Extensible Code Contracts for Scala

Master’s Thesis Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

By: Rokas Matulis
matulisr@student.ethz.ch

Supervised by: Malte Schwerhoff
Prof. Dr. Peter Müller

Date: September 1, 2012
Abstract

Code contracts is a way to attribute the programs with formal the specifications, based on which the verification of those programs can be performed. The focus on software verification is increasing and the mainstream languages start to adopt this technique. Scala is an object oriented and functional programming language that is very expressive and has a strong type system. Hence, developing a code contracts solution for Scala is both interesting and challenging.

In this thesis we build the basic contracts library for Scala which provides the means to express pre- and post-conditions for the methods. The main work is done in the compiler plugin which performs compile time rewriting of the program in order to support the inheritance of contracts and old expressions.

Providing contracts the traditional way for a compiled library is not possible without the recompilation and redistribution. To solve this problem our rewriting tool enables the separation of contracts from the corresponding implementation. Such contracts can then be edited, distributed and used separately from the implementation. Code contracts for Scala is a starting point for further research so it has been designed with the extensibility in mind and could be supplemented with new types of contracts such as object invariants.
## Contents

1 Introduction .................................................. 6  
  1.1 Overview ................................................. 6  
  1.2 Motivation ............................................... 6  
    1.2.1 Code Contracts ..................................... 6  
    1.2.2 Scala ............................................... 8  
    1.2.3 Contracts-only ..................................... 8  
  1.3 Outline .................................................. 9  

2 Code Contracts ............................................ 10  
  2.1 Subtyping Semantics .................................... 10  
    2.1.1 Effective Precondition ......................... 12  
    2.1.2 Effective Postcondition ....................... 13  
  2.2 Well-definedness ....................................... 15  

3 Contracts Library API .................................... 17  
  3.1 Design .................................................. 17  
  3.2 Architecture ........................................... 20  
  3.3 Extensibility .......................................... 22  
  3.4 Other Approaches ..................................... 24  

4 Scala Compiler Background ................................. 25  
  4.1 Constructs .............................................. 25  
    4.1.1 Trees .............................................. 25  
    4.1.2 Symbols .......................................... 26  
    4.1.3 Types ............................................. 29  
    4.1.4 Names ............................................. 30  
    4.1.5 Position ......................................... 30  
  4.2 Code Synthesis ......................................... 30  

5 Rewriting .................................................. 33  
  5.1 Architecture ............................................ 33  
    5.1.1 Compiler Plugin .................................. 34  
    5.1.2 Contracts Compiler Plugin Architecture .......... 35  
    5.1.3 Design and Extensibility ....................... 37  
  5.2 Contracts .............................................. 43  
    5.2.1 Rearrangement of Method Body .................. 43  
    5.2.2 Pre- and Post-conditions ....................... 47  
    5.2.3 Old Expressions .................................. 52  
    5.2.4 Workflow ......................................... 61  
  5.3 Contracts-only ......................................... 62  
    5.3.1 Output ............................................ 63  
    5.3.2 Injection ......................................... 70
List of Listings

List of Figures

1  Contracts API class diagram .......................... 21
2  Extended contracts API class diagram with richer conditions .......................... 22
3  Extended contracts API class diagram with invariants .......................... 23
4  Contracts plugin output .......................... 33
5  Contracts plugin architecture .......................... 36
6  Contracts plugin class diagram .......................... 38
7  Contracts plugin class diagram with contracts-only extension .......................... 41
8  Contracts plugin class diagram with possible object invariant extension .......................... 42
9  Old values of new elements of the list .......................... 58
10 High level workflow of main contracts transformer .......................... 61
11 Contracts Scala compiler plugin: contracts-only .......................... 63
12 Workflow of contracts-only output component .......................... 66
13 Workflow of contracts-only injection component .......................... 72
14 Contracts-only injection in the AST .......................... 74

List of Listings

1  Simple example of code contracts .......................... 7
2  Behavioral subtyping .......................... 10
3  Behavioral subtyping rules for methods .......................... 11
4  Effective precondition .......................... 12
5  Effective precondition with stronger overriding precondition .......................... 13
6  Effective pre- and post-conditions .......................... 14
7  Example of the drawbacks of effective pre- and post-conditions .......................... 15
8  Assertions using Scala’s Predef package .......................... 17
9  Usage of code contracts for Scala API .......................... 19
10 Implementation of code contracts for Scala API .......................... 19
11 Usage of code contracts for Scala API with invariants extension .......................... 24
12 Source code for symbol owners example .......................... 27
13 Symbol owners structure after typer phase .......................... 27
14 Source code for an example of symbol owners with Block .......................... 28
15 Symbol owners with Block structure after typer phase .......................... 28
16 Symbol owners with Block as right hand side .......................... 28
17 Symbol owners with Block as right hand side structure after typer phase .......................... 28
18 Rearrangement of a method .......................... 44
19 Rewriting of explicit return statements .......................... 46
20 Original source code to rewrite effective pre- and post-conditions .......................... 48
21 Rewritten effective pre- and post-conditions .......................... 50
22 Rewriting of simple old expressions .......................... 52
23 Rewriting of old expressions in quantifiers .......................... 55
24 Contracts-only output example .......................... 67
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Contracts-only output example</td>
<td>70</td>
</tr>
<tr>
<td>26</td>
<td>Contracts-only injected at call side</td>
<td>71</td>
</tr>
<tr>
<td>27</td>
<td>Implementation of stack without contracts</td>
<td>76</td>
</tr>
<tr>
<td>28</td>
<td>Injection of contracts-only specifications for ...</td>
<td>77</td>
</tr>
<tr>
<td>29</td>
<td>Variable access via super call</td>
<td>79</td>
</tr>
<tr>
<td>30</td>
<td>Method in superclass accessing local variable</td>
<td>79</td>
</tr>
<tr>
<td>31</td>
<td>Injected contracts-only for super call</td>
<td>80</td>
</tr>
<tr>
<td>32</td>
<td>Usage of a library which feels like a native language support</td>
<td>87</td>
</tr>
<tr>
<td>33</td>
<td>Implementation of a library which feels like native language support</td>
<td>87</td>
</tr>
<tr>
<td>34</td>
<td>Both pre- and post-conditions before the method body</td>
<td>88</td>
</tr>
<tr>
<td>35</td>
<td>Implementation of the API with both pre- and post-conditions before method body</td>
<td>89</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Overview

The goal of this Master’s thesis is to develop a basic code contracts library for the Scala programming language, which will enable Scala programmers to provide formal specifications for their programs, where these specifications will be translated into runtime assertion checks. The project consists of several steps and concentrates on building the whole chain of components to enable the tool to work, rather than supporting a wide range of language specification features. The implementation includes enabling preconditions and postconditions for methods as one part of the project, where the other is to provide a possibility to output and use contract specifications separately from the original source code, for which these contracts were specified. Code contracts for Scala is a starting point for further research and is implemented with extensibility in mind to enable the addition of more specific contracts and the building of more sophisticated tools.

1.2 Motivation

1.2.1 Code Contracts

The correctness of software is not a trivial task to implement and an easy property to maintain, while software is both growing and changing. With the introduction of object-oriented design, a lot of attention is concentrated to the reuse of software components, where the consequences of incorrectness are even more devastating. As a way of reducing bugs and increasing reliability the, “Design by Contract” concept was introduced by B. Meyer to target these problems.

“One of the most important properties of a program is whether or not it carries out its intended function”, which, as suggested by C. A. R. Hoare, “can be specified by making general assertions about the values which the relevant variables will take after execution of the program”. These assertions are called postconditions of the program (or parts of it, for instance, a method). “In many cases, the validity of the results of a program will depend on the values taken by the variables before that program is initiated.” These initial assertions are called preconditions of the program (or parts of it). The connection between all these assertions is the following: if a precondition holds before the initiation of a program, then the postconditions will hold upon its completion.

Code contracts is a way to provide formally verifiable specifications for software components with Eiffel being the pioneering language in the field with native support for contract specifications. Since introduction in Eiffel not many other languages have been created that have native support for contracts built into the language, although
many libraries, preprocessors and other tools, which provide the means of writing contract specifications, have been introduced to extend different languages with contracts. For instance, there are tools for the Java language, such as JML [LBR98], which enables contracts via annotation comments and provides both static and runtime verification, as well as contract4j [Wam06], jcontractor [KHB98] and many others providing runtime verification. Microsoft has built the Spec# language [BLS04] and the Code Contracts library [Cor], both of which provide static and runtime verification support. The latter enables developers to write specifications on the .NET platform using regular library calls.

A simple example of code contracts is shown in Listing 1. The method `setMinute` has contract specifications, where the precondition requires a provided value to be between 0 and 59 inclusively, and the postcondition ensures that a field `minute` of a class `Time` is set to the provided value.

```
class Time
  var hour = 0
  var minute = 0
  var second = 0

  def setMinute(m: Int) {
    requires(m >= 0 && m <= 59)
    ensures(minute == m)
    minute = m
  }

  ...
```

Listing 1: Simple example of code contracts

**Verification** There are two different verification approaches. The first is static verification, which is performed at compile time and therefore does not have any impact on efficiency as errors are found and reported before the execution of the program. However, static checking is a complex task which is difficult to implement and requires extensive contracts in order to be able to prove certain properties about a program. The second type is dynamic verification, which is performed at runtime by assertion checking. Although not all properties of the program can be checked at runtime, but this type of verification is less complex, where partial contracts are more useful and can be a strong supplement to the testing of the program.
1.2.2 Scala

Scala is a relatively new language introduced by M. Odersky at the École Polytechnique Fédérale de Lausanne. It has a powerful type system and combines both functional and object oriented programming techniques in one language. Scala is a very expressive language featuring powerful language constructs, such as traits, and concepts, such as actor-based concurrency, which pose additional challenges to verification. Scala does not have built in support for contracts, and existing approaches, such as general assertions, are not sufficient, for instance, to enforce behavioral subtyping or to support old-expressions.

Typically, the recommended way of introducing new language expressions in Scala is using libraries, as Scala allows programmers to create natural looking DSLs. Scala’s standard library has `assert`, `require` and `ensuring` constructs where the latter two are supposed to correspond respectively to preconditions and postconditions. However, as these are only library solutions, inheritance is not taken into account and behavioral subtyping is not enforced. In order to fully support code contracts a pure library approach is not sufficient and compile time rewriting has to be performed.

This thesis is a part of a long-term project dedicated to developing a verification framework for all of the major language features of Scala.

1.2.3 Contracts-only

The typical approach to providing contract specifications for a program is to write them together with the implementation. However, adding contracts the traditional way to an already compiled and distributed library requires a recompilation of the library, which is often an undesirable option. Another use case would be those libraries that do not have contracts due to efficiency, however, the users of the latter libraries might still be willing to use contracts for debugging purposes.

To provide a solution for these problems, a part of this project is to enable a possibility to extract contract specifications from the implementation into contracts-only files. This way contracts can be used and distributed independently from the implementation for which they are specified. The next step is to enable injection of these contracts-only files into the program where they are needed.

Having the possibility to write, distribute, and use contract specifications separately from the corresponding implementations delivers great flexibility. This enables the possibility to provide contracts for existing libraries and also separates the concerns of implementation and design even more.
1 Introduction

1.3 Outline

The rest of the report is structured as follows: in Section 2 we introduce the theoretical background of code contracts, then in Section 3 we explain the design of the chosen contracts library API. Section 4 contains information about the Scala compiler, which is necessary in order to fully understand the following Section 5 where we explain how the contracts compiler plugin works and how the contract specifications are handled during compilation. Finally, the concise list of limitations is given in Section 6 followed by Section 7 where we conclude our work and specify the possibilities for future work. Appendix A explains how a different contracts library API could be built.
2 Code Contracts

In this section the necessary information regarding code contracts is explained, including behavioral subtyping and well-definedness rules.

2.1 Subtyping Semantics

For an object of type $S$ to be a subtype of type $T$ not only it has to be implemented syntactically, meaning $S$ must provide an interface at least that of $T$, but also semantically, meaning $S$ must have the behavior at least that of its supertype. This way “the objects of the subtype ought to behave the same as those of the supertype as far as anyone or any program using supertype objects can tell” [LW94]. So all the properties that can be proved about the supertype should be preserved by the subtype. To enforce this, code contract specifications are introduced, and they can be expressed using pre- and post-conditions for methods, as well as invariants and history constraints for objects. Having these specifications one can show that type $S$ is a subtype of $T$ if invariant and history constraints of $S$ imply those of $T$. An example of such behavioral subtyping is shown in Listing 2.

```
1 class Super {
  2   val n: Int = 0
  3
  4   // invariant n >= 0
  5
  6   def add(x: Int) {
  7     requires(x >= 0)
  8     ensures(n >= x)
  9     n += x
 10   }
 11 }
 12
 13 class Sub extends Super {
 14   val m: Int = 1
 15   override val n = 1
 16
 17   // invariant n > 0 && m > 0
 18   // implies the invariant of Super
 19
 20   override def add(x: Int) {
 21     requires(true)
 22     ensures(m >= old(m) && m > x && n > m)
 23     m += abs(x)
 24     n += m + 1
 25   }
 26 }
```

In this thesis only contracts for methods are introduced, so the concentration will be focused on preconditions and postconditions of the methods.

Behavioral subtyping rules for methods state that overriding methods of subtypes may have:

- **weaker preconditions**, i.e. $P \Rightarrow P'$
- **stronger postconditions**, i.e. $Q' \Rightarrow Q$

than the overridden methods of the supertype.

In other words, preconditions are contravariant, so the precondition of an overridden method of the supertype has to imply the precondition of the overriding method of the subtype. Postconditions, on the other hand, are covariant, meaning that the postcondition of the overriding method of the subtype has to imply the postcondition of the overridden method of the supertype. An example is shown in Listing 3.

```scala
class Super {
    def foo(x: Int) {
        requires(x > 0)
        ensures(x > 1)
        ...
    }
}

class Sub extends Super {
    override def foo(x: Int) {
        requires(x >= 0) // x > 0 => x >= 0
        ensures(x > 10)  // x > 10 => x > 1
        ...
    }
}
```

Listing 3: Behavioral subtyping rules for methods

The aim of this project is to provide runtime assertions for preconditions and postconditions which support behavioral subtyping, however, with no static checking to be performed.

At runtime, contracts can only be checked for concrete instances, using the actual evaluated conditions, however, checks for the whole heap, as well as all parameters and
results are not possible. So at this point, numerous problems, related to inheritance of contract specifications, arise\cite{Mul11}, such as how to handle implications between pre- and post-conditions.

One approach would be to disallow an overriding method to declare any additional preconditions, which would enforce the preconditions of a method in the supertype for all of its subtypes. However, this approach is too restrictive, because an overriding method can no longer provide a weaker precondition.

The other approach could be to perform implication checks for pre- and post-conditions, namely $P \Rightarrow P'$ and $Q' \Rightarrow Q$, at runtime. However, this can only be done for a concrete program runs.

Another - more sophisticated - solution exists, which is called effective pre- and post-conditions. It allows any precondition to be specified for an overriding method by automatically weakening it. Although it still has its own disadvantages, but it is more permissive and the implication $P \Rightarrow P'$ always holds, which makes it a better approach, than the other ones explained above, to check contract specifications at runtime.

\subsection{Effective Precondition}

The idea behind effective preconditions is to take a disjunction of all preconditions specified for the method in the current type and all preconditions from the methods which the current method overrides at all points in type hierarchy:

\[ P_{T,m} \lor P_{S,m} \lor P_{SS,m} \lor \ldots, \]

where $P$ - precondition, $T$ - current type, $S$ - Super type, $m$ - method. An example of an effective precondition is shown in Listing 4.

```scala
class Super {
  def foo(x: Int) {
    requires(x >= 0)
    ...
  }
}

class Sub extends Super {
  override def foo(x: Int) {
    requires(x != -5)
    // effective precondition: x >= 0 || x != -5
    ...
  }
}
```

Listing 4: Effective precondition
This way the precondition is automatically weakened, by making it sufficient for at least one of the declared preconditions (in the inheritance tree) to hold. However, one has to pay attention that defining a stronger precondition is not disallowed, i.e., a stronger precondition in an overriding method can be defined, but as it is disjuncted with the inherited ones, it gets automatically weakened. An example is shown in Listing 5.

```java
1 class Super {
2   def foo(x: Int) {
3     requires(x >= 0)
4     ...
5   }
6 }
7
8 class Sub extends Super {
9   override def foo(x: Int) {
10      requires(x > 0)
11      // effective precondition: x >= 0 || x > 0
12      ...
13   }
14 }
```

Listing 5: Effective precondition with stronger overriding precondition

We can see in Listing 5 that the precondition of Super implies the calculated effective precondition in Sub, and so it makes class Sub a behavioral subtype of class Super.

