Encoding Scala Programs for the Boogie Verifier

Valentin Wüstholz

Master's Project Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

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

October 1, 2009

Supervised by:
Dr. Yannis Kassios
Prof. Dr. Peter Müller

Chair of Programming Methodology
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Proving the correctness of programs has been a major field of computer science research for the last decades. Over the years, many approaches have emerged that try to solve this problem. One of the main approaches is static verification of programs, which is particularly difficult for object-oriented programs.

Since object-oriented programs are usually more complicated to verify, we designed an encoding for the object-oriented programming language Scala in the imperative programming language BoogiePL, which can be statically verified by the existing Boogie verifier. One reason for choosing Scala is that it supports a lot of interesting and helpful features, that are challenging to verify and are not supported in programming languages that can already be verified by existing static verifiers.

We developed a tool that can translate a usable subset of Scala to BoogiePL. Our main focus was on coming up with an encoding for two of Scala's most interesting features, namely closures and traits.

Acknowledgments

I would like to thank Prof. Dr. Peter Müller for giving me the opportunity to participate in this exciting research project. I am very grateful to my supervisors, Prof. Dr. Peter Müller and Dr. Yannis Kassios, for their great support, the inspiring and interesting discussions and their valuable feedback.

I would also like to thank all members of Prof. Dr. Peter Müller's research group. In particular, Joseph Ruskiewicz for helping me to better understand some of BoogiePL's more advanced features and inner workings, and Dr. Pietro Ferrara for the instructive and valuable discussions of the Scala compiler architecture.

Contents

1 Introduction **11**

1.1 Overview of the Report 12

1.2 Motivation 12

1.3 Scala 13

1.3.1 What is Scala? 13

1.3.2 Interesting Features 14

Type system/Type inference 14

Closures and Higher-Order Functions 15

Pattern matching/Case classes 16

Traits/Mixins 17

Singleton/Companion objects 19

Concurrency 19

1.3.3 Challenges 20

1.4 Boogie 20

1.4.1 What is Boogie? 20

1.4.2 What is BoogiePL? 20

Types 21

Constants 21

Functions 22

Axioms 22

Variables 22

Procedures 22

Implementations 23

2 Design **25**

2.1 Overall Design 25

2.1.1 Supported Scala Features 25

Object-Orientation . . . 25
Expressions . . . 26
Types . . . 26
Other features . . . 26
2.2 Detailed Design . . . 27
2.2.1 Heap Model & Type Hierarchy (Prelude) . . . 27
Top types . . . 27
Bottom types . . . 29
ScalaObject . . . 29
Value types . . . 30
Heap and Fields . . . 31
Arrays . . . 32
Difficulties with Boogie . . . 34
2.2.2 Translation to BoogiePL . . . 34
Classes, Traits and Singleton Objects . . . 34
Fields . . . 36
Constructors . . . 37
Methods . . . 39
Method/Constructor Bodies . . . 40
Closures . . . 41

3 Implementation 43
3.1 Scala Compiler Architecture . . . 43
3.2 BoogiePL AST Library . . . 44
3.3 Code Generation Plugin . . . 45
3.4 Difficulties . . . 46
3.5 Case Studies / Example Programs . . . 47
3.5.1 Classes with Methods and Fields . . . 47
3.5.2 Objects with methods and fields . . . 49
3.5.3 Method Bodies . . . 49
3.5.4 Arrays . . . 51
3.5.5 Closures . . . 52
3.5.6 Traits . . . 53

4 Conclusions 57
4.1 Related Work . . . 57

4.2 Future Work . . . 57
4.3 Conclusions . . . 58

Bibliography 67

A Minor Problems with Boogie 69
A.1 Boogie crashes for nullary functions . . . 69
A.2 `free modifies` clause results in a syntax error . . . 69
A.3 Final types are no longer supported . . . 70

Chapter 1

Introduction

These days, program verification and formal methods are still far from being an industry standard. However, program verification is mainly used in a very small number of projects, where safety is absolutely critical. This is somewhat surprising considering that it has been an active area of research for well over 40 years. The biggest obstacle in making formal methods more popular seems to be the amount of time and skills it demands from the programmer as well as the growing complexity of today's systems.

The Spec# programming language [1, 2] was an important step towards making formal methods more accessible for a wide range of programmers. Together with the Boogie verifier [3, 4], Spec# was developed at Microsoft Research and extends C# with contracts that can be statically verified.

Scala [5, 6] is a object-oriented programming language originally developed by Martin Odersky and his group at Ecole Polytechnique Fédérale de Lausanne (EPFL). It provides a lot of interesting features; many of which are intended to make it easier for programmers to express complex ideas with less code. Usually shorter programs can be verified more easily by a programmer if the verifier is powerful enough. We would expect that a program verifier for Scala will boost our ability to verify more sophisticated programs in less time. This would certainly encourage more programmers to give formal methods a try.

For these reasons it is necessary to design and implement a tool that is able to encode a high-level programming language, such as Scala, in a way that a verifier can understand. BoogiePL is an imperative intermediate language that can be verified by the Boogie verifier. If we can encode Scala in BoogiePL, we are one step closer to having a complete program verifier for Scala. Creating a tool that does this translation is the main goal of this project.

1.1 Overview of the Report

This introduction provides a broad overview of this project and the overall structure of this report. We will then present the motivation behind this project and illustrate how it is different from other similar projects such as Spec#.

Since we do not assume any prior knowledge of Scala, the rest of this chapter covers the basic features and constructs of the Scala programming language. For the same reason, we will also introduce the basics of Boogie and BoogiePL. Readers already familiar with either Scala or Boogie/BoogiePL should be able to safely skip the corresponding sections.

We are not able to cover more than just the technologies and methodologies most closely related to the overall topic. Therefore, we assume some fundamental technical knowledge that a Master's student should have. This includes for example experience in object-oriented programming and programming in general.

Chapter 2 summarizes the design process of the tool. In particular, we introduce the subset of Scala that we worked on and outline the design decisions we made and the rationale behind them. This chapter deals mostly with the encoding of Scala's features in BoogiePL on a purely conceptual level.

The actual implementation of the translation tool will be explained in Chapter 3. In order to demonstrate the practicality of our approach, this chapter features a selection of examples that show how working Scala programs are encoded in BoogiePL. It also includes a section about our experience with the tool. For further, even more detailed documentation of the implementation we encourage the reader to have a look at the actual source code.

Finally, Chapter 4 sums up what we did, draws some conclusions and mentions related and future work.

1.2 Motivation

This project should be seen as a first step in the direction of creating a verifier for Scala programs. There are many reasons for starting a project like this; some of which will be pointed out in this section.

In formal methods, there is still plenty of room for improvement, for new approaches and developments that will allow us to verify software in easier and less tedious ways. We already have some very good tools at our disposal that can deal with systems that are not too complex. Spec# is just one of them. Software complexity can have many causes; concurrency, legacy code and non-modular architectures just being some of them. Unfortunately, a lot of software is way too complex to be verified in a reasonable amount of time by programmers that are not verification experts.

One aspect, that is particularly interesting to look at, is the specification effort needed to verify some piece of software. For some approaches, the size of the specification can even dominate the size of the actual software. The goal should therefore be to design a specification methodology that allows us to verify software with minimal specification overhead.

By basically starting from scratch, instead of working on an existing verifier, we are able to experiment more freely with new ways of specifying the desired behavior of programs. On the other hand, by relying on the existing Boogie verifier, we can still profit from the years of development that made it one of the leading verifiers in industrial use.

Another important reason is that Scala offers a lot of interesting language features that we want to explore. This combination of features is quite unique to Scala and they are not yet supported by other languages, for which there exist mature and usable verifiers. We believe it to be easier and more reasonable to build a new verification tool targeted to Scala instead of, for instance, adding some of Scala's features to Spec#.

In the future, we hope to be able to prove interesting programs by exploiting some of Scala's features. In the end, this should make static verification even more accessible and attractive to ordinary programmers.

We also hope that some of our findings (e.g. concerning closures), could potentially be used to further improve other existing verifiers.

1.3 Scala

1.3.1 What is Scala?

Scala [5, 6] is a multi-paradigm programming language, started in 2001 at Ecole Polytechnique Fédérale de Lausanne (EPFL) by Martin Odersky. One of Odersky's main goals was to design a programming language that combines the respective strengths of (purely) object-oriented and functional programming. Some notable features are its strong static type system, type inference and support for generics, higher-order functions, pattern matching and traits. Its main implementation also provides seamless integration with the Java programming language by targeting the Java Virtual Machine (JVM).

Although Scala's object-oriented features are similar to Java's in a lot of ways, there are a couple of differences worth mentioning. Firstly, there is no strict distinction between primitive types (int, float, char, bool, etc.) and reference types in Scala. Everything is an object and there are no built-in operators such as '+' or '&&'. Every operation in Scala is just a method call.

