default arguments in scala 2.8. Technical report, EPFL, 2009.

[Sab10] Miles Sabin. Scala IDE for Eclipse. http://www.scala-ide.org, 2010.

[SH09] Michael Schinz and Philipp Haller. A scala tutorial for java programmers. Technical report, EPFL, 2009.

[Slo10] A. M. Sloane. Lightweight Language Processing in Kiama and Scala. Presentation at Scala Days 2010, 2010.

[SSR10] Structural Search and Replace: What, Why, and How-to. http://www.jetbrains.com/idea/documentation/ssr.html, Archived at http://www.webcitation.org/5r5TLN8x1, 2010.

[Sto09] Mirko Stocker. Scala refactoring term project. Technical report, Institute for Software, HSR – University of Applied Sciences Rapperswil, 2009.

[Sto10a] Mirko Stocker. Automated Refactoring for Scala. http://days2010.scala-lang.org/node/138/141, Archived at http://www.webcitation.org/5rBZ7ogiz, 2010.

[Sto10b] Mirko Stocker. Refactor – How are Refactorings related? http://refactor.ch, 2010.

[Str10] The Stratego Language. http://strategoxt.org/Stratego/StrategoLanguage, Archived at http://www.webcitation.org/5r1MgRjlD, 2010.

[SZCF08] Peter Sommerlad, Guido Zgraggen, Thomas Corbat, and Lukas Felber. Retaining comments when refactoring code. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 653–662, New York, NY, USA, 2008. ACM.

[Tre10] TreeDSL Scala Source File. https://lampsvn.epfl.ch/trac/scala/browser/scala/trunk/src/compiler/scala/tools/nsc/ast/TreeDSL.scala, Archived at http://www.webcitation.org/5qPR9IALF, 2010.

[ZP09] Sergey Zhukov and Alexander Podkhalyuzin. Scala Plugin for IntelliJ IDEA. http://www.jetbrains.net/confluence/display/SCA, Archived at http://www.webcitation.org/5msesJVev, 2009.