2.1.2 Effective Postcondition

Behavioral subtyping rules for postconditions can be characterized as $Q' \Rightarrow Q$, meaning that they can be strengthened in the overridden methods. Then a postcondition of an overriding method would be the conjunction of postconditions from the current and all overridden methods:

$$Q_{T.m} \land Q_{S.m} \land Q_{SS.m} \land ...,$$

where $Q$ is postcondition.

However, the postcondition of a method must hold at the end of it only if the precondition held at the beginning. Which gives the following interpretation of postcondition:

$$old(P_{T.m}) \Rightarrow Q_{T.m}$$

In the case of inheritance and effective preconditions, all preconditions for current type and all preconditions from the methods, which the current method overrides at all points
in type hierarchy, are considered, and so has to be done with postconditions. Thus, the effective postcondition is the conjunction of implications shown above for all types in the type hierarchy:

$$\text{(old}(P_{T,m}) \Rightarrow Q_{T,m}) \land (\text{old}(P_{S,m}) \Rightarrow Q_{S,m}) \land (\text{old}(P_{SS,m}) \Rightarrow Q_{SS,m}) \land \ldots$$

An example of effective pre- and post-conditions is shown in Listing 6:

```java
1 class Super {
2     def foo(x: Int) {
3         requires(x >= 0)
4         ensures(x > 1)
5         ...
6     }
7 }
8
9 class Sub extends Super {
10     override def foo(x: Int) {
11         requires(x != -5)
12         // effective precondition: x >= 0 || x != -5
13         ensures(x > 10)
14         // effective postcondition:
15         // ( old(x >= 0) => x > 1 ) &&
16         // ( old(x != -5) => x > 10 )
17         ...
18     }
19 }
```

Listing 6: Effective pre- and post-conditions

It can be seen that the effective postcondition of method `foo` in `class Sub` implies the postcondition of overridden method in `class Super`, which makes `class Sub` a behavioral subtype of `class Super`.

However, effective pre- and post-conditions still have their disadvantages:

- Stronger preconditions can be specified for overriding method and they will only be automatically weakened
- It might be counterintuitive that methods might crash although the precondition was meant to prevent that. See the example in Listing 7
- Programmers need to be aware of effective pre- and post-conditions
class Super {
  var x: Int = 0
  var y: Int = 0

  def foo {
    requires(true)
    ensures(x > 0)
    ...
  }
}

class Sub extends Super {
  override def foo() {
    requires(x != y)
    // effective precondition: x != y || true
    ensures(x > 0 && y > 0)
    // effective postcondition:
    // ( old(x != y) => (x > 0 && y > 0) ) &&
    // ( old(true) => (x > 0) )
    assert(x != y) // assertion can fail
    y = -1
  }
}

object Test {
  val strange = new Sub
  strange.x = 1; strange.y = 1
  strange.foo
  assert(strange.x > 0 && strange.y > 0) // assertion fails
}

Listing 7: Example of the drawbacks of effective pre- and post-conditions

Even though effective contracts have their drawbacks, they are still a much more permissive solution for runtime verification, than disallowing redefining contracts in overriding methods, in order to support behavioral subtyping.

2.2 Well-definedness

For contracts to be valid they have to be well-defined, i.e., comply with the following rules:
1. A precondition is a requirement that has to be fulfilled by the caller of a method, so all \textit{field accesses and method calls used in the precondition have to be accessible by the caller} as well. More precisely:

   (a) If the method is private, fields and methods with any modifier are allowed in the precondition.

   (b) Otherwise, only public fields and public methods can be mentioned. It is important to note that in Scala protected methods can be overridden with public ones. In this case, if there were protected fields or methods used in the precondition of the supertype, they would have narrower visibility in the subtype and they wouldn’t be accessible by the caller.

2. Contract expressions must be pure, as contracts are meant to only specify the program and not to modify its behavior. In order to maintain pureness, all expressions in contract specifications must be side-effect free, referentially transparent and also convergent, meaning they must terminate.

3. Contracts usually include special \texttt{old} and \texttt{result} expressions which enable developers to express the relations between pre- and post-state, as well as properties of the result of the method. However, their usage is disallowed anywhere else but in postconditions.

4. Furthermore, the \texttt{old} expression is idempotent, however, due to implementation issues, the usage of it in a parameter of another \texttt{old} expression is disallowed.

5. Usage of \texttt{result} in \texttt{old} expression’s argument is disallowed, because \texttt{result} represents the product of a method which is not existent in the old space.
3 Contracts Library API

In this section the API of our contracts library is introduced, with justifications of the design choices and explanations of usage. The goal is to provide the possibility to express pre- and post-conditions, as well as to be able to refer to old expressions of values and the result of the method in the postcondition. Whenever possible, the existing methods from the Scala language were used, for instance, quantifiers `forall` and `exists`.

3.1 Design

Several factors were in play when choosing the design of the library, where the most balanced approach had to be selected. The factors, not in any order of importance, are the following:

1. *Ease of use.* The way of expressing pre- and post-conditions should be intuitive and clear.

2. *Extensibility.* The API should be easily extensible to provide richer interface and new types of contracts.

3. *Avoidance of rewriting aggravation.* To enable contracts, the compile time rewriting will be performed, so it is necessary to draw attention to this part, so that the chosen API solution would not overcomplicate the rewriting.

Scala provides a certain way of expressing pre- and post-conditions for methods, so before starting to design our own library we will shown what are the existing possibilities and why they are not sufficient.

Scala provides methods `require` and `ensuring` in package `Predef`, which is by default imported into every Scala source file. These expressions enable to write assertions for method’s entry and exit points. An example of their usage is shown in Listing 8.

Listing 8: Assertions using Scala’s `Predef` package.

```scala
class Account {
  var balance: Int = 0
  def withdraw(amount: Int): Int = {
    require(amount >= 0 && amount <= balance)
    balance -= amount
    amount
  }
  ensuring {result => // a function from result
    result == amount // of the method to boolean condition
  }
}
```
The precondition is written at the top of the method body using method `require`, whereas the postcondition is expressed within method `ensuring`, which must be placed right after the body of the method, and necessarily at the same line. This approach is a pure library solution, so it does not require any compile time rewriting, and interestingly, it provides the way of referring to the result of the method. However, it still has numerous drawbacks:

- The syntax is not very consistent, as precondition is written at the top of method within its body, whereas postcondition at the end, but after the method body.

- As it is a pure library solution, it is necessary to place a precondition at the top and a postcondition at the bottom of the method, where they are actually checked at runtime. In Eiffel [Mey92b] the placement of pre- and post-conditions is similar. However, contract specifications are usually considered to be part of the interface, which means part of the declaration of the method, so the preferred way would be to have both pre- and post-conditions written at the beginning of the method, which is the way it is done in Spec# [BL04], JML [LBR98], Ms Code Contracts [Cox] and many others.

- Although it is possible to refer to the result of the method, the old values of expressions can not be referenced in postcondition, which is necessary to express the change of the state, for instance, it is not possible to specify that \( m \) increased in the example in Listing 8. To enable old expressions compile time rewriting has to be done.

- As explained in section 2.1, subtyping is not only syntactical, but also semantical relation, and the way to express the semantics of a program, or part of it, like a method, is by specifying contracts. Using pure library approach to enable contracts, inheritance of them cannot be enabled. To overcome this essential deficiency compile time rewriting has to be done.

In Scala, the preferred way of introducing new building structures to the languages is by building new libraries in such a way that using then would feel like it is a native component of the language, which is possible having the expressiveness of Scala. However, the pure library solution can not support inheritance of contracts and so can not enforce behavioral subtyping. In order to provide the latter, compile time rewriting of contracts has to be done.

During the design of contracts library API a few different approaches were considered. The possible but not chosen solutions, that could possibly seem more like a native language support, are shown in Appendix A.

Taking into account that the main part of the whole process of enabling contracts is going to be rewriting, it draws towards building such a library that would alleviate this part. So simpler API is used, however, without sacrificing the intent and focus of contracts.
being part of the declaration rather than an implementation. The usage example is shown in Listing [9]

```scala
import Conditions._

class Account {
  var balance: Int = 0

  def withdraw(amount: Int): Int = {
    requires(amount >= 0 && amount <= balance)
    ensures(balance == old(balance) - amount &&
            result == amount)
    balance -= amount
    amount
  }
}
```

Listing 9: Usage of code contracts for Scala API

We can see that `requires` and `ensures` are now simply method calls within specified method’s body. `old` and `result` expressions can be used in postcondition to refer to old values of expressions and the result of the method respectively.

The preferred way is to keep contracts at the top of method body, so that it would be as part of method signature. Although it is not mandatory and contracts could be written anywhere in the method, for instance, preconditions at the beginning and postconditions at the end of method body, however, the return value of method is not rearranged. This means that if method’s return value is `Unit` and the last line of the method is `ensures` clause, then all is correct, but if the return type is not `Unit` then method will not compile.

The implementation of such library is shown in Listing [10]

```scala
object Conditions {
  case class PreconditionException(message: String) extends Exception
  case class PostconditionError(message: String) extends Error

  @elidable(ASSERTION)
  def requires(condition: => Boolean): Unit =
    if (!condition) throw PreconditionException('Precondition failed')

  @elidable(ASSERTION)
  def ensures(condition: => Boolean): Unit =
    if (!condition) throw PostconditionError('Postcondition failed')

  def old[T](value: T): T
}
```

19
Both pre- and post-conditions are provided simply just as methods. The `old` and `result` expressions are also methods, however, they are just pure stubs to represent the semantic meaning, and the whole empowering is done during compile time rewriting.

Moreover, `requires` and `ensures` expressions are annotated to be elidable, which means that the bodies of the methods may be excluded from compiler-generated bytecode [OCT2]. The argument to the annotation sets the priority level at which the methods should be excluded, which in our case is the level of assertions. The control of the level of elidable is done by passing `-Xelide-below:<arg>` flag to the Scala compiler, in which case, calls to methods marked elidable (as well as the method body) will be omitted from generated code if the priority given to the annotation is lower than that given to the compiler.

This way the user is able to compile a program without contract assertions and no special rewriting is necessary. As `old` and `result` can only be used within postcondition, they don’t need to be elidable, as they will be removed together with surrounding `ensures` clause.

Additionally, two contracts specific assertion types are specified, namely `PreconditionException` and `PostconditionError`. The reasoning that the former is exception and the latter - an error is the following:

- **PreconditionException** - if the user doesn’t satisfy some of the conditions for the operation they request, the standards of the operation are not complied with. Hence an exception is thrown and not an error, as nothing went wrong yet inside of the specified operation.

- **PostconditionError** - if the postcondition fails, it means that the precondition held and everything from outside was satisfied, but something went wrong in the implementation, hence an error and not an exception.

### 3.2 Architecture

This thesis is only a starting point to enable code contracts for Scala, so currently it only provides pre- and post-conditions for methods. The current architecture of the library is shown in Figure[1]

Traits are the main building component in the API and the idea is to provide all functionality in these traits because multiple traits can be inherited and mixed in so it provides the best reuse. And for developers using the API, the idea is to provide (singleton)
objects at the bottom of the inheritance hierarchy, which would provide easy access to all necessary constructs without a need to instantiate anything.

The top level `trait Contracts` is a base type for all contracts. All specific contracts should inherit this trait like `trait Conditions`, which provides the API for method contracts. The bottom-most `object Conditions` is just an empty `object` that inherits all functionality and provides easy access for the clients.

Pre- and post-conditions are specified using methods

\[
\text{requires(condition: } \Rightarrow \text{ Boolean): Unit}
\]
\[
\text{ensures(condition: } \Rightarrow \text{ Boolean): Unit}
\]

that take a boolean condition as a by-name parameter. The difference between a by-value and a by-name parameter in Scala is that the latter allows having lazily evaluated method parameters. Although this is not necessary for rewriting, it provides easier manipulation of conditions on the library side, where it can be decided when and whether to evaluate the given condition.

Although now, in order to disable contracts’ runtime assertions, one has to provide the Scala compiler an elidable flag with the appropriate value (as can be seen in Listing 9), the by-name parameter would allow to control it in the implementation of `requires` and `ensures` given such a need arises, for instance, if elidable annotation is removed from the Scala API, we can still control the enabling and disabling of contracts from the library side, rather than performing a special rewriting, which is always a more complex solution.

There are two more constructs in `trait Conditions`. One is the method
which allows to refer to old values of expressions. This method takes an expression of any type and returns the value of that argument which it had at the beginning of the method. And another is the method

\[\text{result}[T]: T\]

which allows to refer to the result of the specified method. It takes no arguments and its return type is the same as the return type of the enclosing method. Both of these functions are applied for a certain type, and as \text{old} takes an argument of that type then type inference can infer it, however, that’s not the case with \text{result}, type cannot always be inferred so users have to provide it explicitly.

### 3.3 Extensibility

As already mentioned before, this project is a starting point of contracts for Scala, so it has to be extensible. An example of how richer interface for pre- and post-conditions could be added is shown in Figure 2.

Figure 2: Extended contracts API class diagram with richer conditions. All of the grey filled components symbolize extensions.

Pre- and post-conditions with a user-specified message to be displayed upon the failure of a certain condition can be introduced by adding a method

\[\text{requires}(\text{condition}: \Rightarrow \text{Boolean}, \text{message}: \text{String}): \text{Unit}\]

for preconditions and
ensures(condition: => Boolean, message: String): Unit

for postconditions in trait Conditions. To add exceptional postconditions, a method

ensures(condition: => Boolean, exception: Exception): Unit

would be added to the same trait Conditions which provides contract specifications for methods.

An example of how object invariants could be added to the API of the library is shown in Figure 3.

![Diagram showing extended contracts API class diagram with invariants](image)

Figure 3: Extended contracts API class diagram with invariants

One way of adding invariants would be to introduce a base trait Invariant which, like all contracts, extends trait Contracts and would provide the functionality necessary for all types of invariants. Then, as there are loop and object invariants, two specific traits, trait LoopInvariant and trait ObjectInvariant, could be introduced as subtypes of trait Invariant and would provide more specific contracts if necessary. Finally, object LoopInvariant and object ObjectInvariant should be added to provide clients easy access to all the constructs implemented in inherited traits.

An example of a possible usage of invariants in a program is shown in Listing 11.

---

1In terms of better reuse old and result could be moved to dedicated traits, however, these methods are nothing more than just stubs and have no implementation whatsoever, they are completely rewritten at compile time.
3 Contracts Library API

```scala
import ObjectInvariant._

class Account {
  var balance: Int = 0

  invariant(balance >= 0)

  def withdraw(amount: Int) = ...
  def deposit(amount: Int) = ...
}
```

Listing 11: Usage of code contracts for Scala API with invariants extension

A compile time rewriting (if necessary) would need to be implemented to enable the functionality of new constructs introduced above.

3.4 Other Approaches

There are also other ways to express contracts in Scala without any compile time rewriting, such as using the before mentioned `require` and `ensuring` methods in the Scala library. Another solution is to provide specifications separately as traits as shown in “Contracts for Scala”[Ode10]. Interestingly, traits-as-contracts allow several different specifications for a single implementation since different specification traits can be mixed in. Even more, specifications could be added to a certain type’s declaration as well as mixed in at instantiation step. However, not only do these solutions require writing more boilerplate code, but more importantly they do not allow inheritance of contracts in overriding methods (without super calls), and in order to enable that, compile time rewriting has to be done. On the other hand, these solutions do not interfere with our solution, they can be combined and used together if the user wishes so.
4 Scala Compiler Background

This section contains a high level explanation of Scala compiler internals, which are relevant in order to understand the implementation details in subsequent chapters. In particular, the introduction of how trees, symbols, types and their relationships are built in the AST, as well as some other Scala compiler specific notions, will be covered.