Secondly, Scala supports mixin-based inheritance [7] instead of Java's single-inheritance with interfaces. While in Java the object hierarchy consists of classes, abstract classes and interfaces, Scala further generalizes Java's interfaces. The resulting construct is called a trait and can be mixed into other classes or traits. Traits are more general than interfaces because they are allowed to inherit not only method declarations but also method implementations. This gives the programmer some of the benefits of multiple inheritance without most of its drawbacks. Section 1.3.2 will go more into detail on this interesting feature.

Another interesting aspect is that Scala lets the programmer use a lot of features in a very natural and elegant way, that are popular in the functional programming community. In particular, it supports higher-order functions and even full-blown closures. But it does not force programmers to program in a (purely) functional way. Instead, it lets them decide on which style of programming — purely functional, functional, object-oriented or even a mix out of these — is most appropriate for the particular problem at hand. Section 1.3.2 covers Scala's functional programming aspects in more detail and also presents a few simple examples.

In the following two sections, we present a few of Scala's most interesting features in more detail and mention some of the challenges that were most difficult to overcome. This is not meant to be a complete tutorial on Scala and we do not intend to cover all of these features in our implementation.

For a more complete introduction to Scala, I highly recommend the book by Martin Odersky, Lex Spoon and Bill Venners [8]. As a freely available alternative, I also recommend "Scala By Example" [9] and "A Scala Tutorial for Java programmers" [10].

The examples are deliberately kept quite simple, so that a Java or C# programmer should be able to understand them without a lot of explanations. Features that are not available in these languages or syntax that is not easily understandable are explained the first time they are used.

1.3.2 Interesting Features

Type system/Type inference

The first interesting feature to look at is Scala's type system. Although it tries to stay compatible with Java, it introduces a few novelties. The most obvious change is that Scala's type system offers type inference. This allows the programmer to leave out type declarations if they can be inferred automatically by the type checker. This often makes programs easier to read and less tedious to write. Here is a simple example that demonstrates this:

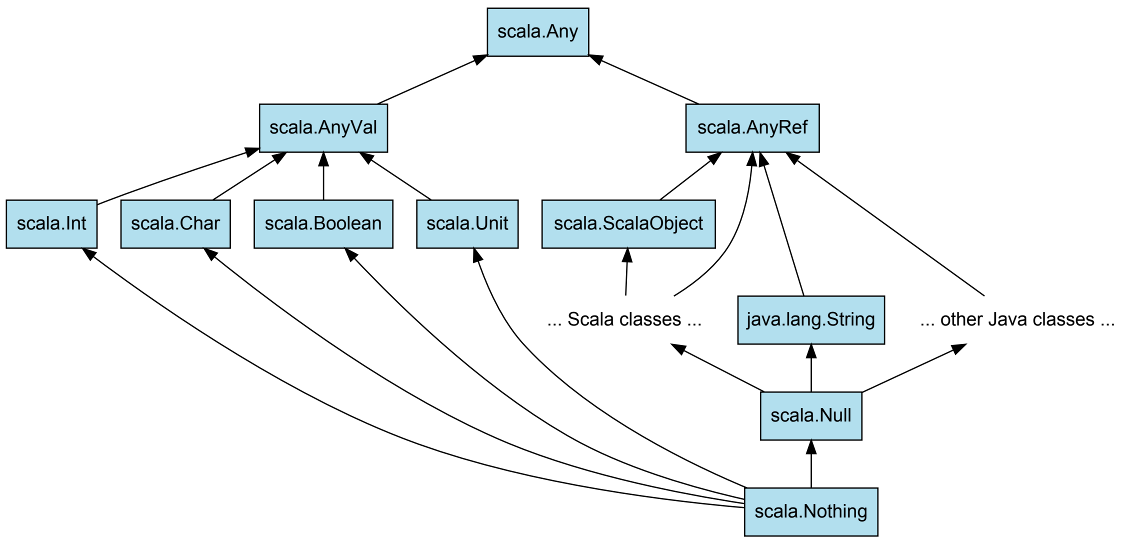
```scala
val i = 3

val a = Array("array", "of", "strings")
```

LISTING 1.1: **Type inference**

In both cases the type checker can figure out that the type of `i` and `a` is `Int` and `Array[String]` respectively. This might not seem too hard in this case, but it also works in more complicated ones.

A second noteworthy change that Scala made to Java's type system, is that it restructured the type hierarchy.

FIGURE 1.1: **Scala's type hierarchy**



As can be seen in Figure 1.1, there exists a single type `scala.Any` at the top of the type hierarchy. But `scala.Any` is not like `java.lang.Object` in Java, because the type is abstract and thus cannot be instantiated. The equivalent of `java.lang.Object` in Java is `scala.AnyRef` instead, which is the supertype of all reference types. Similarly, `scala.AnyVal` is the supertype of all value types, such as `Int`, `Boolean` or `Char`.

In addition to being subtypes of `scala.AnyRef`, Scala objects are also subtypes of the `scala.ScalaObject` trait. This allows us to distinguish Scala types from Java types.

A last important difference is that in Scala there exists a bottom type, `scala.Nothing`, which is a subtype of every other type. Similarly, `scala.Null` is a subtype of every reference type. The `null` reference itself is the only instance of that type and can be assigned to any reference as a consequence.

Interestingly, `scala.Nothing` is also used to type expressions that might throw exceptions. Scala's type system is quite elaborate and offers many more advanced features that we cannot cover in detail at this point, such as variance annotations, existential types or type constructor polymorphism.

Closures and Higher-Order Functions

Closures are one of Scala's features that originally emerged from functional programming. Scheme was the first programming language that made them popular and since then many other programming languages have adapted them. Closures are functions that may refer to variables defined in their lexical environment. They are said to "capture" their environment.

In Scala functions are first-class citizens. This means, that they are treated just like other objects, such as integers or strings. They can be stored in variables, can be passed as function arguments or can be returned by other functions. Functions that accept other functions as their arguments or return them are known as higher-order functions.

But a language that supports higher-order functions does not automatically support closures. A good example for a language like this is C since it supports function pointers. Java does not even support higher-order functions natively.

Example 1.2 demonstrates that functions are first-class citizens in Scala by implementing a simple `map` function that applies an arbitrary function to every element of a list. (Scala already offers a built-in `map` function. But it is still interesting to look at how it might be implemented.)

```scala
def map[A,B](list: List[A], fn: A => B) : List[B] = {
  if (list == Nil)
    Nil
  else
    fn(list.head) :: map(list.tail, fn)
}

def test() {
  val l = List(1, 2, 3, 4, 5)
  println(map(l, (a: Int) => a + 1))
}
```

Listing 1.2: `map`

The type signature of `map` tells us that this function takes a list of elements of type `A` and applies a function to each of them. The resulting list is of type `B`. `A` and `B` are the function's generic parameters and this general concept is called parametric polymorphism. The `map` function provides a good example of a higher-order function because it takes another function as its argument.

If the argument `list` is the empty list (`Nil`) the `map` function simply returns `Nil`. Otherwise it applies the function to the first element (`head`) of the list, calls itself recursively on the tail of the list and appends ("conses") the first intermediate result to the second intermediate result to create the final result list. Note that if there is no explicit `return` statement the function always returns the last expression (the if-then-else expression in our example).

The `test` function finally shows how our map function can be used to increment every item in a list of integers by one. The expression "`(a: Int) => a + 1`" returns an anonymous function that simply returns its argument incremented by 1. In Scala this is called a function literal and is used very frequently.

Now that we have an idea of what higher-order functions are, we want to look at closures. Closures are essentially functions that remember or "capture" the environment in which they are created. This can for instance be used to change their environment later on. To understand this better we present another example.

```
def newCounter: () => Int = {
  var c = 0
  def inc(): Int = {
    c += 1
    return c
  }
  inc
}

def test() {
  val c1 = newCounter
  println(c1()) // 1
  println(c1()) // 2

  val c2 = newCounter
  println(c2()) // 1

  println(c1()) // 3
}
```

Listing 1.3: **Closures**

In this example we define a function `newCounter` which returns functions that act as counters. The return type "`() => Int`" stands for a function of no arguments that returns an integer when it is called. In the function body, the `newCounter` function creates a local variable `c` and initializes it to `0`. The local function `inc` represents the counter, that will be returned. It captures the local variable `c` and increments it by one every time it is called, before returning the current value.

In the `test` function we create two different counters by invoking the `newCounter` function twice. We can observe that they both refer to distinct variables `c`.

These two examples should have covered the fundamental ideas behind higher-order functions and closures. On top of that, they have demonstrated how Scala can be used to program in a more functional style.

Pattern matching/Case classes

Pattern matching is a well-known feature in quite a few programming languages such as ML or Haskell and is also supported in Scala. It allows to match objects against patterns at run-time.

In Scala, it is possible to do pattern matching on user-defined case classes and also on many library types, such as lists or pairs. Case classes are special classes that have additional convenience methods to facilitate pattern matching.

The first example demonstrates this by rewriting the `map` function to use pattern matching. If the list is non-empty, the head and tail of the list get bound to the corresponding variables.

```
def map[A,B](list: List[A], fn: A => B) : List[B] = {
  list match {
    case Nil => Nil
    case head :: tail => fn(head) :: map(tail, fn)
  }
}
```

Listing 1.4: **`map` using pattern matching**

In the second example, we define our own case classes `Number` and `BinOp`, and use them to write a simple evaluator for syntax trees made up of these expressions.

```scala
abstract class Expr
case class Number(n: Double) extends Expr
case class BinOp(operator: String,
                 left: Expr, right: Expr) extends Expr

def eval(e: Expr) : Double =
  e match {
    case Number(n)         => n
    case BinOp("+", l, r) => eval(l) + eval(r)
    case BinOp("-", l, r) => eval(l) - eval(r)
  }

def test() {
  println(eval(Number(1)))                         // 1.0
  println(eval(BinOp("+", Number(2), Number(1)))) // 3.0
  println(eval(BinOp("-", Number(2), Number(1)))) // 1.0
}
```

Listing 1.5: **Case classes**

Traits/Mixins

As mentioned before, traits are one of Scala's key features. Traits or mixins have been known for a long time. They were originally introduced in the Flavors Lisp extension [11]. Later this idea was further refined by Bracha and Cook [7] and Schärli, Ducasse, Nierstrasz and Black [12] among others. Since then, quite a few, mostly dynamically typed languages, have adopted them.

Traits have been proposed as a solution to some of the problems associated with multiple inheritance, such as the "Diamond Problem" [7]. A trait in Scala is basically an abstract class that is parametrized with respect to its parent class. The parent class will only be "filled in" after the trait has been mixed into another class or trait. To make this more clear let us look at two short examples that each demonstrate an important aspect of traits.

```scala
abstract class Shape {
  def area: Double
  def canEqual(other: Any): Boolean
}

trait AreaEquals extends Shape {
  override def equals(o: Any): Boolean = {
    o match {
      case that: Shape =>
        (that.canEqual(this)) && area == that.area
      case _ => false
    }
  }
  override def hashCode: Int = area.hashCode
}

class Circle(val radius: Double) extends Shape with AreaEquals {
  def area = Math.Pi * radius * radius
  def canEqual(o: Any) = o.isInstanceOf[Circle]
}
```

Listing 1.6: **Shapes Example**

With this example we want to illustrate how traits can be used to limit the amount of code duplication. The goal is to define equality on geometric shapes. For certain shapes (e.g. circles, squares) we can check the equality between them by comparing their areas. For other shapes (e.g. rectangles) this might not be the best way to compare them.

We define an abstract class `Shape` that provides two abstract methods. `area` returns the area of this shape. `canEqual` returns true if we can compare this object with the other one.

Every Java programmer can confirm that it is not entirely trivial to override the `equals` method. The same holds for Scala as well. A very common problem is, that people forget to also override the `hashCode` method.

The new trait `AreaEquals` extends the abstract class `Shape`. This implies that every class that mixes in this trait has to be an instance of class `Shape`. `AreaEquals` overrides the `equals` and the `hashCode` methods just like it is supposed to. In the `equals` method, pattern matching is used to ensure that the other object is actually an instance of class `Shape` and has an `area` method consequently.

The class `Circle` can now simply mix in the trait `AreaEquals`, inheriting all of its methods. Of course, `Circle` has to implement the abstract method `canEqual` first, to make sure we only compare circles to other circles. The same would not be possible in languages that only allow single-inheritance with interfaces because the methods in `AreaEquals` would have to be abstract.

The second example will demonstrate another interesting aspect of traits. We start out with an existing class `C` that has a method `m`. Later on, it turns out that we sometimes have to log calls to this method. Unfortunately, it is not possible to change the existing method.

In Java or other languages that do not support traits, one possible solution would be to define a new subclass of `C` that overrides `m` and calls the super method after the logging is done. This works nicely if class `C` is the only class we have to modify. But this approach would generate a lot of duplicate code, if for instance some subclasses of `C` need the logging feature as well.

In general, languages like Java make it easy to add new classes that implement existing methods. But not the other way around. This problem is closely related to the "Expression Problem", which Odersky and Zenger solve in Scala using traits [13].

```
class C {
  def m() {
    ...
  }
}

trait Logged extends C {
  override def m() {
    print(timestamp() + ": ")
    super.m()
  }
}

trait Benchmarked extends C {
  override def m() {
    val old = timestamp()
    super.m()
    print((timestamp() - old) + "ms")
  }
}
```

Listing 1.7: **Logged-Benchmarked Example**

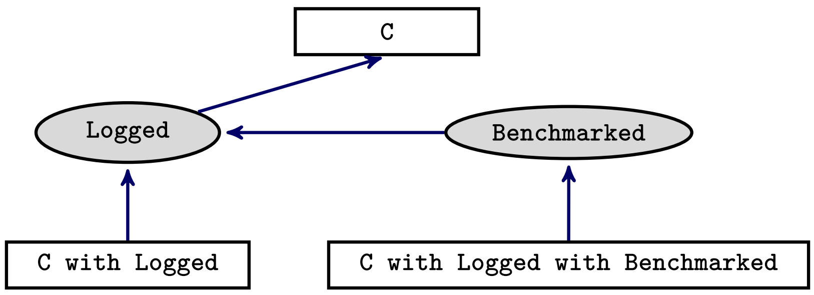
The Scala solution is amazingly straightforward. We define a new trait `Logged` that extends `C` and overrides `m`. Method `m` prints out the current time stamp and calls the super method to do the actual work. Any class, that needs the logging capabilities, can simply mix in the `Logged` trait.

The same strategy could be used to add benchmarking capabilities to existing classes. But what would we do if we want both the logging and the benchmarking capabilities?

Since a class can mix in arbitrarily many traits, we can just mix in both traits. The expression "`new C with Logged with Benchmarked`" would create an instance of `C` that does benchmarking and then logging. The order of the traits is important in this case because it affects the order of the super calls.

Figure 1.2 illustrates how a single call of the method `m` will propagate up via a chain of super calls.

FIGURE 1.2: **Chain of super calls**



Singleton/Companion objects

Singletons are types that have only a single instance. In most programming language singletons can be modeled using the well-known singleton pattern [14]. In contrast, Scala provides a separate language construct to create singletons. Here is an example that shows how to define one:

```scala
object Elvis extends Human with Musician {
  val name = "Elvis Aaron Presley"
  def sing() {
    println("lalala")
  }
}
```

LISTING 1.8: **Elvis the Singleton**

This defines and creates an object `Elvis` with a field `name` and a method `sing`.

A companion object is just a singleton object that shares its name with a class, its companion class. Unlike a regular singleton object, a companion object may access private members of its companion class and vis-versa. This makes it easy for multiple objects of the same class to have shared state, similar to static members in Java.

Concurrency

The last Scala feature that we will cover in this introduction is Scala's concurrency support. Scala borrows basic support for threads from Java and adds a few alternative concurrency models, such as Actors or Transactional Memory. Especially the Actors model [15] is worth looking into at this point, since its lack of shared state makes concurrent programs easier to verify. Instead of relying on locks and monitors to protect shared state, the pure Actors model uses message passing between threads. This eliminates the need for shared state and locks, which are difficult to reason about.

Unlike Erlang or other languages that only support the Actors model, Scala gives the programmer a choice about the best way to make its applications thread-safe. It is not uncommon to supplement the Actors model with some of Java's built-in concurrency abstractions, such as `ConcurrentHashMap` in the `java.util.concurrent` package.

So far, Spec# is not able to verify concurrent programs and one of the reasons for this is that Spec# relies on the thread model quite heavily. But since this is quite a large topic itself we will not be able to deal with concurrent programs in this project.

1.3.3 Challenges

Since Scala and Spec# are similar in a lot of ways, one may think that we could model some of the most common features in a similar fashion. This was certainly the case, although there are also a lot of features that Spec# does not support. The last couple of sections should be proof enough. Of course, these features were the most challenging ones to deal with. In particular, modeling closures, traits and singleton/companion objects were some of the most ambitious goals.

There have been a few proposals (e.g. by Müller and Ruskiewicz [16]) on how to model delegates in Spec# but none of them has been implemented yet. Delegates and closures have a lot in common, while the latter are somewhat more general because closures may refer to their lexical environment. This makes verification more difficult because the environment could even contain local variables, that do not usually have invariants attached to them. How would one ensure, for instance, that the counter objects in example 1.3 always increments the variable by one?

Traits were the second major challenge. The main reason for this, was that they make the modeling of the type hierarchy and super calls in methods more complex. Moreover, mixin-based inheritance in general is not quite as well-explored as single-inheritance with interfaces.

A third challenge were singleton and companion objects because they are unique to Scala. Initially, we were not sure whether they would just increase the complexity of our model or whether they would allow us to avoid some of the problems with static members in Spec#.

These are just a few quite fundamental difficulties that we considered to be worth noting here.

1.4 Boogie

1.4.1 What is Boogie?

Boogie is a modular reusable verifier for object-oriented programs [3, 4]. It was developed at Microsoft Research as part of their Spec# programming system [2, 1]. At first, it was only used to verify programs in their Spec# programming language. The Spec# programming language was also developed at Microsoft Research and is a variant of the C# programming language with support for design by contract. But since Boogie was designed to be reusable, it soon became quite popular as a backend for program verifiers in other languages such as C (e.g. VCC by Microsoft Research [17]) or Eiffel.