It is assumed that the reader has a basic understanding of trees, types and symbols. More precisely, it should be known that source code within a compiler is represented via the abstract syntax tree (AST) data structure, where every node in the tree corresponds to some variable, statement or other language construct. Also types, that represent the kind of language constructs. And symbols which represent declarations of identifiers, and contain information such as type, class hierarchy, position in source code and much more.

4.1 Constructs

4.1.1 Trees

The Scala compiler has many different subtypes of trees to precisely represent all necessary parts of the Scala language according to its specification [OAC+11]. Tree is the root class of all other trees, and it has Type and Symbol properties, both of which are mutable. Interestingly, not all the trees have a valid Type and Symbol; in that case null or accordingly NoType and NoSymbol is set. Trees can only access their subtrees but not their parents.

Following is the list of some concrete trees that are important for contracts rewriting:

- EmptyTree is a sentinel tree to represent that there is no tree instead of having null. It has no Position, no Type and no Symbol.
- PackageDef represents the definition of a package
- ClassDef represents the definition of a class or trait
- ModuleDef represents the definition of an object
- ValDef represents the definition of a variable using val or var
- DefDef represents the definition of method using def
- Block represents a list of statements surrounded by curly brackets {...}, where the last statement is the return value of Block. Important to note is that Block is one of the types of trees which have no Symbol.
• Function represents a function literal in the form of (...) => {...}
• Apply represents the application of the function in the form of foo()
• Super represents super call
• This represents this reference
• Select represents a selection of a name on a qualifier, which is some other tree. For instance, in method call x.foo() the foo part would be Select tree
• Ident represents the identifier of a variable, method, class or anything else that can be represented by a Name

4.1.2 Symbols

Symbols provide more information than trees. They are introduced for declarations and then the same symbols are referenced from identifiers.

Symbol has Name, Position, Type, flags, annotations and owner properties, and all of them are mutable. Information about an Symbol is then extracted through these properties. As symbols are more interconnected and related to each other than trees, they provide more information about class hierarchy, outer classes and much more, which will be explained in this chapter.

Following is a more detailed description of the properties of a Symbol:

• Name
  There are several different kinds of names that can be returned by a Symbol:

  – encodedName - name encoded by the compiler. For instance, method == would be encoded as $eq$eq
  – decodedName - actual name before encoding. In this case == would be returned instead of $eq$eq
  – fullName - the encoded full path of this Symbol including full package and enclosing symbols names. For instance, method $eq$eq would be something like scala.Boolean.$eq$eq

• Position represents the position of the Symbol in the source code

• Type represents the actual type of the Symbol. Typically, Symbol’s and Tree’s types match.

• annotations - list of annotations, if they exist
• owner - another Symbol which owns the current symbol. It could also be seen as a parent, because there can be only one owner of a Symbol. For more information see examples in Listings 12 - 17.

Symbol has many methods to create new symbols, and many methods in the form of isSomething that provide information about it, for instance, isMethod which returns true only for method symbols.

Following is the set of examples showing how Symbols are built in the Scala compiler for a certain source code. As the contracts plugin works after the typer phase, compiler internal structures are shown as they are built immediately after the typer phase.

Also, to simplify the examples, all additional symbols that come from libraries, like `class Object` or `method +`, have been omitted.

```scala
class A {
  var foo = 0
  
  def bar(x: Int) = {
    val y = foo + x
    y
  }
}
```

Listing 12: Source code for symbol owners example

```scala
class A
  constructor A
  method bar
  value x
  value y
  method foo // synthesized getter
  method foo_= // synthesized setter
  value foo_= // setter’s argument
  variable foo
```

Listing 13: Symbol owners structure after typer phase

Listing 13 displays the built owner hierarchy of symbols for the compiled source code in Listing 12. We can see that the owner of method bar is class A, which also happens to be the top Symbol of the hierarchy. The owner of value x and value y is method bar, and the owner of variable foo is, as expected, class A. The rest of symbols are compiler synthesized ones.

The hierarchy of symbols and their owners after each compilation phase can be seen by providing `-Yshow-syms` flag to Scala compiler.
Interestingly, since a \texttt{Block} tree does not have a \texttt{Symbol}, it can not be an owner of any symbols within it unless it represents the implementation of the method or variable.

```scala
1 class A {
2   def bar(x: Int) = {
3     {
4       def add = 1 + 2
5       val z = add + 3
6       z
7     }
8   }
9 }
```

\textit{Listing 14: Source code for an example of symbol owners with \texttt{Block}}

Again in \textit{Listing 15} we can see the symbol hierarchy of the code from \textit{Listing 14}. The newly added \texttt{method add} and \texttt{value z} are in a block, which is in the \texttt{method bar}, and we see that the owner of these new symbols is set to be \texttt{method bar} and not the enclosing \texttt{Block}.

If we set the block in \texttt{method bar} from \textit{Listing 14} to be the right hand side of some \texttt{value c} like in \textit{Listing 16} then this value \texttt{c} becomes the owner of \texttt{method add} and \texttt{value z} as shown in \textit{Listing 17}.

```scala
1 class A {
2   def bar(x: Int) = {
3     val c = {
4       def add = 1 + 2
5       val z = add + 3
6       z
7     }
8     c
9   }
10 }
```

\textit{Listing 16: Symbol owners with \texttt{Block} as right hand side}
As mentioned before, symbols represent declarations so a block alone is not a declaration. Thus it does not contain any Symbol and cannot be an owner. Once it is attached to some declaration, like a method or variable, which in our example is val c, then the symbols within a block are owned by the Symbol of that declaration.

In Scala, classes can be declared within other classes or traits or objects or even methods and variables. In other words, any declaration in Scala can be placed within any other declaration. However, the top declaration has to be either a class, trait or object. The symbols of these surrounding structures are called enclosing and should not be confused with owners.

A good example of enclosing symbols would be the method add in Listing 16. We have already seen in Listing 17 that its owner is value c, however, it does not match either with its enclosing method, which is method bar, or with its enclosing class, which is class A. Although quite often the enclosing method or class might also be the owner of a Symbol, it is important to understand the difference between them.

Another type of symbol relation can be via class inheritance hierarchy, namely:

- super class - a Symbol of a class, which is inherited by current class
- overridden symbol - a Symbol that is overridden by the current Symbol
- overriding symbol - a Symbol which overrides the current Symbol

Again it is important to separate these symbols from owners and enclosing methods or classes. It is assumed that super classes and overriding is widely known, so no examples are provided to explain this concept.

4.1.3 Types

The Scala compiler has many different concrete type classes to represent all of the types that are possible to express in Scala, where Type is the root class for all types. Types have symbols, and like the latter, are fully introduced after the typer phase.
Type class provides a rich API to extract information about types. For instance, type parameters, if it is a parameterized type, result type, if it is a type of a method, as well as compare types to check for subtyping relations and similar.

4.1.4 Names

Name represents the literal title of a Symbol. Important to note is that it is fine for different symbols to have the same name, for instance, in variable shadowing. Name class is the root for all concrete names, and it provides the means of creating and manipulating names.

4.1.5 Position

“The Position class and its subclasses represent positions of ASTs and symbols. Except for NoPosition and FakePos, every position refers to a source file and to an offset in the source file (its point).” [OC12]

For full compilation of programs, the position of trees and symbols is set to OffsetPosition, which represents only a single point indicating the offset in the source file. Another type of compilation is interactive one, which is used, for instance, by IDEs, and which provides means of obtaining the parsed or typed AST of the source code. For this type of compilation, RangePosition is generated for trees and symbols, and it indicates a start and an end in addition to the offset in source file. Trees with RangePosition need to satisfy the following invariants [OC12]:

- INV1: A tree with an offset position never contains a child with a range position
- INV2: 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.
- INV3: Opaque range positions of children of the same node are non-overlapping (this means their overlap is at most a single point).

To illustrate, lets say a compiler synthesized tree is added to an AST which was obtained from an interactive compiler, then the former tree will have OffsetPosition, which is different from the rest of the trees as they have RangePosition. So according to INV1, compiler synthesized trees can not have children trees of compiled original source code.

4.2 Code Synthesis

We have explained the basic structures that are used by the compiler and that are necessary to understand how the rewriting mechanism works. However, another important
topic is code synthesis. During rewriting many new trees are introduced into the AST that were not present in the original code, and the challenge of this task, since we are running after the typer phase, is that all the following phases expect typing information to exist and to be valid. As we introduce new trees, they have to be fully typed in order to continue compilation without failures.

The simplest approach would be to modify the AST with new synthesized trees, let it go through typer phase again and only then continue with subsequent compilation phases. However, it is not an option as the Scala compiler does not allow such deceptions, which means that trees have to be fully typed manually.

In order to perform this task, we must understand what it means to fully type synthesized code, or in other words, what information must be present and how it should be introduced so that it would be safe to continue the compilation process. The following list, although it may be incomplete, introduces the steps necessary to synthesize a typed tree:

1. The tree must be built from other trees - its children.

2. The tree and its children must be built from concrete appropriate tree subclasses in the Scala compiler. For instance, a method’s tree has to be DefDef, a class’s tree has to be ClassDef, and so forth.

3. Tree structure must be valid, i.e. built according to the rules of the Scala compiler. For instance, an application of method - Apply, should consist of the identifier of the actual method - Select, and its arguments - List[Tree].

4. Trees and their children must have symbols.

5. Symbol hierarchy has to be maintained correct, and after adding new symbols, they have to be introduced into the hierarchy, i.e. new symbols must have owners.

6. Both trees and symbols must have types.

7. Trees are immutable (except for some inner state as explained in section 4.1.1), so they must be copied or created from scratch.

8. The same tree, most of the times, cannot be used more than once in the AST, as it would not be a tree anymore but a graph.

9. Synthesized Symbol of declaration should have a flag <synthetic>.

\[\text{The easiest way to figure out how the correct tree should be composed is to write the source code that one wishes to synthesize, and compile it with -Y:browse:<phase> compiler flag. Given this flag Scala compiler will open the tree browser in the user interface with the AST as built after provided <phase>.}\]
To correctly perform all of these steps every time is rather complicated, because quite thorough knowledge of compiler internals is necessary. As one could already notice, there are two main parts: one is to build the tree structure, and the other is to type it. To alleviate both of these tasks, Scala compiler has:

- **trait TreeDSL** - which can be mixed-in and provides means of creating some trees
- **class TreeGen** - provides means of generating trees attributed with symbols and types. An instance of this class can be accessed via `global.gen`, where `global` is an instance of the compiler (**class Global** is a class of the compiler and its instances are most of the times called `global`).
- **localTyper** - an instance of typer, which is available for every subtype of TypingTransformer, which is a transformer that additionally provides an access to typer.
- **global.typer** - an instance of typer that is available directly from an instance of the compiler, which means it can be accessed by objects that are not subtypes of TypingTransformer.

The first two are dedicated to synthesizing trees, where the second one synthesizes trees that are fully attributed. However, the methods these helpers provide require either valid symbols or trees, which still have to be created manually. Moreover, the generation of only some of the trees is provided and is definitely not exhaustive, so the missing trees have to be synthesized manually.

The last two items are dedicated typers, that provide methods to type trees. However, these typers are based on type inference, meaning that if a plain tree without any type or symbol information is given, it will not be typed and will be returned just as it was. Moreover, during the experience of this thesis, localTyper proved to perform better than global.typer, although the difference cannot be quantified. This means that in order to synthesize the trees it has to be done in a transformer which extends TypingTransformer. The best practice when using the typer is to provide at least the basic types and symbols information for the trees before giving them to be fully typed. However, the more information that is added manually, the more trustworthy is the result of the typer, which in turn means the safer it is to return the AST to the subsequent compilation phases.

To conclude, code synthesis requires some knowledge of Scala compiler internals, first of all to figure out that there are components, which help to perform this task, and secondly, to be able to use them. Moreover, only some trees and only in their basic form can be synthesized using these components, so for the scope of the contracts plugin project, numerous tree generators were created in a similar fashion to that of the compiler.
5 Rewriting

The tool implemented during this master thesis consists of two basic parts: API and rewriting. In this section the latter will be explained. At the beginning it will be shown how the rewriting mechanism is composed, what is its workflow, and how to extend it. Then follows the explanation of the actual rewriting for method pre- and post-conditions as well as related constructs to express them. The final part will talk about contracts-only functionality.

5.1 Architecture

The aim of rewriting is to modify a program at compile time to enable and correctly arrange runtime assertions for contracts. There are several types of rewriting that have to be supported by the tool and they are shown in Figure 4.

![Figure 4: Contracts plugin output. Thick lines represent the output from the original program, thin line - the injection to some other program.](image)

At the top of the figure, there is an original Scala program with contract specifications. Often, it is desired to enable and disable contracts for compiled programs, for instance, contracts are helpful during testing and debugging, but the developer of some library might wish to disable them for release builds due to efficiency. Then, however, the clients of such library will only have the implementation, but not the contracts. To be able to have a better separation of contracts and implementations of the program, the part of this project is to provide a possibility to generate and use contracts-only specifications.

On the left side of Figure 4, we can see that an original program can be compiled into a binary .class file with runtime assertions enabled or removed. As already mentioned
before, the latter is done by providing an elidable flag for the Scala compiler, and there is no dedicated rewriting for this part. As rewriting is a more complex task, we leave the control of this functionality to the library, where, even if elidable flag is removed from the Scala compiler, we still have conditions passed as by-name parameters, which enables to fully control their evaluation from the library.

The other result on the left side of the Figure 4 is a compiled .class file with assertions enabled. For this part the rewriting must be performed, namely to arrange the pre- and post-conditions to be checked accordingly at the beginning and at the end of the method. Also, rewriting of old expressions and the specified method’s result value must be performed.

The remaining part is the contracts-only output and injection as shown on the right side of Figure 4. The idea is to be able to extract contract specifications from an original program into another .scala file, which could then be distributed separately from the implementation. This way, the separation of contracts and their corresponding implementations is increased, which enables, for instance, providing contracts for existing libraries. Moreover, as contracts-only output is a pure .scala file, it can be edited or even written by hand from scratch. Having these separate contracts one has to also be able to inject them into the program which uses a corresponding implementation. This is the last type of rewriting form which is performed by the tool: given separate contract specifications for a particular implementation it injects those contracts for every usage of that implementation.

To sum up, there are three types of rewriting: enabling contract assertions for a compiled original program, outputting separate contracts-only files, and finally injecting them into some program. On the other hand, no special rewriting is done for removing contract assertions.

5.1.1 Compiler Plugin

Rewriting is done at compile time, which means that the Scala compiler has to be extended to provide the means of interpreting code contracts. Even though the Scala compiler is an open source project, meaning it can be extended by anyone to meet his/her needs, the main distribution of compiler does not contain all the custom changes and additions. The problem in the case of code contracts is that it must be possible to distribute this part of the compilation for everyone. Luckily, the Scala compiler provides a way of solving this problem via compiler plugins, which can be developed by third party developers and distributed separately from the compiler. Every Scala developer using the compiler can obtain all plugins they want and tell the compiler to use them.

A compiler plugin itself consists of components where each of them are meant to perform some specific task. There are already several existing base components, like tree traversers or transformers, that can be extended to provide the custom behavior. It is
very favorable for this project where one of its aims is to make it extensible. As these extensible parts come from a compiler itself it is very reliable and future-proof because for every new version of the Scala compiler these base components will be adapted by the compiler itself.

The Scala compiler works in a sequence of phases, where each phase comes after the preceding one has completed. Compiler plugins (or more precisely each component within a plugin) can tell after which phase they must be plugged in, which in turn provides flexibility to be able to develop custom compilation phases that can run in any step of main compilation process.

One of the main functionalities for compiler phases are abstract syntax tree (AST) traversers and transformers:

- **Traverser** can be seen as a function from tree to `Unit`, meaning it can collect information about the given tree and change its mutable state.

- **Transformer** can be seen as a function from tree to tree, meaning that given a tree it can return some other tree which from now on will be placed in the AST instead of the given one.

As these components are extended, only plugin relevant work has to be done, while the whole traversal over the AST is provided by base components from compiler.

### 5.1.2 Contracts Compiler Plugin Architecture

The code contracts for Scala plugin has to perform several different types of rewriting, so for each there is a dedicated component. Their composition and workflow is shown in Figure 5.