1.4.2 What is BoogiePL?

BoogiePL is a procedural language for checking object-oriented programs [18]. Just like Spec# and Boogie, it was developed at Microsoft Research as an intermediate language, between the high-level programming language Spec# and a theorem prover such as Z3 or Simplify.

Thus, another way to look at BoogiePL, is to see it as a high-level interface to theorem provers. Boogie can generate verification conditions out of BoogiePL code for a wide range of theorem provers. This flexibility allows us to use Boogie as a verification backend for our Scala verification tool by translating the Scala source code to BoogiePL first. Boogie can then be used to verify the generated code.

FIGURE 1.3: **Program verification with Boogie**

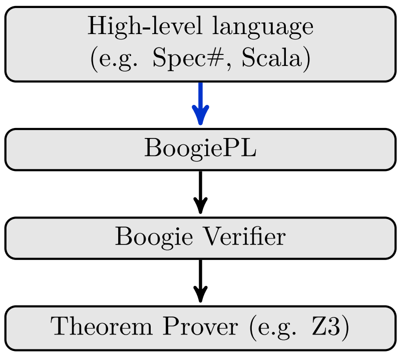


Figure 1.3 shows the steps that are necessary to verify a program with Boogie. During this project, we will focus on the first step, which generates BoogiePL code from Scala source code.

In the following subsections we give a quick overview of the BoogiePL language. We focus on the features that are necessary to understand BoogiePL programs that show up in this report. For further information about BoogiePL, I would like to refer the reader to Rustan Leino's "This is Boogie 2" [4] manual. BoogiePL has only seven top-level declarations, each of which will be explained.

Types

Type declarations are used to declare new type constructors, which may have type parameters. They may also be used to define aliases for existing types. There are only a small number of build-in types (e.g. int, bool). Listing 1.9 provides a few example declarations for each use-case. The first declaration defines a new type constructor for references. The second declaration defines a type constructor `List` that takes an arbitrary parameter, while the last declaration creates an alias to one particular list type.

```
type ref;
type List _;
type RefList = List ref;
```

LISTING 1.9: **type declarations**

Constants

Constant declarations give names to certain fixed (but unspecified) values of some type. Optionally they may be declared to be `unique`. All unique constants have values different from each other. Listing 1.10 demonstrates how (unique) constants may be used to define the RGB colors — red, green and blue.

```
type Color;
const unique RED: Color;
const unique GREEN: Color;
const unique BLUE: Color;
```

Listing 1.10: `const` declarations

Functions

Function declarations allow us to define mathematical (side-effect free) functions, which take a number of parameters and return a result. Functions might be polymorphic and they might return a fixed expression. Listing 1.11 defines two functions. `inverse#1` declares a function that takes an integer parameter and returns an integer. `inverse#2` on the other hand provides an "implementation" for that function on top of that.

```
function inverse#1(int) returns (int);
function inverse#2(i: int) returns (int) { -i }
```

Listing 1.11: `function` declarations

Axioms

Axiom declarations are used to state properties that should hold. For instance, we could add an axiom stating that function `inverse#1` actually returns the additive inverse (see listing 1.12).

```
function inverse#1(int) returns (int);
axiom (forall i: int :: inverse#1(i) == -i);
```

Listing 1.12: `axiom` declarations

Variables

Variable declarations define global variables, that may be modified by BoogiePL procedures. For example, we could declare a variable `favoriteColor` that holds our favorite color (see listing 1.13).

```
var favoriteColor: Color;
```

Listing 1.13: `var` declarations

Procedures

Procedure declarations provide procedure signatures that may optionally have additional specifications (e.g. pre- and postconditions) associated with them. Just like functions, procedures may take a number of parameters and might return a result. Procedures might also be polymorphic and can optionally have an implementation. For instance, we could introduce a procedure that sets the variable `favoriteColor`.

```
procedure setFavoriteColor(c: Color);
  modifies favoriteColor;
  ensures favoriteColor == c;
```

Listing 1.14: `procedure` **declarations**

Implementations

Implementation declarations add implementations to previously declared procedures. They essentially provide an implementation body to the existing procedure specification. Implementation bodies can consist of various declarations (e.g. local variables, labels) and statements (e.g. assignments, procedure calls). For example, we could associate an implementation with the previously defined `setFavoriteColor` procedure (see listing 1.15).

```
implementation setFavoriteColor(c: Color)
{
    favoriteColor := c;
    return;
}
```

Listing 1.15: `implementation` **declarations**

Chapter 2

Design

In this chapter we present our design of the Scala to BoogiePL translation tool. The first section will cover the overall design and will clarify what subset of the Scala language we are dealing with as part of this project.

The second section then presents how we encode certain Scala features in BoogiePL. Whenever possible, we try to include examples that help in illustrating the theoretical aspects.

2.1 Overall Design

We have already mentioned in Section 1.4.2 that we planed to design and implement a tool that translates a valid Scala program to BoogiePL. To achieve this we indented to modify Scala's code generator to emit BoogiePL code instead of Java bytecode. We were hoping that this would be relatively straight-forward once we knew how we could encode Scala's feature set in BoogiePL. But since Scala has so many features, we had to pick a subset that we would focus on primarily. This subset will be defined and presented in the following sections.

2.1.1 Supported Scala Features

We divided Scala's feature set up into different groups or categories. Each of them covers a separate aspect of the Scala language.

Object-Orientation

Since Scala is primarily an object-oriented programming language, this is the first aspect we will look into. But what exactly do we mean by "Object-Orientation"? In this context, we consider Object-Orientation to be a set of Scala features that affect the way objects behave at run-time (e.g. classes, constructors, field access, method calls).

Classes are basically blueprints for objects. This makes them a fundamental part of most object-oriented programming languages. Therefore we consider them to be an essential feature we will have to support. The same holds for constructors, methods and fields. Since we have to deal with classes, we will also have to deal with abstract classes and traits which are closely related.

Of course, we will also have to deal with objects themselves. In particular, we will have to think about how to represent the heap in BoogiePL. Singleton and companion objects are just variants of regular objects and we treat them in a similar way.

Since in Scala a class can only inherit from a single base class, we have to model single class inheritance. This is almost the same as in Spec#. At first we thought, that trait inheritance would make this more complex, but it turned out that we could treat trait inheritance separately just like interface inheritance is treated separately in Spec#.

We thought about including generics in this set of features. But since Scala uses type-erasure at run-time just like Java, the correct use of generics is already statically checked at compile-time. Thus, we did not deal with generics in this project, although in general they might still be useful for verification purposes.

Expressions

Expressions in Scala are, roughly speaking, everything that might be written inside of a method body. This includes calls to `new`, assignments, conditionals, while-loops and function/method calls. Our tool should handle all of these.

One interesting difference between Scala and Spec# is, that there is no clear distinction between expressions and statements in Scala. For example, an if-then-else construct returns a value (the last expression in one of the branches). The same thing also holds for other constructs, that would be statements in Spec#, such as loops. We model this by storing the returned value of an expression in a local variable.

Another difference worth noting here, is that Scala treats assignments to fields just like method calls to automatically generated getter- and setter-methods. This seems to be a mixed blessing because it is not entirely obvious what the default contract/specification for these setter- and getter-methods should look like. Making them too strong, might restrict classes, that override them, too much; making them too weak, will make it hard or even impossible to verify certain programs.

Types

We have already talked about Scala's type hierarchy in section 1.3.2 and have also compared it to Spec#'s type hierarchy.

Diagram 1.1 shows some of the built-in types that our tool will be dealing with. Additionally we will add support for arrays, functions and user-defined types. We will focus on pure Scala types instead of also worrying about Java classes and interfaces.

Scala's type hierarchy made it necessary to add the bottom types `scala.Nothing` and `scala.Null` and the top type `scala.Any` to the BoogiePL model and adapt the axioms accordingly. We also had to add the value type `scala.AnyVal` and its direct subtypes `scala.Int`, `scala.Boolean`, `scala.Char` and `scala.Unit`.

Other features

Another feature that we have to support is loop invariants. Since Scala currently does not come with a contract specification language, we might interpret the first statement of a loop body as the loop invariant if it is an `assert` statement. The main reason for adding loop invariants is, that it will be impossible to prove a lot of interesting things without them.

Some Scala features that we explicitly ignore because they are already enforced by the compiler include type safety/generics, access modifiers, packages and existential types.

The section on future work (4.2) mentions a couple of features that were left out intentionally and could be added in the future.

2.2 Detailed Design

This section explains our design in more detail. To keep it less abstract and theoretical we will use real BoogiePL code whenever it seems reasonable.

As we have mentioned previously in section 1.4.2, the Boogie verifier takes BoogiePL code as its argument and then tries to verify it. For practical reasons the input consists of two main parts.