In the top left corner we can see the original source code which has to be compiled. Just below it is the whole Scala compiler which at the very end emits the compiled bytecode of the program. Inside the compiler there are several phases, the first of which is the parser, which parses the source code and builds an AST representation of the program. Then follows the namer and typer phases which are uninterruptible, and after those the contracts plugin comes in to perform contracts specific rewriting. The reason why the contracts plugin comes after the typer is that in order to be able to rewrite contract specifications we need to be able to recognize them, and to do that we need the trees to be typed so that based on type information we could distinguish contracts from the rest of the code. The disadvantage of running after the typer is that all the succeeding phases expect typing information to be existent and valid, which means that all new trees that are synthesized during contracts rewriting have to be fully typed to ensure that the rest of the compilation proceeds successfully. And as Scala has a very rich type system the latter task becomes quite complicated and must be performed with care.
Inside of the contracts plugin, every component in the diagram, as already mentioned above, corresponds to a specific rewriting type and performs its dedicated task, where the input is always an AST of the currently compiled program, and the output is a possibly modified AST of the same program, which is then passed on to the succeeding compilation phases.

The first component (if enabled) is the contracts-only output. It comes first because we need to output contracts as they were written by the developer before rewriting them, otherwise, the AST of the program will be modified and contracts-only output will not look the way they were originally written. This component does not modify the main AST; its main task is to output a separate contracts-only file of the currently compiled program. As it does not alter the main AST, the contracts-only output component is a traverser that extends the base traverser in the Scala compiler.

The second component is the main rewriting to enable contract assertions in the compiled binary file. It takes the AST of the current program, modifies it to correctly arrange contracts and returns this modified AST to be passed to the next compilation phase. It is a transformer component as it modifies the main AST, and it extends Scala’s base transformer.

The last component is responsible for injecting contracts-only into the currently compiled program. It takes an AST, possibly with some original contracts already rewritten, compiles contracts-only files on its own and injects these contracts-only specifications at
the call side of some implementation for which those contracts-only are written. Finally, the modified AST is returned to the main compilation process, possibly containing original contracts that are now rewritten as well as contracts-only injected at every usage of the specified implementation. It is also a transformer component because it modifies the AST of the compiled program.

The functionality of the contracts-only components can be enabled and disabled via the following the compiler plugin options:

- enable contracts-only output: `-P:contractsplugin:outputcontractsonly`
- enable contracts-only injection: `-P:contractsplugin:usecontractsonly`

where `-P` flag means it is an option to the compiler plugin and colon symbol separates the flag into parts. First part tells what plugin, in our case it is `contractsplugin`, and the second part is the actual option passed to plugin.

The paths of contracts-only files to be read and written can also be provided via the plugin options:

- contracts-only output path: `-P:contractsplugin:outputpath:<path>`
- contracts-only usage path: `-P:contractsplugin:usagepath:<path>`

where `<path>` should be replaced with an actual path in the file system. Similarly, the path of configuration file for contracts-only injection has to be provided:

- contracts-only usage configuration path: `-P:contractsplugin:usageconfigpath:<path>`

more about it is explained in section 5.3.2.

### 5.1.3 Design and Extensibility

**Design** The code contracts Scala compiler plugin has to perform numerous different tasks and rewritings, as well as be built in a way that anticipates and facilitates future extensions. In this section we will show what is the design of contracts plugin and why it was chosen, finally, how it can be extended.

The class diagram of the contracts compiler plugin is shown in Figure [6]. To simplify the diagram, the contracts-only output and injection parts were omitted.

In Figure [6] we can see the main traits in the compiler plugin that can be reused by mixing them in, as it is already done in between of them. The following is a brief explanation of all the traits:
5 Rewriting

Figure 6: Contracts plugin class diagram. All the lines represent inheritance, all the classes inside the traits represent inner classes.

- **trait** ContractsUtils provides the means to check certain properties relevant to contracts, for instance, is some tree a contracts expression from our library, as well as some other base components that can be extended by new specific rewriters.

- **trait** ContractsTreeGen provides the means to synthesize new trees as well as fully typing them.

- **trait** MethodContracts provides the means to access all contracts of a method by extracting local as well as computing effective pre- and post-conditions. For the latter it uses **trait** ContractsTreeGen to generate new trees of effective contracts.

- **trait** WellDefining provides the means to perform well-definedness checks and mixes-in **trait** ContractsUtils to access contracts relevant information from the trees.

- **trait** QuantifiersTransforming provides the means to transform old expressions within quantifier functions (forall and exists) and mixes-in **trait** ContractsTreeGen to synthesize and type new trees.

- **trait** OldTransforming provides the means to transform old expressions. It mixes-in **trait** QuantifiersTransforming so that the clients would not need to care about
differences in old rewriting. It also uses trait ContractsTreeGen to synthesize and type new trees.

- **trait MethodTransforming** provides the means to transform methods in order to enable contracts runtime assertions. It mixes-in the functionality from numerous other traits to be able to access the method’s contracts, transform old expressions, check well-definedness and synthesize new trees.

An interesting construct in Figure 6 is classes within traits, which is an infrastructure used in the Scala compiler:

- The types of trees (and other compiler specific classes which were explained in Section 4) are path-dependent on the instance of the compiler, meaning that for each compilation process the types of trees are different.

- Most of rewriters are transformers that work with type information, synthesize new trees and then must type them. To do the latter, these transformers have to extend Scala’s TypingTransformer, which resides in trait TypingTransformers and is again path-dependent. This implies that in order to use TypingTransformer, trait TypingTransformers has to be mixed-in, for example, by some trait, and an instance of compiler has to be provided, only then, within this mixing-in trait, custom transformers, which extend TypingTransformer, can be created. This is the reason why every transformer in contracts plugin is inside of a trait, as the latter mixes-in trait TypingTransformers and then the former can extend TypingTransformer.

We can see now that trait composition is a technique used within the compiler, where composable traits, like trait TypingTransformers, have an abstract value val global, which corresponds to the current instance of the compiler. So it means that every class or object which can provide an instance of compiler can mix this trait in, whereas other traits can do that without any obligations.

Once traits, like trait TypingTransformers, are mixed-in, and the variable global (the actual compiler instance) is provided by the mixing-in class, then the functionality within a trait can be used. So such composable traits stand more as the containers of some utilities, and the mixing-in class composes its behavior via these traits rather than just simply being a subtype of them. The 'is-a' relationship of inheritance can now also be seen as 'has-a' relationship because the mixing-in class now has the functionality within the container traits.

Following are the main factors that shaped the design decision of the contracts plugin:

1. The Scala compiler enforces its infrastructure on how the plugin and parts within it have to work
2. The Scala compiler itself already provides many base components, like tree traversers or transformers, that can be extended to perform certain rewritings.

3. Newly created rewritings and other routines in the plugin should be reusable in many places, for instance, functionality like synthesizing trees or accessing contracts of a method.

After considering all these factors, the decision was made to design the plugin in such a way that it would not enforce another infrastructure on top of the compiler’s, but rather give freedom to assemble new concepts by extending base structures from the compiler and reusing existing trait components from the contracts plugin.

This approach is also more future-proof because if another infrastructure within the level of the plugin was introduced, it would enforce a certain amount of rules which would have to be conformed to. In case such infrastructure would become obsolete and would not be sensible anymore, it would still have to be conformed to. In order to avoid this kind of “sacred cow” solution, a design based on trait composition was chosen as shown in Figure 6.

To sum up, in Scala traits are components that can be mixed-in by the different classes, objects and other traits. This ideology of using traits as a composition of functionalities is the basis in the compiler plugin’s design, where components are composed from these different traits.

As a prove of such approach, ContractsTransformer, at the bottom of Figure 6, which performs the main rewriting of contracts, was implemented based on trait composition. We can see that it only mixes-in trait MethodTransforming to transform methods, whereas trait MethodTransforming is composed of many other traits, which perform some more specific tasks. We see how the whole rewriting for methods is in essence assembled from more granular traits, all of which can be later mixed-in by new rewriters or any other components.

**Extensibility** Besides the main contracts transformer, there is contracts-only functionality, which is part of this project as well. Both contracts-only output and injection are separate components, independent of the main contracts transformer. However, many tasks are repeated and this is where the reuse of trait composition comes in handy. Figure 7 displays how contracts-only output and injection components mix-in the other existing traits. To simplify the diagram and emphasize the mix-in composition, internal details of these components are omitted.

We can see that the ContractsOnlyOut object mixes-in trait ContractsUtils and trait MethodContracts, as it needs to recognize contracts relevant information, which has to be output, and these traits perform exactly the necessary tasks. On the other hand, the ContractsOnlyIn component needs to synthesize and inject contracts into an AST, so
it mixes-in `trait MethodContracts` to ask for computed effective contracts and `trait OldTransformer` to transform old expressions in the given contracts.

As the aim of the project is to have a basic working tool chain, the functionality of the contracts-only components is not complete and might be extended. For instance, when contracts-only output is generated by the tool, well-definedness is checked in the original code which had those contract specifications. Nonetheless, contracts-only can also be written from scratch by hand and asked to be injected. In this situation, the well-definedness will not be checked upon creation of contracts-only, and so invalid contracts might be injected. To repair this issue, the `ContractsOnlyIn` component would have to mix-in `trait WellDefining` and then use `WellDefinednessChecker` to possibly validate the contract specifications to be injected.

To illustrate another extension, let’s introduce an object invariant into the contracts compiler plugin. Figure 8 shows this extension in a diagram.

We can see that an object invariant implementation would have to mimic other transformers because of the before mentioned path-dependent types (of the current compiler instance). This way there would be `trait ObjectInvariantTransforming`, which could be mixed-in to allow object invariant rewriting. That is what `ContractsTransformer` would have to do: mix-in `trait ObjectInvariantTransforming` and call `ObjectInvariantTransformer` where appropriate. The latter is a transformer, so it would likely extend Scala’s `TypingTransformer`
and provide the rewriting which is specific to object invariants. For the synthesis of new
trees it could mix-in `trait ContractsTreeGen`, for old rewriting it could mix-in `trait
OldTransformer` and, if necessary, to check validity of invariants it could mix-in `trait
WellDefining`. So we see how the base functionality is extended from certain Scala
compiler constructs like traversers or transformers, while contracts specific tasks are
collected via trait composition from the traits in the contracts compiler plugin.

It is possible that `WellDefinednessChecker` does not perform all the necessary checks for
new contracts, like invariants, which means that it would have to be extended. The
API of `WellDefinednessChecker` provides the means to validate trees, while the rest of
analysis, like recognizing pre- or post-conditions, is performed internally. The specific
checking is performed for contracts, as well as for the rest of the source code, so if
some new rules are introduced for preconditions, then the precondition checker has to
be extended. If some specific validity rules are needed for object invariants, then the
object invariant checker would have to be introduced and called where it is needed.

To sum up, the code contracts Scala compiler plugin can be extended to add new types
of contracts or some other, more independent functionality like contracts-only. In the
latter case, it should be built as another component and added to the plugin, similar to
`ContractsOnlyOut` or `ContractsOnlyIn`. The sequence of components, or when they are
plugged in, is important as some of them must come before others in order to function
properly.
To add new types of contracts, the main contracts component has to be extended. Since all base building blocks for performing tasks on the AST come from the compiler, there is no higher level structure enforced by the plugin, which would have to be extended or somehow attached to. The way to make the contracts plugin extensible is to concentrate on reusable and loosely coupled components so that they could be mixed-in wherever necessary. Moreover, rewriting of new contracts has to be implemented separately on their own, so it is important to be able to reuse existing transformations, for instance old, and concentrate only on new rewriting. So in general, extensibility is reached via trait composition rather than enforcing a certain infrastructure at the plugin level. Especially when plugins already must conform to a compiler structure, which provides the necessary constructs to traverse, transform, generate trees, and much more.

5.2 Contracts

In this section the main rewriting of contracts to enable them in compiled .class file is explained, including rearrangement of a method’s body, rewriting details of effective pre- and post-conditions, and old expressions.

5.2.1 Rearrangement of Method Body

We have seen that the contracts API provides methods for expressing pre- and post-conditions which are supposed to be used within the body of the method for which the contract specifications are devoted. The recommended position of the contract specifications is at the top of the method body, before the actual implementation. To avoid a hard enforcement of the contracts position and provide a certain degree of flexibility, we allow contracts to be written anywhere in the method body and leave it for the rewriter to collect them all and place them where appropriate. Moreover, this approach is chosen to also provide better extensibility, as we will see in the next paragraph.

The first step is to collect contract expressions, which is done by going over the statements in the method’s body and distinguishing which of them are supported contract expressions from the contracts library and which are not. Regarding extensibility, if a new type of contract is introduced, then it would only need to be added to the list of supported contracts and the current mechanism would distinguish it in the method’s body. In contrast, this would not be so easily extensible if contracts were only looked for at the beginning of the method.

At this point we know if the method has contracts or not and if there is no other reason to rewrite the method, for instance, in case object invariants were introduced, we leave it as is, otherwise we continue rewriting.

In the case of preconditions, it is clear that they should be placed at the very beginning of the method. However, there are several problems with postconditions:
• if postconditions are *originally written* at the end of the method, what is then the result of that method?

• if *during rewriting* we put postconditions at the end of the method, then we substitute its original return value with a value of *ensures* clause, which is *Unit*.

Regarding the first point, contracts are allowed to be written anywhere in the method body, but the return value of that method is not changed, i.e. if a postcondition is the last statement, then the method returns *Unit*. This is correct if the method’s intended result is *Unit*, but not otherwise. With this approach, where the result of the method is not manipulated - if it is a contracts expression - we retain more safety in case of extensibility. Let’s say a new type of contracts is added which, for instance, could actually be the return value of some method. As we don’t manipulate the method, it would be correct, but not the other way around, when a special criterion would have to be introduced to handle all kinds of contracts if they are written as the last statement in the method body.

Concerning the second point, we need to put postconditions as the penultimate statement of the method. However, the problem arises that the last statement, which is also the return value, could not only be an identifier but also an expression or method call. And it is possible that only this last statement actually establishes the postcondition. Moreover, it should also be possible to refer to the result of the method within a postcondition. To solve all these problems, we need to take the last statement of the method, assign it to a synthesized local variable, which then would represent the result of the method. Then place the postconditions where our synthesized result variable should be used in the postcondition instead of the *result* expression, and finally use it as the return value of the method.

A rearrangement example of a simple method is shown in Listing 18.

```scala
// original method
def negPow4(x: Double) = {
  requires(x >= 0)
  ensures(result == -math.pow(x, 4))
  val foo = x * x
  foo * -foo
}

// rewritten method
def negPow4(x: Double): Double = {
  // precondition
  requires(x >= 0)
  // method body
  val foo = x * x
  ...
```

44
5  Rewriting

// result of the method
<synthetic> val $result: Double = foo * -foo

// postcondition
ensures($result == -math.pow(x, 4.0))

// return value
$result

Listing 18: Rearrangement of a method

At the top of the listing the original method is shown, and right below, its rearranged
version. We can see that the precondition is placed at the beginning of a method as
the first statement to be executed, because if it fails the method’s requirements are not
fulfilled so it doesn’t compute anything.

Then comes the body of the method, which in this case was only two statements, where
the last one is also the result of the method. As explained above, the result is assigned
to a synthesized local variable $result. The <synthetic> flag before the variable is only
available internally for the compiler, and it means that this code was synthesized during
compilation and is not present in the original source code. Moreover, the variable names
with $ signs as separators are dedicated to the names of compiler generated code. For
more about code synthesis see section 4.2.

The penultimate statement of the method is the postcondition, where the result expres-
sion from contracts library is now replaced with the actual result of the method. And
finally the identifier representing the method’s result is returned, this way satisfying
both type system and program correctness.

Importantly, the synthesized result value of the method is necessary even if the method
returns Unit because the last statement could actually establish the postcondition. Even
if the result is Unit, it is still allowed to be referred to in the postcondition and so has
to be rewritten.

The whole transformation of a method is done in a sequence of steps: pre-transformation
of a method, then transformation of all its children trees and finally post-transformation
of a method, before placing it in the AST instead of the original method tree. Following
is the explanation of the actions that are performed in all of these steps.

Pre-transformation of method  First, the method is checked for well-definedness
before any actions are performed. In case of invalid contracts’ usage, errors are raised and
compilation halts. Second, pre- and post-conditions are extracted (more about this in
section 5.2.2) from the whole method, and if they exist then we know that the method has
to be rewritten. Third, we synthesize the method’s result value and create an identifier
which will be used as the return value of the rewritten method. Finally, we rearrange statements in the method’s body by putting preconditions at the beginning, then the actual implementation, and last postconditions with the original result. However, the synthesized result is kept but not inserted into method yet. Moreover, the result of the method has to be synthesized only if the method has postconditions because that’s the only place where it is relevant.

Transformation of method’s children During this step, all other expressions from the contracts library are rewritten, which at the current state of the project is only old and result. The rewriting of old is given to OldTransformer, as shown in Figure 6, which synthesizes variables to save requested old values. More about old rewriting will be explained in section 5.2.3.

The result expression has to be replaced by the actual result of the method, which is why it was synthesized in the pre-transformation step. So whenever a method call result from the contracts API is found among the subtrees of the currently rewritten method, it is entirely substituted by the identifier of the synthesized result value.

Explicit return

Although Scala discourages the usage of the return keyword, it is still available and could be used. In the context of contracts, when the return keyword is found, it means that the method finished its computations and is about to exit, so the postcondition must be checked at this point. And as there might be many return statements, postconditions have to be checked at all of these places.

An important difference is that the actual result value is different for every return statement, which means that we need to synthesize a new local variable for every return tree and use this particular one when replacing the result expressions in the postconditions. When inserting postconditions to be checked before returning, the return node is substituted with a Block node which contains the locally synthesized result value, postcondition nodes and finally the return statement which now returns the identifier of the previously synthesized result value. As postcondition trees originally existed in only one place of the AST but now are used in many places, they must be copied for every such usage to avoid the AST becoming a graph. An example of a rewritten explicit return is shown in Listing 19 where it can be seen that the postcondition is inserted twice and the result value of the method is assigned twice - the unique value for each exit point.

1  // original method
2  def absSqrt(x: Double): Double = {
3      requires(true)
4      ensures(result[Double] * result[Double] == math.abs(x))
5
6      if (x < 0) return math.sqrt(-x)
7  }
Post-transformation of method  In the last step of the method transformation all internal contract expressions have been rewritten, all the necessary information, like synthesized old variables (explained in more detail in section 5.2.3), has been collected, and the remaining part is to perform a final rearrangement: precondition trees come first, then synthesized old variables, then the rest of method body and finally the rewritten result value, postconditions and synthesized result as a return value. One might wonder why pre- and post-conditions were already rearranged in pre-transformation, and the reasons for this are explained in section 5.2.2.

5.2.2 Pre- and Post-conditions

In this section rewriting of pre- and post-conditions is explained. In particular, the focus is concentrated on computation of effective conditions.

We have already seen that pre- and post-conditions are extracted from all of the statements in method body. At this point, it is necessary to be aware that in Scala the right hand side of a method can be either the whole block of statements or just a single one, which could be a pre- or post-condition, or some other expression. So extraction has to handle all these cases.

To avoid confusion as we continue to talk about effective conditions, we call those pre- and post-conditions that were defined locally for a particular method as local contracts.

In order to compute effective conditions for a method, first of all, they have to be collected from all overridden methods. To do that the trees of those overridden methods
need to be accessed, however, in the Scala compiler the class hierarchy information resides in symbols. So the solution would be to access the overridden symbols, and as they represent declarations, look up the symbol table for the corresponding trees. Interestingly, the Scala compiler does not keep a symbol table, at least not one that could be easily accessed from outside, but a snapshot of it for a current compilation unit could be requested.

Having a symbol, trees of overridden methods can be retrieved and their local contracts can be extracted in order to use them in the currently rewritten method. The extraction is performed in the order of the linearization until any - the top class of Scala’s type hierarchy.

At this point, as we work after the typer phase, we already face several problems. Names of all identifiers are already fully resolved, which means that, for instance, every variable access is named Class.this.x instead of simply x. Moreover, symbols have been introduced with their owner hierarchy, which means, for instance, that if a pre- or post-condition in a super class references a parameter of a method, then when we inject this condition in the subclass, the symbol of that parameter will still be set to the one of the superclass, and it will cause the lambdalift phase of the Scala compiler to fail. To solve this problem the symbol of a method’s parameter has to be changed from the one of the super class to the one of the subclass, for which effective contracts are being computed. For better illustration of these problems and to give a thorough understanding of how to solve them, we will go through a set of examples, as the same problems occur in other places, and might as well be relevant when introducing new contracts.

Listing 20 shows the original source code for which we will compute effective pre- and post-conditions.

```java
class Cell {
    var x = 0

    def add(n: Int) = {
        requires(n > 0)
        requires(n < 100)
        ensures(x >= n)
        x += n
        x
    }
}

class SubCell extends Cell {
    var y = 0
}
```

A unit of compilation that has been submitted to the compiler, which typically corresponds to a single file of source code [OC12].
We can see that there is class Cell which has method def add with contract specifications, and the derived class SubCell which has an overriding method override def add, for which effective conditions will have to be computed. It happens that the overriding method has its own local contracts, but even if it would not, inherited contracts would need to be inserted anyway.

Transformation of this Let’s consider we want now to take a postcondition from def add in class Cell and insert it into override def add in class SubCell. The postcondition contains variable x, and so after the typer phase it would look like the following (emphasis only on variable x, the rest is simplified):

\[ \text{ensures}(\text{Cell}.\text{this}.x \geq n) \]

and when inserted into class SubCell it would not be type correct. To solve the problem we need to reference \( x \) from the current class, which in our case is class SubCell. This means, that before inserting contracts from overridden methods, all references via this of superclasses have to changed into this of the current class. The result in our example would be:

\[ \text{ensures}(\text{SubCell}.\text{this}.x \geq n) \]

It is important, nonetheless, to distinguish this references of some other classes, that are not inherited, like, for instance, immutable.this.Nil, as they don’t have to be changed.

Parameter symbol resolution To illustrate this issue, let’s consider we want to take a precondition of def add in class Cell and insert it into overriding method of class SubCell. In this case the result would look like the following:

\[ \text{ensures}(n > 0) \]
where everything seems fine regarding the trees. However, as this precondition is taken from `def add`, the symbol of the method’s parameter `n` is that of `def add` in `class Cell`. And if we simply insert this precondition into the overriding method, then the symbol of parameter `n` from `class Cell` will, of course, be not reachable from the `override def add` in `class SubCell`.

In order to solve this problem we need to change the symbols of the method’s parameters that are used in contracts of overridden methods, into the corresponding symbols of parameters of the method for which effective conditions are currently being computed.

**Effective contracts computation** Having solved the problems of injecting contract trees from overridden methods, the remaining part is to perform the actual computation of effective pre- and post-conditions.

First, there might be several `requires` and `ensures` clauses for a single method, which semantically means that all of conditions provided in these expressions must hold. And as explained in section 2 if a precondition holds and if the body terminates then a postcondition must hold for a single method of a certain type, so no matter how many `requires` expressions are written, we need to collect them all as one condition. In order to that, we simply take a conjunction of all local preconditions and a conjunction of all local post-conditions specified for a method and treat them as a single pre- and post-condition respectively.

The old value of the precondition for a method in a certain type has to imply the postcondition of that type. So in order to have this old value, we need to save it at the beginning of the method and then use it to build an implication.

Finally all contracts are collected throughout the linearization chain and computed according to the rules of effective pre- and post-conditions, where the former is the disjunction of current and inherited preconditions, and the latter is the conjunction of current and inherited postcondition implications. The final computed effective contracts for `override def add` in `class SubCell` are shown in Listing 21. In order to simplify the example, full names and other rewritings performed by the `typer` phase are omitted.

```
1  class SubCell extends Cell {
2     var y = 0
3
4     def add(n: Int): Int = {
5         // computed effective precondition
6         <synthetic> val $requires$SubCell: Boolean = n >= 0
7         <synthetic> val $requires$Cell: Boolean = n > 0 && n < 100
8         requires($requires$SubCell || $requires$Cell)
9
10        y += 1
11        x += y + n
12        <synthetic> val $result: Int = x
```
50
In order to compute effective pre- and post-conditions, the trees of overridden methods have to be extracted from the snapshot of the symbol table, contracts need to be transformed to be able to insert them into overriding methods without breaking tree and symbol hierarchies, and finally new trees have to be synthesized and used to conform with the rules of effective contracts. Regarding reuse, all this functionality resides in \texttt{trait MethodContracts}, as shown in Figure 6, and can be mixed-in by any component that needs to access contracts information of a method.

**Position resolution of failed contracts** There is one problem remaining: after all code synthesis and manipulations, the actual conditions are hidden deeply behind the boiler plate expressions. So when some pre- or post-condition fails, the IDE or any other compiler environment cannot pin-point directly to the position of the contract which actually failed.

One solution to enable this functionality would be to perform manipulation of contracts in a more fine grained way, for instance, expressions within all \texttt{requires} clauses could not be assigned to a single precondition variable, so that the position of each of those expressions would be preserved. Having these positions preserved and mapped from actual conditions to synthesized variables, the stack trace could track back to the position in original code where those conditions where written.

Another solution would be to use the originally written preconditions in the \texttt{requires} clause instead of our synthesized variables, which would then be used only in the computation of effective postcondition. The implications of such solution would be the following:

- No special care of position resolution would be necessary anymore, as original code is used and it has the position.
However, all preconditions would be executed twice: once when evaluating synthesized variables and once when evaluating the actual precondition in requires clause. As contracts are meant to be pure expressions, this solution is safe, regarding program correctness, although, performance would be worse, especially as contracts can be any amount of expressions with a Boolean return value.

Having such precise position resolution of failed contract specifications would improve the user experience and the problem in the source code could be tracked much more easily.

5.2.3 Old Expressions

Old expression represents the value the expression held at the pre-state of the method, i.e. using old expressions provided by contracts library one can refer, in a postcondition, to the values that the expressions had before the execution of the method. It is one of the functionalities that could not be provided without compile time rewriting.

To the best of our knowledge, there exist no formal semantics of old in combination with collection operations, such as forall or exists, especially in case of cross-state comparisons. Allow to refer to single values is simple, however, from a usage perspective that is very often insufficient in order to be able to fully express postconditions.

In this section we introduce the rewriting performed by the contracts compiler plugin for simple old expressions, then we extend the model to provide rewriting of old expressions used in quantifiers. After that the design of the contracts plugin for old rewriting is explained and, finally, having understood how the rewriting mechanism works, the intuitive semantics of old expressions are introduced together with relevant implementation challenges.

**Simple expressions** The main idea behind rewriting of old expressions is to take the expression for which the old value is requested, assign it to a synthesized variable at the beginning of the method, and replace the original old expression with the new synthesized variable which represents the value of that expression in method pre-state. An example of such rewriting is shown in Listing 22.

```scala
// original code
class A {
  var x = 0
  def foo(n: Int) {
    requires(n % 2 == 0)
    ensures(x >= old(x))
    x += n / 2
  }
}
```
We can see that requested $old(x)$ is assigned to the synthesized variable $old$x, which is then placed in the postcondition instead of the former. Old variables are inserted after the precondition because they are needed only in the postcondition, which must hold only if the precondition held.

The approach chosen to implement old expressions for Scala is to never perform a deep copy but rather assign only a reference to the actual requested value. For instance, to access a value of some object one could use

```scala
val y = old(foo.bar)
```

then if the current value of some property of the old expression would need to be accessed, one could do it the following way:

```scala
val y = old(foo.bar).x
```

where the rewritten version of such expression would look like this:

```scala
<synthetic> val $old = foo.bar

val y = $old.x
```

We see that only a reference to the requested expression is saved without any transitive copy of objects. This restriction is necessary to simplify rewriting, because in case any
copying would be necessary, a level of confusion would be introduced regarding how deep
the copy should be. And in Scala, as well as many other languages, all objects might
reference each other, so the copy of the whole heap would have to be done, which is
undesirable.

Quantifiers  One of the different types of expressions that could be used in the old
clause, are the elements of collections. The preferred way to iterate over collections in
Scala is to do it functionally by using quantifiers forall and exists, which we consider
to be the most relevant ones in the context of postconditions.

The recognition of when special rewriting of quantifiers has to be performed, is based
on checking whether an expression given to old contains a variable bound by such a
quantifier. Quantifiers in Scala take as a parameter a function, in which case this variable
does not represent a simple value anymore, but rather it represents each element of the
collection. The previously introduced rewriting does not conform to these semantics.

To support quantifiers with old expressions within them, a different rewriting solution
was introduced. To make the explanation easier it will be based on the following post-
condition example:

\[
\text{ensures}(\text{xs}.\text{forall}(x \Rightarrow \text{old}(x.\text{foo}) > 0 \&\& \text{old}(x.\text{foo}.\text{bar}).y < 0))
\]

The postcondition is expressed via quantifier forall, which contains a function

\[
x \Rightarrow \text{old}(x.\text{foo}) > 0 \&\& \text{old}(x.\text{foo}.\text{bar}).y < 0
\]

where parameter \(x\) is function’s local variable, and

\[
\text{old}(x.\text{foo}) > 0 \&\& \text{old}(x.\text{foo}.\text{bar}).y < 0
\]

is function’s body.

Just like rewriting of methods, quantifier rewriting is also performed in two steps: pre-
and post-transformation. In the first step all necessary information is collected, whereas
the actual rewriting is done in the second. The whole algorithm is divided and performed
in the following parts:

1. Traverse the body of the quantifier’s function and collect all old expressions,
which contain the local variable of the function. In our case we collect \(x.\text{foo}\)
and \(x.\text{foo}.\text{bar}\).

2. Synthesize a new list $xs$, which will contain all collected old expressions and will
later replace original list \(xs\)
3. Iterate over elements of original list \(\text{xs}\) and save the values of requested old expressions (in our case \(x\.foo\) and \(x\.foo\.bar\)) into list \(\text{xs}\), synthesized in the previous step

\[
\text{xs}.\text{foreach}(x \rightarrow \text{xs} += \text{List}(x\.foo, x\.foo\.bar))
\]

4. Rewrite the original quantifier to be called on our synthesized list. Preserve the quantifier function’s body but replace every old expression with a corresponding value from synthesized list

\[
\text{xs}\.\text{forall}(x \Rightarrow x(0) > 0 \&\& x(1).y < 0)
\]

Finally, the whole rewritten postcondition would look like this:

\[
\text{ensures}(\text{xs}\.\text{forall}(x \Rightarrow x(0) > 0 \&\& x(1).y < 0))
\]

It will replace the original postcondition, whereas other synthesized trees will be placed at the beginning of the method, so that old values would be saved in the pre-state of method’s execution.

However, simple old expressions can still be used in a quantifier, and they should not be rewritten this way. That’s why it is necessary to collect only those old expressions which contain the quantifier function’s local variable. A full example of quantifier rewriting in a method is shown in Listing 23.
We see that the synthesized code is placed at the beginning of the method. Moreover, special rewriting of quantifiers is performed only for those old expressions which contain the local variable of a quantifier’s function, which in our example is old(x), whereas others, like old(y), are rewritten using the simple approach.

**Rewriting design** The design approach of old rewriters is to make them as reusable components because old needs to be rewritten in the main contracts transformation and also when injecting contracts-only from a separate file, which will be explained in section 5.3.2. Moreover, when introducing new types of contracts, like loop or object invariants, old expressions will have to be rewritten there as well.

The API of the old transformer provides the means of giving a tree with old expressions while returning its rewritten version. It is important to note that during old rewriting new variables are synthesized, and they hold old values of requested expressions at the pre-state of the method. And as any tree can be given for old expressions to be rewritten, the synthesized code cannot just be inserted into the given tree, because, for instance, a postcondition is placed at the end of the method, where synthesized old trees cannot be put, as there they would not represent the values of expressions in the pre-state anymore. So the design choice was made, that the client of old rewriter gives as many trees to rewrite as needed, and only when necessary the synthesized variables can be retrieved, whereas they are accumulated in the meanwhile. This way the client can then insert these synthesized trees where it is appropriate.

Furthermore, old rewriter incorporates the functionality of both simple and quantifiers rewriting, as will be explained in section 5.1.3 in more detail. This way the client does not need to be concerned where and what old rewriting cases might be used. The returned synthesized trees contain all necessary code that needs to be executed at the pre-state of the method.