The first one is a language specific prelude that encodes fundamental parts of the Scala programming language and its semantics, such as the heap model or the type hierarchy. The prelude also encodes certain commonly used objects and methods.

The second part contains the encoding of the particular program that should be verified. This includes things such as classes and their methods or fields.

The design of these two parts will be discussed in the following subsections. We will start out with the design of the prelude since this was one of the first major tasks during this project. This will lay the groundwork for Section 2.2.2, which will explain how we encode a given Scala program in BoogiePL using the axiomatizations from the prelude.

We will only show the actual BoogiePL code for the most interesting parts of the prelude. If the triggers do not add to the general discussion, they will be left out, even though they are an indispensable part of the actual prelude.

2.2.1 Heap Model & Type Hierarchy (Prelude)

Since the prelude is usually specific to the targeted programming language, we could not simply use an existing one. We therefore started out with the Spec# prelude and gradually modified it to capture the differences compared to the Scala semantics. Since this is the most interesting aspect, we will focus on how we adapted the existing Spec# prelude to fit Scala's and our needs.

The first thing we did was to remove everything from the prelude related to Spec#'s ownership model, frame conditions, pure functions and the .NET types.

Then we replaced the .NET types with Scala types to model the Scala type hierarchy and introduced types that have no Spec# counterpart, such as the top type `Any` or the bottom type `Nothing`.

Top types

Listing 2.1 shows how the top types are encoded in BoogiePL.

```
// Any (supertype of everything)
const unique scala.Any: TName extends complete;
axiom $IsClass(scala.Any);
axiom $IsMemberlessType(scala.Any);
axiom (!$IsReferenceType(scala.Any) && !$IsValueType(scala.Any));
axiom (forall $T: TName :: $T <: scala.Any);

// AnyRef (supertype of all reference types)
const unique java.lang.Object: TName extends unique scala.Any;
axiom $BaseClass(java.lang.Object) == scala.Any;
axiom $IsClass(java.lang.Object);
axiom (forall $T: TName ::
       ($T != scala.Nothing && $T <: java.lang.Object)
       <==> $IsReferenceType($T));

// AnyVal (supertype of all value types)
const unique scala.AnyVal: TName extends scala.Any complete;
axiom $BaseClass(scala.AnyVal) == scala.Any;
axiom $IsClass(scala.AnyVal);
axiom $IsMemberlessType(scala.AnyVal);
axiom (forall $T: TName ::
       ($T != scala.Nothing && $T <: scala.AnyVal)
       <==> $IsValueType($T));
```

Listing 2.1: **Encoding top types in BoogiePL**

We introduced several new functions:

- `$IsClass` returns true, if the type models a class.
- `$IsTrait` returns true, if the type models a trait.
- `$IsReferenceType` returns true, if the type is a reference type.
- `$IsValueType` returns true, if the type is a value type.

We also added an axiom that guarantees that types are not both reference and value types. A similar axiom was added for the two functions `$IsClass` and `$IsTrait`.

The function `$IsMemberlessType` returns true for types that are abstract.

For the `Any` type we added an axiom stating that it is a supertype of every other type. Another axiom states that `Any` is neither a reference type nor a value type.

For the class `AnyRef`, which is functionally equivalent to Java's `Object` class, we added axioms saying that its base class is `Any` and that all of its sub types, except `Nothing`, are reference types. We have to exclude `Nothing`, because it is neither a reference nor a value type.

`AnyVal`'s axiomatization is similar to `AnyRef`, except that it is an abstract type and it is the supertype of all value types.

We also experimented quite a bit with Boogie's `extends`, `unique` and `complete` keywords, which essentially allow the programmer to add more restrictive subtyping information to a type.

For example, the "`extends complete`" clause in `Any`'s `const` declaration states that it does not have any direct supertypes and that all its direct subtypes are known.

`AnyRef`'s "`extends unique scala.Any`" clause states that `Any` is its only direct supertype and that `AnyRef`'s subtypes cannot extend from `Any` directly.

Essentially, these declarations are just syntactic sugar and could just as well be added as axioms. Rustan Leino's "This is Boogie 2" [4] manual explains these declarations in more detail. It turned out that we could not use these declarations for a lot of types because they only seem to work nicely in the absence of a bottom type. We also had to change quite a few other existing axioms because of this. Unfortunately, tracking down these axioms was quite difficult because they simply introduce inconsistencies and there is no automatic way of finding the axioms that caused them in the first place.

Bottom types

Now let us have a look at these bottom types. Listing 2.2 includes the important parts of the prelude.

```
// Nothing (subtype of everything)
const unique scala.Nothing: TName;
axiom $IsTrait(scala.Nothing);
axiom $IsMemberlessType(scala.Nothing);
axiom $IsFinal(scala.Nothing);
axiom (!$IsReferenceType(scala.Nothing)
       && !$IsValueType(scala.Nothing));
axiom (forall $T: TName :: scala.Nothing <: $T);

// Null (subtype of all reference types)
const unique scala.Null: TName;
axiom $IsTrait(scala.Null);
axiom $IsFinal(scala.Null);
axiom (forall $T: TName ::
       ($T == scala.Any || $IsReferenceType($T)) <==> scala.Null <: $T);
axiom (forall $T: TName ::
       $T <: scala.Null ==> ($T == scala.Null || $T == scala.Nothing));
axiom (forall $r : ref ::
       $typeof($r) == scala.Null <==> $r == null);
```

LISTING 2.2: **Encoding bottom types in BoogiePL**

We were not able to use the `extends` clauses for the `Nothing` and `Null` types because they basically extend an arbitrary number of types.

Another interesting fact is that these types are modeled as traits in Scala. We know that traits are abstract in general. But we also mentioned that `null` is an instance of type `Null`. This seems to be somewhat contradictory. Still we decided to model it like this for consistency with Scala's type declarations and left out an axiom stating that all traits are also abstract. Instead we explicitly state this for every trait except `Null`.

Besides being traits, these types are also marked `final`. This obviously makes sense for `Nothing`, but not so much for `Null` because `Nothing` is its subtype. We decided that this is not a problem if we treat `Nothing` and `Null` as special types that even final types may be supertypes of.

In practice, final traits are used very rarely. One possible usage scenario is to use them to create phantom types, which are somewhat popular in the Haskell community. They are often used to enforce certain constraints that can be checked by the type checker instead of relying on run-time checks.

ScalaObject

Another important trait from the prelude is `ScalaObject`. Its encoding is shown in Listing 2.3.

```
// abstract trait scala.ScalaObject extends java.lang.Object
const unique scala.ScalaObject: TName;
axiom $IsMemberlessType(scala.ScalaObject);
axiom $IsTrait(scala.ScalaObject);
axiom (scala.ScalaObject <: java.lang.Object);
axiom (forall t: TName ::
       ((scala.ScalaObject <: t) && (t != scala.ScalaObject))
       ==> (java.lang.Object <: t));
axiom $BaseClass(scala.ScalaObject) == java.lang.Object;
```

Listing 2.3: **Encoding `ScalaObject` in BoogiePL**

Just as for `Any` and `AnyRef`, we do not use the `extends` clauses here. Instead we added two axioms. The first one simply states that `ScalaObject` is a subtype of `AnyRef`. The second one states that all supertypes of `ScalaObject` are also supertypes of `AnyRef`, which implies that `AnyRef` is the only direct supertype.

Value types

Since value types are not implemented in Scala itself, we also had to model them in the prelude. The relevant parts of the prelude are shown in Listing 2.4. The declarations for `Boolean` and `Char` are very similar to the ones for `Int`. During the implementation, it turned out that we had to add more code to the prelude that deals with operations (e.g. arithmetic operations on integers or logical operations on booleans) on value types. This will be covered in more detail during the discussion of the implementation details in Chapter 3.

```
const unique scala.Int: TName extends scala.AnyVal;
axiom $BaseClass(scala.Int) == scala.AnyVal;
axiom $IsClass(scala.Int);
axiom $IsFinal(scala.Int);

const unique scala.Unit: TName extends scala.AnyVal;
axiom $BaseClass(scala.Unit) == scala.AnyVal;
axiom $IsClass(scala.Unit);
axiom $IsFinal(scala.Unit);
const unique unit: ref;
axiom (forall $r : ref ::
       $typeof($r) == scala.Unit <==> $r == unit);
```

Listing 2.4: **Encoding value types in BoogiePL**

All value types extend `AnyVal` and are final classes. Additionally we had to define a new constant `unit`, which stands for the unit value (`()` in Scala). This is the only instance of the `Unit` class. We could have also modeled it as a singleton object, but we wanted our encoding to resemble the Scala declarations as closely as possible.

Since in Scala everything is an object, values are also objects and have to be stored on the heap (more about this in the next section). But Boogie supports certain value types natively (e.g. int, bool) and we have to use these if we want to proof interesting properties. This made it necessary to store the corresponding Boogie values together with the Scala value objects. These corresponding Boogie values are stored in a two-dimensional array `$ValueFields`, that maps a reference to a Scala value and a field name to the respective Boogie value. For instance, a reference to a Scala `Int v` would be associated with a Boogie value stored in `$ValueFields[v, $int]`.