Regarding extensibility, now only forall and exists quantifiers are supported in contracts. However, conditions can contain any amount of statements as long as the return
value is Boolean, so other collection operations like foreach or find could possibly be used with old expressions as well. If such a need arises, the list of supported operations could simply be extended with new ones, and the current rewriting mechanism should transform them as well.

Limitations and semantics  Rewriting of old expressions in quantifiers is only meant to be a glimpse into the possibilities of old expressions and to show how complicated it is to actually implement some of the intuitive semantics of using old in non-traditional ways. So currently, this functionality is in a very experimental state and has many limitations.

List vs. old list
In the current implementation, the rewriting result of the following quantifiers

\[
\text{ensures}(xs.\text{forall}(x => \text{old}(x) > 0))
\]
\[
\text{ensures}(\text{old}(xs).\text{forall}(x => \text{old}(x) > 0))
\]

is the same because in both cases the values of the elements of xs are collected and saved before the execution of the method. This means that the amount of elements and their order is the same as xs in the pre-state of the method. The currently implemented rewriting for both postconditions is more likely to correspond to the intuitive expectations that developers might have about the second, but not the first. For instance, if any elements of the list change, the first postcondition would be checking the old values of these new elements. However, the paradox here is that some elements are new and possibly did not even exist in the pre-state of the method. On the other hand, it is probable that the new elements existed in the pre-state, and were only put into the list during the execution of the method. In such cases, a copy of the whole heap would be necessary to access these elements in the pre-state.

A graphical representation of such a situation is depicted in Figure 9, where space is divided by the state of the method. The top of the figure shows the pre-state of the method, which is the moment after the precondition was established but before the execution of the method. And final part, at the bottom of the figure, is the post-state of the method, where the postcondition has to be checked. In the pre-state our heap consists of four objects a, b, c, d and list xs, which contains three of those objects, namely a, b, c. During the execution of the method, new object e is created and the elements of list xs are changed. The heap at the post-state of the method consists of five objects a, b, c, d, e and list xs, which now consists of elements d, a, e, whereas b and c are removed.

Let’s analyze the following postcondition

\[
\text{ensures}(xs.\text{forall}(x => \text{old}(x) > 0))
\]
In this case, the semantics of this postcondition would require checking whether $d$, $a$ and $e$ were greater than zero in the pre-state. The implementation problem for runtime verification with element $d$ is that even though the value exists at the pre-state, we cannot obtain a reference to it for old rewriting, and even more, capturing its old value would require copying the whole old heap. Another problem is element $e$, which did not even exist in the pre-state of the method so the semantics of its old value are questionable.

It could be argued, though, that the old value of an object which is created during the execution of a method is the value that the object obtained upon creation. However, the creation of an object might include transitive creation of other objects that it contains, which might even be executed in parallel, so it is hard to determine when an object is fully created. Considering that the object is fully created when all its transitive objects are fully created would mean that its value might not even be established yet in the post-state of a current method, for which the postcondition needs to be evaluated.

So there are both implementation and perception limitations when speaking about referencing old values of new elements of a collection when it comes to runtime verification. The same applies to all objects, not only collections, when the old value of some property of a new object is referred to in a postcondition.

*Size change*
Not only can new elements be introduced, but their amount could also change. And if the size of collection on which the quantifier was used in the postcondition changes in the execution of the method body, then there are problems in both cases:

- If the size of the collection increased then the quantifier, as rewritten with the current implementation, would not refer to the newly added element, no matter its position.
- If the size of the collection shrank then the condition would still be checked for the removed element as well, because elements are saved at the beginning of the method.

**Old list, new elements**

Another case that could be of interest is the following:

\[
\text{ensures}(\text{old}(xs).\forall x (x > 0))
\]

where elements in the post-state, of the list in the pre-state, are referred to. The semantics of such a postcondition would be: take the list in the pre-state, meaning the elements it contains and their order, and check the values of all those elements at the post-state. The current implementation rewrites exactly this way. Being more precise, it should be noted that in this case simple old rewriting is used and not the one for quantifiers because \(\text{old}(xs)\) is not used within a quantifier. Again it is worth noting that no deep copy is ever taken, so in this case when saving a reference to \(\text{old}(xs)\) only the actual list object is saved and not its elements.

**Old and new values of the same element**

It is often necessary to compare old values with new ones for each element of the list, as shown in the example below:

\[
\text{ensures}(xs.\forall x (x > \text{old}(x)))
\]

However, this functionality is not supported by the current implementation. One possible solution would be to zip\(^5\) the synthesized list \(\$xs\) with the original list \(xs\) and then take an appropriate element from the tuple when rewriting the quantifier. Namely, if an old value is requested use the first element of the tuple - from synthesized list \(\$xs\), otherwise use the second - from original list \(xs\).

Nevertheless, this solution also has limitations, for instance, if new elements of the list are in the different order, then comparison semantics are broken. Moreover, if size of the

\(^5\)Zipping takes two input sequences, and produces an output sequence in which every two elements from input sequences at the same position are combined in a tuple.
list changes, then this solution would follow the rules of zip function, which zips until at least one of the given sequences ends, and that means that removed elements would not be checked, which in turn means that solution is semantically correct, as only elements that are left in the list are checked. On the other hand, if list increased, then new elements would not be included in the result of zip function. Semantically, as already discussed above, we face the same paradox that these new elements should be checked, but their old values might not even exist in the pre-state.

\texttt{foo.old(x)}

Such expressions are also currently not supported. Their intuitive semantics would ask for a member \( x \) in the pre-state, of object \( foo \) in the post-state. However, it is again the situation we saw before, where in the implementation of the runtime verification it is not trivial how to reference a value which is not accessible or known at the pre-state of a method, as is the case with \( x \) in this example. Because \( foo \) could be created only during the execution of a method, \( foo.x \) would become a valid reference only then. Although, \( foo.x \) could be set to some other value which existed in the pre-state, but then it would be recommended to use this known value in the postcondition when a need of such expression arises.

Other limitations

- \textit{Nested quantifiers} with old expressions are currently not supported because a quantifier within a quantifier represents a collection within a collection, and more involving rewriting would be necessary. For the scope of this project it was omitted, in order to concentrate on the implementation of the working toolchain of contracts.

- \textit{Different collections} with quantifiers and old expressions within them, are likely to not function properly, as old values are saved in an ordered collection to preserve the sequence of them, where, for instance, iterating over the sets could result in a different sequence at the different points in a program. The current implementation works with lists and possibly with some other ordered collections.

- \textit{Iteration with indexes} are also possible in Scala, although quantifiers are a preferred way. For the scope of this project the emphasis was put on quantifiers rather then loops. The simple rewriting of iteration over collections using loops and indexes would not work because indexes are not known at the pre-state of the method, so for such scenario a special rewriting is also necessary.

As mentioned before, the rewriting of more involving old expressions is only in an experimental state and so has many limitations. However, the idea was to explore this area and see where semantics confront with implementation challenges. As we saw, many improvements are still possible, although not every use case can be implemented for runtime verification to correctly represent the semantics. All these difficulties could be the reason why the other implementations of contracts, like MS Code Contracts \cite{Cor}.  

60
have a very limited support, or none at all, for old expressions used with collections and especially the cross-state comparisons. The development of the formal semantics for such use cases and the runtime verification could be done as a future work.

5.2.4 Workflow

The previous sections explained how every different part of the rewriting contract specifications is performed, and what the results are of the manipulated AST. Here we introduce the high level workflow of contracts rewriting as implemented in the contracts compiler plugin.

The whole functionality related to contracts rewriting is assembled from different traits into the main contracts component, which in essence is a transformer, meaning that it traverses the whole AST and for every tree it returns a modified one if some contract expressions were to be rewritten. The basic workflow of this component is shown as a sequence diagram in Figure 10.

![Figure 10: High level workflow of main contracts transformer](image)

On the left side of the diagram, there is the same Scala compiler structure as we have already seen in Figure 5. On the right side, there is the main contracts rewriting component and the sequence diagram displaying the high level procedure performed by it.

The basis of this component is the transformer, which is divided into several subparts including the method rewriter and the old rewriter, which are tree transformers themselves. This way the granularity is increased and every transformer is responsible for a
Rewriting

certain contracts expression, which in turn gives us better reuse as these granular parts can be utilized in different places. For instance, old rewriter is used by the main contracts component as well as by contracts-only injection component, as will be explained in section 5.3.2. The old rewriter consists also of a quantifier’s rewriter, which transforms old expressions within a quantifier’s local function, as explained in section 5.2.3.

The essential workflow is the following:

1. Traverse the AST and look for methods
2. Give each method to the method rewriter to be rewritten when contracts are specified

Finally, the rewritten AST with correctly arranged contract assertions, where preconditions are placed at the beginning of the method and postconditions at the end is returned to the main compilation process together with other synthesized trees.

It is important to note is that the method transformation works in two main steps: pre-transformation and post-transformation. The former is when traversing the AST we get to the tree of the method for the first time where checks are performed to determine whether or not the method needs to be rewritten and also some initial rewriting. Then all the children of the method tree are transformed, which includes collecting the necessary information and performing other rewritings within the method’s body, such as old expressions. Finally, comes the post-transformation of the method, where all contracts within the method body are already rewritten, all data, like synthesized trees, are collected and the remaining part is to perform the final rewriting, after which our transformed method will be placed in the AST instead of the original one.

To sum up, the main task is performing the manipulation of the AST to recognize contract expressions, extract and arrange them appropriately, synthesize new trees when necessary and rewrite the original code with them.

5.3 Contracts-only

In this section the explanation is given for the second part of the contracts plugin functionality, which includes outputting and injecting contract specifications separately from the implementation of the program. The idea is to be able to extract contracts from source code and output them into a regular .scala file. Then these contracts-only specifications could be edited and distributed on their own, and injected into a program which uses an implementation for which these contracts are specified. The overview of the contracts compiler plugin rewriting tasks, with emphasis on contracts-only, is shown in Figure 14.

The explanation is split into two parts, the first explaining contracts-only output and the second describing their injection.
5.3.1 Output

Contracts-only output functionality resides in a separate component of the contracts compiler plugin, which means it’s workflow is different and it does not depend on the main contracts rewriting component. In fact, it is even plugged-in before the main contracts component because the latter already performs code rewriting, which is not necessary for contracts-only output because we want this output to contain only the originally written source code and not rewritten one.

As it is a separate independent component, its functionality could be easily enabled and disabled when necessary. This is done using compiler plugin options, as well as providing the directory where contracts-only files will have to be output once they are generated. Corresponding plugin options are the following:

- enable/disable: `-P:contractsplugin:outputcontractsonly`
- output path: `-P:contractsplugin:outputpath:<path>`

where `-P` flag means it is an option to the plugin, the second part tells which plugin, in our case it is `contractsplugin`, and the last part is the actual option passed to the plugin.

The requirements for the contracts-only output component is to extract contract specifications from the currently compiled program and output them into another file. In essence, it needs to take the AST of the program, remove all trees from it except those that are relevant for contracts, and then print this transformed AST in the form of source
code. The work has to be done after the _typer_ phase as well as in the main contracts component because, in order to recognize contract trees, they have to be typed.

Although the task seems clear, the implementation is not trivial, especially considering that we have to look one step further: the output of this component will be an input for contracts-only injection. To inject contracts into some program, we need to find the places where the implementation for which contracts-only are provided is used, and to do that we again need typing information. So injection will be performed after the _typer_ as well, which in turn means that in order to be able to inject contracts-only, they have to be compiled at least after the _typer_ phase. This way the requirements are strengthened as output has to be fully type checkable.

Before we dive into implementation details, the implications of these requirements have to be explained in order to understand why a certain approach was chosen.

**Solution Approaches**  The straightforward solution would be to step into the main compilation process after the _typer_, take an AST representation of the program and copy it as we will have to change it. Then transform the AST by leaving only methods with contract specifications but remove their implementations and the rest of the code which is not specified. Finally, print the resulting AST into a file.

However, the first problem encountered with this approach is that even though the Scala compiler performs the printing of trees in the form of source code it is more for debugging purposes rather than user friendly code. As we run after the _typer_, many expressions are already rewritten, name resolution is performed and object names contain the full path with the names of the packages they reside in. Not only is such output undesirable from the usage perspective, as it is very hard to read and understand, especially as we’d like to allow manual editing of contracts-only, but it also does not compile. The trees printed using the default Scala tree printer contain all synthesized trees and their flags are output as well. As we have seen synthesized declarations contain the `<synthetic>` flag which raises a compilation error, complaining that xml could not be parsed. These results illustrate, that the Scala compiler does not have a dedicated pretty printer from an AST to source code, and the only tree printer available is meant more for debugging purposes.

The next step would be to either obtain or implement a pretty printer. As Scala has many different concrete trees representing certain parts of the program and also provides many different syntactical expressions, the implementation of such a printer would need to be aware of every construct and possible expression of the Scala language. Luckily, there is a library for creating automated refactorings for Scala[^10], where refactoring is in essence getting an AST, modifying it and then outputting the refactored code.

This library faces the same problem as the contracts-only output component, and so it has the pretty printer implemented within it. For the scope of this project, support for refactorings is not necessary, so only the part to generate source code was used. It
is a third party library which is still in development and has some bugs, so it might seem risky to depend on it. However, even though it has to be obtained separately, it is in the scala.tools package, which is the same package where the Scala compiler resides. Moreover, it is still a developed and maintained library, and some problems were reported that were found during the implementation of the contracts plugin, and they were fixed shortly thereafter.

The scala refactoring library has two different printers: one is the pretty printer, which attempts to generate source code on its own, and the other is the reusing printer, which tries to match the AST to the original source code and just uses it for printing, instead of any generation. However, both of these printers have some shortcomings. The pretty printer generates source code which is already much more user friendly than Scala’s default printer, however, it still does not perform perfectly and the output does not always pass a type checker. The possible reason could be that Scala has a wide variety of building structures which are still not handled correctly.

The only solution seems to be the reusing printer because it takes the original source code and uses it for output, so no code generation is performed for non-synthesized trees. Moreover, the original code comments, indentations and coding style are also preserved in the output, which, although not trivial to implement, is the naturally expected result of such functionality.

Nevertheless, to enable the reusing printer, trees in the AST representation of the program must have RangePosition rather than any other kind. As was explained in section 4.1.3, RangePosition is assigned only for interactive compilation, and as we are using the AST that was built by the main compilation, the position is of type OffsetPosition, and so the reusing printer cannot take the original code of the trees, because they only have a point location, which corresponds only to single character and not the whole expression.

**Chosen Solution** The reusing printer being the last resort to enable contracts-only output leads to a solution which would use the interactive compiler. Thus, the contracts-only output component, when called during the compilation process, checks what file is currently compiled, and then compiles the same file again on its own using the interactive compiler, which will return an AST with trees having RangePosition. The whole workflow of contracts-only output component is shown in Figure 12.

We can see in the figure, as already mentioned, that the first step is taking the currently compiled unit, which represents a source file, and giving it to the interactive compiler that will return an AST with RangePositions. Before the output, the AST has to be transformed to remove all implementations and retain only contracts relevant information. This task, which will be discussed later in more detail, consists of two parts:

1. Traverse the whole AST and mark those trees that are necessary in the output, for it to type check.
2. Transform the AST by retaining only those trees that were marked during traversal in the first step and removing the rest.

After obtaining this modified AST, we finally give it to the Output part of the component, which uses the reusing printer from Scala refactoring library to generate source code and then writes it to a file in the path which was provided via the plugin option.

**AST transformation** The aim of this part is to transform the AST in such a way that it would contain all information necessary for the output to pass the type checker. Importantly, only declarations of trees are necessary and all implementation within them is always removed. At first sight, the transformed AST would consist of the following definitions:

- methods, which contain contract specifications
- classes containing methods with contracts
- contracts themselves
- methods, variables and everything else that is used in contract expressions.

The variety of necessary definitions increases with every newly added one, which means that there are much more constructs to be retained. As we need to keep all methods and
variables used in contract expressions, it means that classes, objects, traits or any other `Symbol` containing them must also be retained. Moreover, symbols might be subtypes of some other classes or traits that are also necessary for the type checker. Furthermore, symbols might be enclosed within some other definitions, which makes them inner declarations. And in Scala, inner classes can access all variables from their enclosing `Symbol`.

This growing multitude of definitions converges to a solution that would need to keep:

- the symbols of all the trees mentioned in the list above
- their owners and superclasses, which must be retained recursively

The recursion continues until we reach the top owners or base classes or definitions that are not contained in a currently compiled file. In case of the latter, those symbols are imported and in the current implementation, all `import` statements are retained.

The need to retain many symbols for the output to be type checkable is the reason why AST manipulation consists of two steps: traversal, which does all the bookkeeping, and only then transformation, which retains declaration trees based on whether their symbol was marked during the first step.

The example of contracts-only output is shown in Listing 24. We can see that implementations were removed and only declarations, which are necessary to satisfy the type checker, are printed.

```scala
// original code
class Stack[T] {
  var elems: List[T] = Nil

  def push(elem: T) {
    requires(elem != null)
    ensures(elems.contains(elem))
    elems = elem :: elems
  }

  def pop: T = {
    requires(!isEmpty)
    ensures(result[T] == old(elems.head))
    val top = elems.head
    elems = elems.tail
    top
  }
}
@pure
```

67
def isEmpty = elems.isEmpty

// contracts-only output
class Stack[T] {
  var elems: List[T] = Nil

  def push(elem: T){
    requires(elem != null)
    ensures(elems.contains(elem))
  }

  def pop: T = {
    requires(!isEmpty)
    ensures(result[T] == old(elems.head))
    null.asInstanceOf[T]
  }

  @pure
  def isEmpty = null.asInstanceOf[Boolean]
}

Listing 24: Contracts-only output example

After the implementation is removed, the body of the method remains either empty or only with contract specifications. A problem arises at this point when the return type of a method is not Unit, in which case the output will not type check. In order to satisfy the typer, another important necessity must be introduced - the placeholder of the return value with the correct type. For this reason, the expression null.asInstanceOf[T] is used as the return value for every non-Unit method where T is replaced with the actual type of every method.

It would seem nicer to introduce a dummy variable with implicit conversion to any type T, and then use it as the return value instead of the strange looking null.asInstanceOf[T]. This solution would be based on type inference, which would infer the type from such a method’s declaration:

```
def isEmpty: Boolean = false
```

However, in Scala it is possible to declare a method without explicitly stating its return value, like the following:

```
def isEmpty = false
```

in which case, the return value is inferred from its body. And if the body is replaced with some value, for which type also has to be inferred, like
then type inference fails, and the type of the method is set to $T$. As a result, the whole program will not type check correctly.

### Limitations

**Overriding methods without local contracts.**

The output does not contain overriding methods which don’t have local contracts, even if their overridden method did have contracts. This means that a method has effective contracts, although, as it is not present in the output, the contracts for it will not be injected during the contracts-only injection phase.

The solution would be to extend the traversal part of the contracts-only output, where for every method which contains contracts, the symbols of all its overriding methods should also be retained. Then the declarations of these methods will be printed in the contracts-only output.

**Output artifacts**

For source code generation this component uses the Scala refactoring library, and the reusing printer sometimes prints out artifacts like additional unnecessary symbols which cause the type checker to fail. At this point the solution would be to report a bug, as the library is still being improved and maintained. A few problems were already reported and were fixed during this project. In the meantime, to avoid artifacts, in case these artifacts appear, one should attempt to write the code excluding expressions which cause those artifacts.

So we have seen that contracts-only output must use the interactive compiler in order to obtain `RangePosition`, having which the output of contracts-only files can be performed based on original source code, with comments and user coding style preserved. The transformation of contracts-only needs to gather symbols of many trees in order for the output to be type checkable. If more symbols are required to be retained for the output, only the traversal part has to be extended by adding these new symbols, and the rest of the component will perform the rest of the necessary tasks.

As contracts-only are simple `.scala` files which only need to pass the type checker, and not the full compilation process, it is very easy to edit and create them manually using an IDE. As IDEs use the interactive compiler to perform type checking when the user is typing, all errors and mistakes while writing contracts-only files will be shown immediately, without a need to perform the full compilation.
5.3.2 Injection

It was explained how contract specifications are extracted from the program and that they can be distributed independently from implementation. However, to be able to use these specifications and complete the whole circle of contracts-only functionality, the means of injecting them into some program has to be provided. So the final part of this project is to enable injection of generated, or hand written, contracts-only into any program that uses the implementation for which the contracts-only were specified.

As explained in section 5.1.2, contracts-only injection is a separate component in the compiler plugin, just like contracts-only output. Thus its functionality could be easily enabled and disabled using compiler plugin options as again described in section 5.1.2. Moreover, the path from which contracts-only files have to be taken must also be provided via a plugin option.

Another requirement for the contracts-only injection component to function properly is a configuration that describes the mapping of contract specifications to the corresponding implementations. This way, we can also provide more flexibility as the same contracts can be configured to be injected for several implementations. For instance, if users have several implementations of a stack data structure, which is internally implemented differently but have the same interface, then contract specifications for this interface could be the same for both of the implementations.

The configuration is provided using a JSON file, an example of which is shown in Listing 25. As we can see, the mapping of contracts to implementations is done by defining their full package and a class name. The structure of the configuration is the following: string of contracts name mapping to an array of strings with full names of implementations.

```json
{
    "structures$Contracts.Stack": [
        "structures.nocontracts.Stack",
        "structures.nocontracts.StackList",
        "structures.nocontracts.StackArray"
    ],

    "structures$Contracts.CellStack": [
        "structures.nocontracts.CellStack"
    ],

    "structures$Contracts.Account": [
        "structures.nocontracts.Account"
    ]
}
```
With such a configuration, the usage of the component becomes much more flexible and also more controlled. Moreover, using such a configuration, it could be possible to map several contracts to a single implementation if such a need arises, however, this functionality is not fully tested yet. On the other hand, the semantics of the program, which would have several contracts, could become vague and unclear.

Importantly, as Scala is a statically typed language and the contracts plugin works after the typen phase, contracts must type check with the implementations for which they are specified. It means, for instance, that if we have two implementations of stack, where one is internally implemented using List and the other using Array, these data structures are mentioned in contracts, then contracts-only injection will not compile for at least one of implementations. In such a case, the recommended solution would be to surround these expressions in a generic method, which could then be used in contracts and be safely injected for both implementations.

**Requirements** The working scenario of contracts-only injection is different from the main contracts component because in this case the implementation for which contracts are specified could be in many different forms: it could be a file in the current project, the sources of which are accessible, however, it could also be a file in some other project, or even compiled library. In case of the latter two, we might not be able to access the source code anymore, so to provide a flexible solution where contracts-only could be used for an implementation in any format, we need to inject them at the call side, rather than in the actual implementation, of the method. A conceptual solution is shown in Listing 26.

```
val car = new MyCar

// <- inject pre-conditions of go
car.go(x)

// <- inject post-conditions of go
...
```

Listing 26: Contracts-only injected at call side

Notably, the source code for which the contracts-only are specified could have its own contracts, which are meant to be rewritten by the main contracts component. The latter is a transformation based on recognizing contract expressions and rewriting them as explained in section 5.2. For this reason, the contracts-only injection component must perform its rewriting after the main contracts transformation is finished, as shown

71
in Figure 5. Otherwise the injected contracts would be rearranged, which is obviously undesirable because they are specified for a program to be called from the particular position, and not for the program which is executed at the current position.

With such a solution, we enable the possibility to have local contracts, as well as to provide separate contracts-only for the same implementation. Although conceptually it is a questionable design, it is at least technologically possible and supported by the current implementation of the contracts compiler plugin.

**Implementation** Having explained the idea behind contracts-only injection and what the requirements are for this part, we continue with a description of implementation details and challenges.

The workflow of the contracts-only injection component is shown in Figure 13.

![Figure 13: Workflow of contracts-only injection component](image)

On the left side, as we have already seen before, there is a Scala compiler which takes the source code of the program as an input and produces compiled bytecode for it. The contracts plugin comes in after the typer phase. On the right side, the actual workflow of the contracts-only component is displayed. It is, in essence, a transformer which takes an AST of a program, manipulates it and returns to the subsequent compilation phases.

The idea is to traverse the whole AST and look for method calls of the implementations for which contracts-only specifications were provided via a configuration file (shown in Listing 25). Once such method call is found, its tree is transformed by inserting pre- and post-condition trees around it.
In order to know if a particular method call has contracts-only specifications and to be able to inject their trees, at first we have to obtain those trees. As contracts-only files might not be on a project path, they are not included in the main compilation and have to be compiled separately. For this purpose, an interactive compiler is used, which provides the means of compiling source code and getting an AST of a program. The next task is to traverse the obtained AST and collect all methods which have contract specifications. After performing this traversal, we know which methods in the original program have contracts-only specifications that need to be injected.

At this point, however, we have to look one step further, which is the actual injection of contracts-only into the original program. The major issue here is that the original program is compiled using the main compiler, whereas contracts-only are compiled using our own instance of the interactive compiler. The problem is that types in the Scala compiler are path-dependent on the actual instance of the compiler, meaning that types of trees, symbols, positions and all other internal objects are simply different between the AST of original program and the AST of contracts-only specifications.

Consequently, we are not able to inject contracts-only into the original program because the types of trees do not match, and such an implementation will not even compile, raising an error about type mismatch. To the best of our knowledge, there is no other way it could be done because even if contracts-only files were in the main compilation path, the source file, where these contracts-only are needed, might be compiled first. The consequences are that we will not have contracts-only trees available at this point, and so they will not be injected.

One possible way of overcoming the problem of path-dependent types in Scala is simply casting them, as quite often the underlying objects are dynamically the same, although their static type differs. In our situation, nonetheless, it is not the case, as we instantiated another compiler, which is another object, and its internal types like trees and symbols are different statically as well as dynamically.

Up to this point, the contracts compiler plugin was developed against Scala version 2.9, and as the Scala compiler is constantly being developed and improved, a newer version 2.10 is already in the works. However, it is still neither released, nor even in the beta version. But an improvement was made adding macros to the Scala language, which enables metaprogramming [BO12], that caused many changes within the whole compiler. Concerning the contracts compiler plugin, an important new feature was added that enables importing trees, symbols and types from one compiler instance into another, which is exactly what is necessary in our situation.

After migrating to Scala version 2.10 and being able to use tree importers, we are able to inject contracts-only trees into the original program by first importing them from our interactive compiler into the main compiler. The next step is to perform the actual injection of contracts-only by surrounding a method call with these specifications. The graphical representation of such an AST manipulation is shown in Figure 14.
Figure 14: Contracts-only injection in the AST. The AST of the original program is shown on the top left, where the black node represents the tree of a method call for which contracts have to be injected. The AST of the contracts-only specifications is shown on the top right, where the black node represents the tree of the specified method, its left child is a precondition and right child is a postcondition. The AST of the original program with contracts-only injected is shown at the bottom, where the grey node represents the `Block` that replaces the rewritten method call, and which consists of a precondition, followed by the actual method call, and then a postcondition. The subtree of the original method call remains unchanged, this way preserving the semantics of the program.

Now the implementation details of the actual injection will be explained by first providing the general steps. However, some details will still need to be revised, in order to have a complete solution.

Assuming we have imported all contracts-only trees, then traversed the AST of the original program and found a method call for which contracts-only were specified, then the implementation of their injection would consist of the following steps:

1. Select contracts-only for a current method call
2. Compute their effective pre- and post-conditions
3. Copy contract trees from the AST of compiled contracts-only files
4. Rewrite every occurrence of this reference in contract expressions with the identifier of the object on which the currently rewritten method is actually called

5. Rewrite parameters passed to the method, to be able to refer to them in the precondition

6. Rewrite the result of the method call, to be able to use it in the postcondition

7. Rewrite old expressions

8. Arrange the trees for preconditions to come first, then the actual method call and finally postconditions

9. Inject trees into the AST of the main program (like shown in Figure [14])

All these steps are necessary in order to perform a functioning injection of contracts-only, however, a few inadequacies appeared during implementation. The first one is that effective pre- and post-conditions were not computed. This is because we are importing the ASTs of contracts-only specifications from the interactive compiler into the space of the main compilation, where these imported trees are neither attached to the AST of the main program nor even known. For computation of effective contracts, the overridden methods need to be accessed, which is done using the symbol table, but in this case the imported trees are not included in the symbol table, as they were only manually imported. Consequently, it is not possible to inject contracts-only specifications with computed effective pre- and post-conditions.

In order to avoid this deficiency, we need to improve our solution by providing the symbols and trees of overridden methods manually, which has to be done right after the compilation of each contracts-only file where valid symbol table will be available. So when traversing the AST of compiled contracts-only files, we need to not only collect methods with their contracts, but also all their overridden methods as well, because after importing, it will not be possible to obtain them.

This way we improve the step of computation of effective contracts by collecting overridden methods manually and providing them at the time when the actual computation is performed.

Another problem that was faced during contracts-only injection computation was at the actual injection step, where trees from contracts-only ASTs were put into the AST of the main program. This problem consists of two main concerns:

- trees and symbols from the contracts-only AST are unknown to the AST of the main program. Moreover, trees and symbols are built in a certain hierarchy, which has to be preserved or reintroduced upon injection

- there might be many calls of the same method which has contracts-only specifications, and so injection will have to be performed for all of them.
Regarding the first point, symbols must have valid owners, which are introduced by the \textit{typer} when compiling contracts-only files. However, when we take these symbols and inject them into the AST of the main program, their owners are not accessible anymore because they remain in the AST of contracts-only specifications. To solve this issue, the owners of injected symbols have to be changed to some symbols in the AST of the main program. This step has to be performed with care, having in mind that we are rewriting a method call with a block of statements, and \texttt{Block} in the Scala compiler does not have a symbol, as explained in section \ref{sec:limitations}, which means some other symbol has to be used. The current implementation uses the symbol of the enclosing method in which the currently rewritten method call is made. However, this solution is limited and has its drawbacks, which will be discussed in the Limitations part of the current section.

In order to avoid the problems related to the second item, we need to clone all the symbols and perform a deep copy of all the trees upon every injection. Moreover, symbols are created only for declarations and then the same symbol is used for all the identifiers of that declaration. During cloning, the symbol of declaration is changed, but all identifiers of this declaration still reference the old symbol. Which means, that we need to traverse the AST and change the cloned symbols of the relevant identifiers. Moreover, the symbols whose owner was cloned have to be cloned as well, so the whole procedure has to be performed recursively until no symbols with cloned owners remain.

We have seen that in order to be able to inject contracts-only specifications we first need to compile them on our own and collect all methods with specifications, as well as all their overridden methods. Then we must import the ASTs of compiled contracts-only files into the compilation of the main program, which is a facility available in the Scala compiler only since version 2.10. Finally, we perform the actual injection of contracts-only specifications into the main program at the call side of the corresponding implementation for which these contracts-only are provided. This last task requires accuracy when injecting symbols from one AST to another.

For illustration of injected contracts-only, let’s use the stack for which contracts only were output as shown in Listing \ref{lst:stack-contracts}, however, this time we will use the implementation without contracts as shown in Listing \ref{lst:stack}.

```scala
1 class Stack[T] {
2   var elems: List[T] = Nil
3
4   def push(elem: T) {
5     elems = elem :: elems
6   }
7
8   def pop: T = {
9     val top = elems.head
10     elems = elems.tail
11     top
12   }
13 }
```
Rewriting

```scala
@pure
def isEmpty = elems.isEmpty
```

Listing 27: Implementation of stack without contracts

The example of injected contracts-only specifications for the stack without contracts is shown in Listing 28.

```scala
val stack = new Stack
stack.push(0)
val peak = stack.pop
assert(peak == 0)

val stack = new Stack
{  
    <synthetic> val $elem: Int = 0
    requires($elem != null)
    <synthetic> val $result = stack.push($elem)
    ensures(stack.elems.contains($elem))
    $result
  }

val peak = {
    requires(!stack.isEmpty)
    <synthetic> val $old$head = stack.elems.head
    <synthetic> val $result = stack.pop
    ensures($result == $old$head)
    $result
  }

assert(peak == 0)
```

Listing 28: Injection of contracts-only specifications for ...

We can see that every method call which had contracts-only specifications, in our case `push` and `pop`, now has these contracts injected at the call side. A simple method call needs to be replaced with numerous statements, so they are enclosed by a block with a return value being the result of the original method call.