Heap and Fields

To model fields, we use the type `Field` that has one generic argument (e.g. `Field ref` for fields that hold references). This allows us to model the heap as a global variable that refers to a two-dimensional array. This array maps an object reference and a field to the corresponding value of that field.

```
type Field _;

type HeapType = <beta>[ref,Field beta]beta;

var $Heap: HeapType where IsHeap($Heap);
```

Listing 2.5: **Encoding of fields and the heap in BoogiePL**

Listing 2.5 shows the corresponding declarations in the prelude. `IsHeap` is a function that is basically used to tag valid heaps. We thought about removing the generic `beta` argument for the `HeapType`, since we make no distinction between fields to value and reference objects. This might be something that could be changed in the future. Since this was not our main focus, we decided not to change it in the end.

To get the field `f` of an object `o`, we can access the heap like this: `$Heap[o, f]`. To store a new value `v` in that field we assign to the heap variable like this: `$Heap[o, f] := v`.

We also experimented with a special encoding for "val fields", since Scala supports fields that are constant. These fields are initialized in the constructor of a class and cannot be changed later on. To encode this we added a new field `$ValFieldsAreInitialized` that is set to true after an object has been constructed (see Listing 2.6).

```
function $IsValField<alpha>(f: Field alpha) returns (bool);

const unique $ValFieldsAreInitialized: Field bool;

function #UltimateValue<alpha>(ref, Field alpha) returns (alpha);

axiom (forall o: ref, f: Field ref, h: HeapType ::
      { IsHeap(h), $IsValField(f), #UltimateValue(o, f) }
      IsHeap(h) && h[o, $ValFieldsAreInitialized] && $IsValField(f)
      ==> h[o, f] == #UltimateValue(o, f));
```

Listing 2.6: **Encoding of "val fields" in BoogiePL**

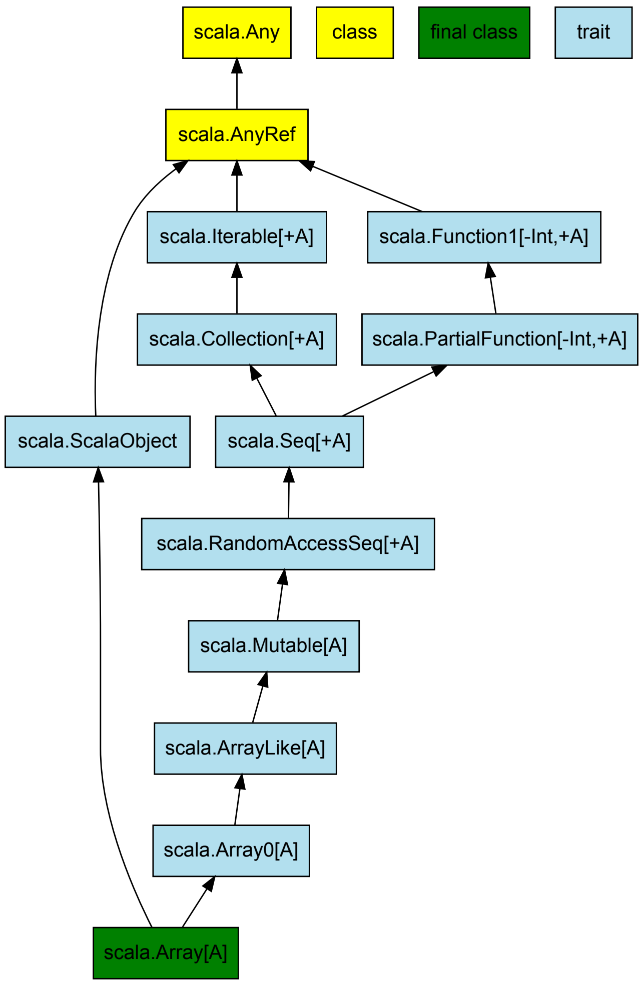
The property that this field never changes its value is encoded by the axiom shown in Listing 2.6, which basically states that once the field is initialized, its value does not depend on the heap. We used this particular encoding because it lets us use very specific triggers. This helps Boogie a lot in the verification process.

Arrays

Arrays are the last type we discuss here. There are a couple of interesting differences between the way arrays are modeled in Scala and in Spec# or Java. The most important one is certainly that arrays are not a “built-in” or “primitive” type in Scala. `Array` is just a regular class in the class hierarchy.

The `Array` class is also a good example of how traits are used in Scala programs. Figure 2.1 illustrates this very well by showing all of `Array`’s supertypes.

FIGURE 2.1: **`Array`’s supertypes**



Since most of `Array`’s supertypes are traits, `Array` can mix in not only a whole lot of abstract methods, but can also reuse a lot of concrete methods. For instance, `Array` mixes in `Function1`, which is a trait that all functions of one argument mix in. Arrays can therefore be treated as functions that take the array index and return the corresponding element. It is therefore not necessary to have a special syntax for array access in Scala.

All of this made it necessary to add more traits to the prelude. Listing 2.7 shows just a small part of the corresponding declarations. The declarations for the other supertraits are quite similar to the declarations for `Array0` and none of these supertraits are essential for this report.

These declarations are pretty straightforward. Notice that the base class of both `Array` and

```
// trait Array0 extends ArrayLike
const unique scala.Array0: TName;
axiom $IsTrait(scala.Array0);
axiom $IsMemberlessType(scala.Array0);
axiom scala.Array0 <: scala.ArrayLike;
axiom scala.Array0 <: scala.ScalaObject;
axiom (forall t: TName :: scala.Array0 <: t && t != scala.Array0
       ==> (scala.ArrayLike <: t || scala.ScalaObject <: t));
axiom $BaseClass(scala.Array0) == java.lang.Object;

// final class Array extends Array0
const unique scala.Array: TName;
axiom $IsClass(scala.Array);
axiom scala.Array <: scala.Array0;
axiom scala.Array <: scala.ScalaObject;
axiom (forall t: TName :: scala.Array <: t && t != scala.Array
       ==> (scala.Array0 <: t || scala.ScalaObject <: t));
axiom $BaseClass(scala.Array) == java.lang.Object;
```

Listing 2.7: **Encoding `Array` and `Array0` in BoogiePL**

its supertraits is `AnyRef`. The reason for this will be explained in a later section on how traits are encoded.

```
type Elements _;

const unique $elementsRef: Field (Elements ref);

function ArrayGet<alpha>(Elements alpha, int)
    returns (alpha);

function ArraySet<alpha>(Elements alpha, int, alpha)
    returns (Elements alpha);
```

Listing 2.8: **Encoding Array access in BoogiePL**

Listing 2.8 shows how array operations are modeled. The field `$elementsRef` stores the elements of an array on the heap. Our prelude also formalizes arrays of primitive types, such as integers, but we only use arrays of references in the end. We kept the other types of arrays for now, but they could probably be removed at some point.

Multidimensional arrays can be encoded as arrays of arrays, since this is the way arrays are handled in Scala, whereas Spec# also supports "true" multidimensional arrays as an alternative to these "jagged arrays".

The functions `ArrayGet` and `ArraySet` are used to get and set the elements of an array. The prelude contains a lot of axioms that encode certain properties that should hold. For example, setting an array element to a value `v` and accessing it should return the same value `v`.

We also had to adapt many axioms to ensure that arrays are invariant in their element type. This means for instance, that the following declaration is not allowed in Scala, unlike in Spec# or Java: "`var a: Array[Super] = new Array[Sub](1)`", where `Sub` is a subtype of `Super` (`Sub <: Super`).

Difficulties with Boogie

Now that we covered the most interesting parts of the prelude, I would like to add a few words about my experience with Boogie. The hardest part about designing the prelude was to find out how to express properties in a way that Boogie can deal with efficiently. I often thought of ways to encode properties, that would be very easy for humans to understand, only to discover that unfortunately Boogie does not (yet) pass the Turing test.

This should not necessarily be seen as a criticism, but as a word of caution. There usually does exist a way to encode things that Boogie knows how to deal with. It might just take some time to find it.

Another difficulty was that it turned out to be quite tricky to "debug" the prelude. Most often this was necessary because there was an inconsistency somewhere in the prelude that would allow Boogie to proof anything ("ex falso quodlibet"). But finding these inconsistencies could take hours, because it was usually very difficult to track down the axioms that caused the inconsistency. This difficulty is not really specific to Boogie though.

2.2.2 Translation to BoogiePL

Now that we presented the Scala prelude, we want to look at how specific parts of Scala programs are translated to BoogiePL. We basically turn the spotlight on various fundamental parts of a Scala program. First, we will look at how Scala's basic building blocks — classes, abstract classes, traits and singleton objects — are axiomatized in BoogiePL on the type-level. In the next sections, we present how fields, constructors and methods of a class, trait or object are translated to BoogiePL. And finally, we focus on method/constructor bodies and closures.

Classes, Traits and Singleton Objects

This section describes how we modeled classes, traits and singleton objects in BoogiePL. For now, we concentrate on the type-level. This includes how these constructs are integrated into Scala's type hierarchy and how we can encode certain properties about them (e.g. final or abstract classes). We start with "normal" classes before we move on to abstract classes, traits and finally singleton objects.