In order to refer to parameters of a method in a precondition, we first need to the precondition and as the parameter to the actual method. As parameters could be any expressions, possibly non-pure, this way we avoid multiple evaluation of them, which is necessary, in order to preserve the program unchanged.
Next, we can see that every variable access or method call which is defined in the class Stack is now accessed through the instance of stack, on which rewritten methods are called. Notably, postconditions might access private variables and methods even when specified for a public method, and as injection is performed at the call side, these private symbols will not be accessible anymore. However, as validation of accesses is done during the typer phase, this problem does not seem to cause any compilation problems in the subsequent phases.

Finally, for the rewritten method call stack.pop, we see synthesized old variables used in the postcondition together with rewritten result value.

**Limitations**

The current implementation performs the basic operations and provides the whole mechanism of injecting contracts-only specifications, however, it still has many limitations, that are due to the focus on the working tool chain rather than support of wide language specification features. The limitations will be discussed in the following paragraphs.

*Contracts-only injected only when within a method*

Currently, contracts-only specifications will be injected only for those method calls, that are made within a body of some method. For method calls made at the top level of a class, trait or object, which in Scala is treated as constructor, or when given as a parameter to a higher-order function, contracts-only specifications will not be injected. This is because of the intricate resolution of the owners of injected symbols, where the current implementation sets the enclosing method as an owner. To eliminate this shortcoming, the more granular analysis of possible owners, of the injected symbols, should be performed. For instance, if enclosing method is not available, then a class, trait or object could possibly be used.

*Methods with multiple parameter lists*

The current implementation supports only method calls with either a single parameter list, for instance,

\[ \text{obj.foo(x)} \]

or no parameter list at all, for instance,

\[ \text{obj.foo} \]

However, a method call with multiple parameter lists, for instance,

\[ \text{obj.foo(x)(y)} \]

or

78
Rewriting

\[ \text{obj.foo}(x,y,z)(a)(b,c) \]

is not supported, and for such a method call contracts-only specifications will not be injected. The reason is that, even though symbols have information for all parameter lists, the trees for such methods are built using multiple \texttt{Apply} trees, while current rewriting is done only for single \texttt{Apply} tree. The solution is to perform the transformation for this particular step recursively. However, at this point testing is more involving than the actual implementation, to be able to claim that this feature is fully supported. Due to time constraints of the project, the support of this feature was omitted.

\emph{Super calls}

In Scala the variable access of a super class via \texttt{super} call is restricted. As shown in the Listing 29, only a method is allowed to be accessed in such a way.

```
1 class A {
2     var x = 0
3 }
4
5 class B extends A{
6     x = 1  // ok
7     super.x = 1 // causes compilation error
8 }
```

Listing 29: Variable access via super call

For this reason, contracts-only cannot be injected for methods called on \texttt{super}. If a variable is referred to in the contract specifications for a method of a superclass, then when injecting contracts-only for a super call in the overriding method, every \texttt{this} reference is replaced with a reference on which the actual method call is made, and if it happens to be \texttt{super}, then the program will not compile. This is because the Scala compiler checks for such accesses in the \texttt{superaccessors} phase, which comes after the contracts plugin. An example of such scenario is shown in the Listing 30.

```
1 class A {
2     var x = 0
3 
4     def foo {
5         requires(x > 0)
6         ...
7     }
8 }
9
10 class B extends A{
11     override def foo {
12         super.foo
13     }
```

Listing 30: Accessing a variable via super call in an overriding method
Now let’s say, for instance, that class A is put into some other library for which contracts-only are then output. Our program then consists only of class B, for which we want to inject the previously output contracts-only. The injection of a precondition, which after the type phase will be

\[ \text{requires} (A.\text{this}.x > 0), \]

should be performed for a method call super.foo. Now the contracts-only rewriter replaces every this reference with the reference on which the method call is made, and in our case it is super. The result after contracts-only rewriting will be as shown in Listing 31.

```scala
class B extends A{
  override def foo {
    requires(super.x > 0)
    super.foo
  }
}
```

Listing 31: Injected contracts-only for super call

The injected precondition accesses a variable in the superclass via super, which, as already explained, is restricted by the Scala compiler.

The current implementation of the contracts-only injection component just skips method calls on super without performing contracts-only injection. In this case, the limitation is not in the contracts plugin, but in the Scala compiler, so we just have to conform to these rules as well.

**Enclosing symbols**

If the contract specifications for a method access variables from enclosing classes, traits or objects, or any other enclosing symbols, then after injection of such contracts-only, the program will not compile anymore. The reason is that contracts-only are injected at the call side, and variables from enclosing symbols, of declaration of a currently rewritten method call, are not accessible from the call side. To avoid such a scenario, the user should wrap the variables of enclosing symbols into local methods (getters) which would then be accessible from the call side. The current implementation of contracts-only injection raises a compilation error if contracts-only contain such accesses from enclosing symbols.
5.4 Well-definedness

The rules of the well defined contracts were explained in section 2.2, here the details of the implementation are provided for each rule respectively:

1. *The visibility of field accesses and method calls used in the precondition.* If this rule is violated a compilation error will be thrown.

2. *The pureness of the contract specifications.* For the scope of this project, the \texttt{@pure} annotation for methods is introduced. Only those methods that are annotated as pure are allowed to be used in pre- and post-conditions. If this rule is violated a compilation warning will be thrown. In this case it is only a warning because pureness is not actually checked, so we can not claim that it definitely is not pure. Moreover, in Scala every operation is a method call, which means that all arithmetic, assignment and logical operations are actually methods that, of course, don’t have \texttt{@pure} annotations. All these actions (and many more) will trigger compilation warnings when they are used in contracts. To avoid this problem and suppress these kind of warnings, a list of methods which are known to be pure is introduced. Now both methods annotated to be pure and those known to be pure are allowed to be used in contracts.

3. *old and result expressions can only be used in postconditions.* Failing to do so and using \texttt{old} or \texttt{result} in the precondition or elsewhere in the method body will trigger a compilation error telling that \texttt{old} and \texttt{result} can only be used in the postcondition of the method.

4. *Usage of old expression in a parameter of another old expression is disallowed.* Otherwise, a compilation error will be raised. Usage of \texttt{result} in \texttt{old} expression’s argument is disallowed and will raise a compilation error as well, because \texttt{result} represents the product of a method which is not existent in the old space.

5. *Usage of result in old expression’s argument is disallowed* and will raise compilation error as well.
6 Limitations

In this section, the limitations of contracts compiler plugin, the most of which were explained throughout this paper during the discussion of a certain functionality, are collected here into one place for a better reference.

1. **Contracts for abstract methods.** The current implementation of the contracts compiler plugin does not support contracts in combination with the abstract methods, to be more precise, as contracts are specified in the body of the method, the latter can no longer be abstract, because it has a body. This way contracts will be evaluated and inherited by the subtypes, however, as soon as there are some contracts specified for a method, it will no longer be enforced to be implemented by the subtypes of the current type.

   The possible solutions of how to overcome such a limitation are the following:

   - Introduce an annotation to provide the means to annotate the methods to be abstract. Then contracts could be specified the regular way and the methods would be rewritten during the compile time to enforce the subtypes to provide implementations.

     One way of implementing such a solution could possibly be done by plugging-in after the `parser` phase to remove contracts from the abstract methods and put them in some other synthesized auxiliary methods, then the `typer` phase would correctly type the program as meant by the user, as well as typing the contract expressions. Finally, after the `typer` phase, we could perform the typical rewriting for contracts, where those in the synthesized auxiliary methods would be treated as contracts for the corresponding abstract methods.

     This way, there is not much additional code to write for the user and the whole arrangement is done by the compiler plugin, which is very often how things are approached by the Scala compiler itself.

   - Another way this problem could be solved is to let the users write those auxiliary methods, which would then be used by the compiler plugin during rewriting. The similar approach is implemented in Ms Code Contracts [Cor] library. Such a solution emphasizes the separation of contracts from the corresponding implementation, and does not alter the original program if the compiler plugin is not used. However, much of the boilerplate code that was handled by the compiler plugin in the previous solution will now have to be written by the users.

2. **Position resolution of failed conditions.** If a pre- or post-condition does not hold, the current implementation does not provide a position for the stack trace which would map directly to the failed condition. More explanations and the possible solutions are given in section 5.2.2.
3. *Old expressions.* The possibilities of old expressions, especially when used in the quantifiers and cross-state comparisons, are limited. A comprehensive explanation and analysis of these limitations are given in section [5.2.3](#) whereas here we only list them for a better reference. Not supported features:

- access of the elements in the pre-state, in a list in the post-state, of the method, in the form of
  
  \[
  \text{ensures}(xs.\text{forall}(x \Rightarrow \text{old}(x) > 0))
  \]

- comparison of the old and new values of the elements in the list, in the form of
  
  \[
  \text{ensures}(\text{old}(xs).\text{forall}(x \Rightarrow \text{old}(x) > x))
  \]

- size change of the collection
- iterating over the collections in loops and accessing the elements using indexes
- only the lists and possibly some other ordered collections are supported when using the old expressions in quantifiers
- nested quantifiers with the old expressions

4. *Contracts-only output* limitations, as explained in more detail with the solution suggestions in section [5.3.1](#) are the following:

- the overridden methods of the method with the contract specifications are not output
- the source code generation is done using a third party library (which is in the same package as the Scala compiler, but has to be obtained separately), that is still under the development, so some Scala expressions might be printed with the unnecessary artifacts
- the contracts-only files, that are written by hand, might be not checked for well-definedness

5. *Contracts-only injection* limitations with the possible solutions were explained in detail in section [5.3.2](#) The list of them is the following:

- the contracts-only are injected only if a method call, for which these contracts have to be injected, is made within a body of some method and not at the top level of the class, object or trait.
- the injection is not supported of the contracts-only for the methods with multiple parameter lists, in the form
  
  \[
  \text{obj.foo}(a,b)(c)(d,e,f)
  \]

6. As the aim of this project is to build the working tool chain of components rather than supporting wide specification features of the Scala language, only the subset
of those features were chosen to support, namely classes, methods, inheritance and recursion. Although traits and higher-order functions were excluded and not targeted, the current implementation does support inheritance via traits as well as contracts for the declarations of the methods that take functions as parameters. The support for the rest of the Scala language features is not assured.
7 Conclusion

The code contracts library for the Scala programming language was successfully implemented, and provides the means to specify pre- and post-conditions for the methods that are rewritten during compile time into the runtime assertion checks. To enforce the rules of behavioral subtyping, the pre- and post-conditions are inherited from the supertypes and the resolution of contract implications between the types is approached by computing the effective pre- and post-conditions. Additionally, the means to refer to the result value of the method and the old values of the expressions are provided. The latter can also be used with the quantifiers, but as the formal semantics of this functionality are not yet defined, and there are numerous implementation challenges to represent at least the intuitive semantics, the support of the quantifiers with the old expressions is still limited.

Apart from a typical approach, where contracts are written together with the implementation, our solution also enables the extraction of contracts into a separate .scala files. It can then be edited and distributed, and finally, these contracts can be injected during compile time into any Scala program which uses the corresponding implementation. The form of the latter is not relevant, it could be source files as well as compiled libraries, as contracts-only are injected at the call side.

The contracts library API provides the methods that are mostly just stubs, which represent the semantics of what they are, however, the whole empowering is done in the compiler plugin, which performs all the rewriting of contracts at compile time. Both of the parts are implemented with extensibility in mind, to enable the extension of the existing functionalities and more importantly, the addition of the new types of contracts.

7.1 Related Work

Our solution of providing code contracts for a language - which does not support them natively - using a library approach is similar to other works like JML [LBR98], which provides support of contracts for the Java language using annotation comments, Microsoft has built Code Contracts library [Cor] which enables the developers to write the specifications on the .NET platform using the regular library calls. There are also languages that support code contracts natively, like Spec# [BLS04] or Eiffel [Mey92b].

As regards Scala, our target language, it has no native support for contracts, however, it provides the means to express the runtime assertions in several interesting ways as explained in “Contracts for Scala” [Ode10] and discussed more in section 3.4 and appendix A.
7.2 Future Work

All the limitations enumerated in section 6 are considered future work, especially the implementations for the limitations of both the contracts-only output, as discussed more in section 5.3.1, and the injection, as discussed in section 5.3.2, because these are just implementation and not conceptual issues.

Scala has very rich type system and is both object-oriented and functional programming language, so the support for wider range of specification features should be implemented in order to enable the users to specify contracts for this huge variety of language constructs. For instance, adding the support for exceptions by proving the means to specify exceptional postconditions, or full support of contracts for higher-order functions and function literals.

The development of the formal semantics for usage of old expressions with quantifiers and especially cross-state comparisons with respect to the runtime verification, is also considered to be a future work. This subject was discussed in more detail in section 5.2.3.

New types of contracts are probably one of the most interesting future works. For instance, addition of object invariant, which is an indispensable part in order to fully support the behavioral subtyping. Loop invariant, which is a challenging task, because loops in Scala are very expressive, moreover, iteration over collections can be done functionally, where loop invariants might be of interest as well.

Regarding grand-scale efforts, the building of static analysis tools that would reason about the programs based on the formal specifications and that would also provide the compile time results about program correctness, as well as checking the pureness of the conditions used in the contracts. Moreover, implementing test generation for the programs based on the provided specifications.
A Contract Library API Approaches

Other possible solutions for contracts library API are shown in this appendix.

Scala’s `Predef` package provides `require` method, which represents the precondition of the method and is placed inside of the method, unlike the `ensuring`, representing the postcondition, which is placed outside of the method. The first approach was to be more consistent and provide both pre- and post-conditions to be expressed outside of the method. Usage example of such a solution is displayed in Listing 32.

```scala
object Test {
  var n = 0
  var m = 0

  def foo(x: Int) = requires(x > 0) {
    n = x * 10
    m += x
    n + m
  } ensures {result => // a function from result
    result >= 10 // of the method to boolean condition
  }
}
```

Listing 32: Usage of a library which feels like a native language support

We see that `requires` clause comes before the method body, just like `ensures`. The implementation of such a library solution is shown in Listing 33.

```scala
object requires {
  def apply[T](condition: Boolean)(body: => T) = {
    if (!condition) {
      throw new PreconditionException
    } else body // execute body only if precondition holds
  }
}

class Ensures[T](val result: T) {
  def ensures(condition: T => Boolean): T = {
    if (!condition(result))
      throw new PostconditionError
    result
  }
}

implicit def any2Ensures[T](result: T): Ensures[T] =
  new Ensures(result)
```

87
A Contract Library API Approaches

Listing 33: Implementation of a library which feels like native language support

This way we have a precondition before the body of the method and a postcondition after it. Contracts are placed where they are actually checked, namely the precondition is checked before the execution of the method, and the postcondition after that. Similar syntax is also used in Eiffel [Mey92]. With this solution no compile time would be necessary for contracts to be executed correctly, however, old-expressions and inheritance of contracts are not supported.

Using solely a library solution, we can still enable a postcondition to refer to the result of the method. This solution for postconditions is the same as the one which currently exists in Scala’s Predef package, that is by default imported into every Scala source file. However, there are numerous drawbacks to this approach:

- Firstly, there is only one `requires` and `ensures` clause per method, which means that several pre- or post-conditions must be conjuncted inside these calls, which makes it harder to separate contracts from each other and enable or disable them.

- Secondly, contract specifications are considered to be part of the interface, which means part of the declaration of the method, so the preferred way would be to have both pre- and post-conditions written at the beginning of the method.

The usage example of a solution targeting both of these problems is shown in Listing 34, while the implementation of such API is shown in Listing 35.

```
object Test {
  var n = 0
  var m = 0

  def foo(x: Int) =
    requires(x > 0).
    requires(x < 10).
    ensures(n >= 10).
    ensures(n < 100) {
      n = x * 10
      m += x
      n + m
    }
}
```

Listing 34: Both pre- and post-conditions before the method body
The main difference between Listing 32 and Listing 34 is that the latter is not anymore a pure library solution because postconditions are now placed at the beginning of the method, where they will be executed unless rewritten at compile time, which is anyway necessary to enable old-expressions and inheritance of contracts.

Concerning style, now “.” is required after every requires or ensures call which is followed by another contract expression.

Using both of the approaches shown above, everything declared in method body is not accessible by contracts, as they are not within the block of the method, so the rewriting of old expressions and effective pre- and post-conditions becomes much more complicated.
References


References