Every class has a base class. In fact, every Scala type has a base class. But the base class itself must be a class. It cannot be a trait or a singleton object. Since in Scala classes are only allowed to inherit from a single super class the super class and the base class are the same thing. To make this whole discussion less theoretical we will look at a simple example (see Listing 2.9) and look at the BoogiePL code that this translates to.

```scala
abstract class C

trait U

trait V

trait T extends C with U with V
```

```

class D extends C with T

object O extends D
```

Listing 2.9: **Classes, Traits and Singleton Objects in Scala**

In Listing 2.10 we see that there is an axiom that states that `C` is `D`'s base class. We also have axioms, that declare all the direct super types of class `D`, including the axiom that declares that there are no others. The direct super types of class `D` are of course the base class `C`, the trait `T` and implicitly also the trait `ScalaObject`. To make class `D` abstract, we would have to add an additional axiom, saying that it is a memberless type (`axiom $IsMemberlessType(C);`). The same holds for final classes (`axiom $IsFinal(E);`).

```
const unique D: TName;
axiom $IsClass(D);
axiom (D <: C);
axiom (D <: T);
axiom (D <: scala.ScalaObject);
axiom (forall t: TName :: (((D <: t) && (t != D))
        ==> (C <: t) || (T <: t) || (scala.ScalaObject <: t)));
axiom $BaseClass(D) == C;
```

Listing 2.10: **Classes in BoogiePL**

Traits are encoded just in a slightly different way (see Listing 2.11). Traits are always memberless, just like abstract classes. Traits also have a single base class, which has the additional property that it restricts the set of classes, that may mix in that trait, to the subclasses of its base class. In our example trait `T` does not have any concrete methods. Therefore it is basically an interface and does not mix in `ScalaObject` implicitly. The order of the traits is not encoded yet, because it does not matter for the type hierarchy. It only matters for the super calls.

```
const unique T: TName;
axiom $IsMemberlessType(T);
axiom $IsTrait(T);
axiom (T <: C);
axiom (T <: U);
axiom (T <: V);
axiom (forall t: TName :: (((T <: t) && (t != T))
        ==> (C <: t) || (U <: t) || (V <: t)));
axiom $BaseClass(T) == C;
```

Listing 2.11: **Traits in BoogiePL**

Singleton objects are basically non-abstract final classes that have only a single instance. This is exactly how we encode them in BoogiePL (see Listing 2.12). The prelude contains two axioms that help in encoding these properties (see Listing 2.13). Besides this we also declare the singleton instance as a constant reference, that can be used to refer to the singleton.

```
const unique O: TName;
axiom $AsSingleton(O) == O;
```

```

const unique O$$instance: ref;
axiom $typeof(O$$instance) == O;

axiom (O <: D);
axiom (O <: scala.ScalaObject);
axiom (forall t: TName :: (((O <: t) && (t != O))
        ==> (D <: t) || (scala.ScalaObject <: t)));
axiom $BaseClass(O) == D;
```

Listing 2.12: **Singleton Objects in BoogiePL**

```
// A singleton may only have a single instance.
axiom (forall S: TName, o1: ref, o2: ref ::
        $typeof(o1) == S && $typeof(o2) == S
        && $AsSingleton(S) == S ==> o1 == o2);
// A singleton is final.
axiom (forall S: TName :: $AsSingleton(S) == S ==> $IsFinal(S));
```

Listing 2.13: **Axioms from the Prelude**

Fields

So far, we only looked at how to encode "empty" classes, traits or singleton objects. Since fields are a integral part of objects, we will look at how to encode them first. In Scala, there exist two different kinds of fields — "val fields" and "var fields". We already showed how to model "val fields" in section 2.2.1. "var fields" are fields that can be changed. For these, Scala will provide both a getter- and a setter-method. Whereas for "val fields" there will be just a getter-method. Section 2.2.2 will cover how these methods look like in BoogiePL. In this section we will only look at how fields are declared.

The easiest way to explain this, is by looking at another example. Our very simple Scala program declares a class with two fields (see Listing 2.14). Fields and variables always have to be initialized in Scala. But since this happens in the constructor, we will discuss this in Section 2.2.2 on constructors.

```
class C {

  val i: Int = 1

  var b: Boolean = true

}
```

Listing 2.14: **Class with two fields in Scala**

```
// var C.b: scala.Boolean
const unique C.b: Field ref;
axiom DeclType(C.b) == C;
axiom AsRefField(C.b, scala.Boolean) == C.b;
```

```

// val C.i: scala.Int
const unique C.i: Field ref;
axiom DeclType(C.i) == C;
axiom AsRefField(C.i, scala.Int) == C.i;
axiom $IsValField(C.i);
```

Listing 2.15: **Class with two fields in BoogiePL**

Instead, let us look at how the corresponding BoogiePL declarations look like (see Listing 2.15). We have to use the field's fully qualified name (e.g. `C.b` instead of `b`) to prevent name clashes. For our purposes all fields are reference fields and both of them are declared in class `C`. The types of the fields are `scala.Boolean` for field `b` and `scala.Int` for field `i`. The conceptual difference between them is that we add an axiom for field `i`, stating that it is a “val field”.

Constructors

In Scala, a class has a single primary constructor and possibly a certain number of auxiliary constructors, that have to call the primary constructor eventually. Only the primary constructor may call the super constructor. Scala's rules are a little more strict than in other languages, such as Spec# or Java. Traits and singleton objects do not have auxiliary constructors.

Just like methods, constructors are modeled as procedures in BoogiePL, that take the `this` reference as their first argument. Since Scala supports method and constructor overloading we have to generate unique procedure names by appending the types of the method/constructor parameters to the actual name. The following example should highlight some more important points (see Listing 2.16).

```
trait T

trait U {
  val u: AnyRef = null
}

class D(val d: AnyRef)

class C(val c: AnyRef) extends D(null) with T with U {
  def this() {
    this(null)
  }
}
```

Listing 2.16: **Constructors in Scala**

Class `C` has two constructors. The primary constructor takes one argument while the auxiliary constructor takes no arguments at all and calls the primary constructor.

```
procedure C..ctor$$scala.AnyRef(
    $this: ref where ($IsNotNull($this, C) && $Heap[$this, $allocated]),
    c: ref where $Is(c, scala.AnyRef));
  // The val fields are initialized.
  free ensures (($typeof($this) == C)
                  ==> $Heap[$this, $ValFieldsAreInitialized]);
  free ensures $HeapSucc(old($Heap), $Heap);
  // The target object is allocated upon return.
  free ensures $Heap[$this, $allocated];
  modifies $Heap;

implementation C..ctor$$scala.AnyRef($this: ref, c: ref)
{
    // Initialize fields.
    assert($this != null);
    $Heap[$this, C.c] := c;
    assume(IsHeap($Heap));

    // Call the super constructors.
    call D..ctor$$scala.AnyRef($this, null);
    call U..ctor($this);

    // Execute the rest of the constructor (e.g. method calls).
    ...
}
```

Listing 2.17: **Primary constructors in BoogiePL**

The general pattern can be observed in Listings 2.17 and 2.18. The "procedure" declarations specify the interface of the constructors while the "implementation" declaration contains the actual constructor body. The primary constructor initializes the fields before calling the super constructors; starting with the base class constructor, followed by the trait constructors (from left to right in the "extends" clause).

In our example trait T does not have a constructor because it is completely abstract, just like an interface. The auxiliary constructor, first calls the primary constructor before doing anything else.

```
procedure C..ctor(
    $this: ref where ($IsNotNull($this, C) && $Heap[$this, $allocated]));

  // The val fields are initialized.
  free ensures (($typeof($this) == C)
                ==> $Heap[$this, $ValFieldsAreInitialized]);

  free ensures $HeapSucc(old($Heap), $Heap);
  // The target object is allocated upon return.
  free ensures $Heap[$this, $allocated];
  modifies $Heap;

implementation C..ctor($this: ref) returns ($result: ref)
{
    // Call the primary constructor.
    call C..ctor$$java.lang.Object($this, null);

    ...
}
```

Listing 2.18: **Auxiliary constructors in BoogiePL**

Methods

Methods and constructors are encoded in very similar ways. In fact, constructors are special kinds of methods. Just like constructors they are encoded as two different BoogiePL declarations — the “procedure”, that expresses the method signature together with the specification, and the “implementation”, that holds the method body. The separation of these two method fragments allows us to model abstract methods in a natural way, by merely providing a procedure declaration without a matching implementation.

Before we can move on to method bodies in the next section, let us look at another example that focuses on super calls (see Listings 2.19 and 2.20).

```
class C {

  def m() { }

}

trait T extends C {

  override def m() {
    super.m()
  }

}
```

Listing 2.19: **Super calls in Scala**

Trait `T` calls the super method `super.m` in method `m`. Since `T` is a trait, we cannot tell which method will ultimately be called. This depends on the class that mixes in trait `T`. But we would like to be able to verify `T` even if no class has mixed in this trait yet.

If method `T.m` contains, for instance, a division by zero, we can spot this even though no class has mixed in trait `T`. To make this possible, we introduce an abstract method `T$$super$m`, that classes will have to implement when they mix in the trait `T`. In the method `T.m`, this abstract super method `T$$super$m` will be called instead of an actual super method. The example also illustrates that all method calls in Scala return a value, even if it is just the `unit` value.

```
implementation T.m($this: ref) returns ($result: ref)
{
  var $tmp$1$3: ref where $Is($tmp$1$3, scala.Unit);

    call $tmp$1$3 := T$$super$m($this);
}

procedure T$$super$m(
    $this: ref where ($IsNotNull($this, T) && $Heap[$this, $allocated]))
    returns ($result: ref where $Is($result, scala.Unit));
```

Listing 2.20: **Super calls in BoogiePL**

Method/Constructor Bodies

So far, we have only covered the encoding of method and constructor signatures. This section discusses the translation of method and constructor bodies to BoogiePL. As mentioned earlier, a Scala method body is transformed to an "implementation" declaration in BoogiePL. These method bodies can contain different declarations and instructions, such as variables declarations, assignment statements, conditionals, loops or method calls.

Variable declarations in Scala can be nested; the inner declarations "shadow" the outer ones. This feature is not supported in BoogiePL; instead all variables have to be declared at the beginning of the implementation. Since we want to support variable "shadowing" we have to introduce unique names for all variables. We also have to support additional temporary variables. These are mostly used to store the result of statements that are expressions at the same time (e.g. conditionals, method calls).

Assignments are either directly translated to BoogiePL assignment statements (e.g. assignments to local variables) or to a call of the corresponding setter-procedure (e.g. assignments to fields). The latter case is encoded just like a normal method call.

We decided to translate conditionals to BoogiePL if-then-else statements. The alternative would have been to encode them using labels and goto statements, which is how Spec# encodes them. We mainly decided to use if-then-else statements to improve the readability of the generated code. The price for this is, that we have to deal with the issues of indenting the code correctly.

Initially, we wanted to do the same for while loops. It turned out though, that loops are already translated to labels and goto statements in a previous phase of the Scala compiler. Since we did not want to revert this transformation, we also use labels and goto statements in the BoogiePL code that we generate. For the verifier this does not make a difference. The only thing we had to think about was how to encode loop invariants.

The following pseudo-Scala code illustrates how loop invariants could be encoded if Scala would allow to specify them (see Listings 2.21 and 2.22). A similar transformation allows us to encode do-while loops.

```
while (COND)
invariant: I_1
free invariant: I_2
{
    BODY
}
```

Listing 2.21: **Loop invariants in Scala**

```
while_label:
  assert(I_1);
  assume(I_2);
  if (COND) {
      BODY
      goto while_label;
  } else {
  }
  assert(I_1);
  assume(I_2);
```

Listing 2.22: **Loop invariants in BoogiePL**

Method calls are translated to procedure calls in BoogiePL. They always return a result, which is always stored in a temporary variable. In the future, this could be optimized in many cases because methods are frequently called simply for their side-effects. For now, we preferred to keep things simple and focus on more interesting problems.

Closures

The last major design decision we will discuss concerns the way we model closures. In the introductory section 1.3.2 on closures and higher-order functions we have seen the Scala syntax for declaring closures together with a simple example. In this section we demonstrate how closures can be transformed in such a way that they can later be expressed in BoogiePL.

In the examples from section 1.3.2, we used a special notation to express the types of functions (e.g. `A => B` or `() => Int`). This notation is just syntactic sugar and will be transformed to a more conventional type that has generic parameters for the types of the functions arguments and the return value (e.g `Function1[A, B]` or `Function0[Int]`. The Scala type hierarchy contains different generic traits for functions of different arities (from `Function0` up to `Function22`). Note that we have already pointed out in Section 2.2.1 on arrays that, arrays mix in the trait `Function1[Int, A]`. The most important part of this "family" of traits is the abstract `apply` method. It takes the arguments of the function and comprises the body of the function; calling it will evaluate the function. Every function can therefore be expressed as a class that mixes in one of these "Function traits". Listing 2.23 demonstrates this transformation on a simple example.

```
// original function
def succ(x: Int): Int = x + 1

// transformation
class Succ extends Function1[Int, Int] {
  def apply(x: Int): Int = x + 1
}
```

Listing 2.23: **Transforming functions**

Now how does this work for closures? What happens to the captured variables? An additional complication is that multiple closures might capture the same variable. To illustrate the needed transformations, we modified example 1.3 slightly (see Listing 2.24).

```
def newCounter: (() => Int, () => Int) = {
  var c = 0

  def inc(): Int = {
    c += 1; c
  }
  def dec(): Int = {
    c -= 1; c
  }
  (inc, dec)
}

def test() {
  val (c1Inc, c1Dec) = newCounter
  println(c1Inc()) // 1
  println(c1Dec()) // 0
  println(c1Inc()) // 1

  val (c2Inc, c2Dec) = newCounter
  println(c2Dec()) // -1
}
```

Listing 2.24: **Counters**

The function `newCounter` now returns a pair of closures that both refer to the same lexical environment. Executing these closures will increment and decrement the same variable. To model this kind of aliasing we have to create a wrapper object for the captured variable. Listing 2.25 illustrates the necessary transformation. Note that the `IntRef` class serves as a wrapper class for integers. Similar classes also exist for other types.

The transformed program shows how the formerly nested functions `inc` and `dec` were lifted up to the method level and they now take the captured variable as their argument. In practice, the classes `Inc` and `Dec` are anonymous and they take all captured variables as arguments in their constructor. This makes the aliasing very explicit. After these transformations, the program can be translated to BoogiePL by applying procedures from previous sections.

```
def inc(c: IntRef): Int = {
  c.elem += 1
  c.elem
}

def dec(c: IntRef): Int = {
  c.elem -= 1
  c.elem
}

class Inc(c: IntRef) extends Function0[Int] {
  def apply(): Int = inc(c)
}

class Dec(c: IntRef) extends Function0[Int] {
  def apply(): Int = dec(c)
}

def newCounter: (Function0[Int], Function0[Int]) = {
  var c = new IntRef(0)

  (new Inc(c), new Dec(c))
}
```

Listing 2.25: **Counters (Transformation)**

We also though about other ways of modeling closures. In the end, we decided to use this encoding because the Scala compiler does this already and we tried to reuse existing functionality whenever possible.

Chapter 3

Implementation

This chapter describes the implementation of our Scala verification tool in more detail. Section 3.1 covers the general architecture of the Scala compiler and how it can be modified using compiler plugins. The following Section 3.2 presents our BoogiePL Abstract Syntax Tree (AST) library which was used to abstract over the low-level parts of the code generation. This library was then used by our code generation plugin to transform the Scala AST to a BoogiePL AST. This transformation is the main topic of Section 3.3. The last two sections mention a few difficulties that we encountered and present a couple of programs that our code generator already deals with.

3.1 Scala Compiler Architecture

This section describes the overall architecture of the Scala compiler. In particular, we will look at the various compiler phases and how we can extend the compiler with plugins.

The Scala compiler is based on a pipeline of phases. Each phase takes an AST representation of a program as its input, does something with it and finally passes a potentially modified AST to the next phase in this linear chain. Only the last phase will emit actual code that can be run for instance on the JVM. The first few phases are traditionally responsible for type checking the input program and enforcing other properties. This way, the later phases can assume that the input adheres to the basic conventions of the programming language.

Since producing executable code is the ultimate goal of a compiler, the remaining phases transform the AST in various ways that usually try to simplify certain parts or boil them down to use more primitive constructs. For example, a phase might transform while- or do-loops to labels and goto statements. Some phases may also perform certain optimizations that make the resulting code faster or smaller in size. Dead code removal is just one example for these kinds of optimizations.

In Scala, it is possible to write new compiler phases and plug them into the existing compiler pipeline. Of course, custom plugins should not transform the AST in ways that later phases cannot deal with. Unfortunately these dependencies among the various existing phases are not well documented. There are plans to make these kinds of dependencies more explicit in the upcoming 2.8 release of the Scala compiler and even enforce them automatically. During this project we still used the latest stable release of the compiler (2.7.5). Table 3.1 provides a good overview of the existing phases and includes a short description (taken from the Scala compiler sources). They are listed in the order that they are called in. Since not all of them are relevant for us, we will not cover them in detail here.

TABLE 3.1: **Existing compiler phases**

Phases	**Short Description**
namer, parser, typer	Parsing, type checking etc.
superAccessors	Adds super accessors.
pickler	Serializes symbol tables.
refchecks	Performs reference and override checking; translates nested objects.
liftcode	Generates reified trees.
uncurry	Translates function values to anonymous classes.
tailCalls	Replaces tail calls by jumps.
explicitOuter	Replaces "C.this" by explicit outer pointers; eliminates pattern matching.
erasure	Erases generic types to Java 1.4 types; adds interfaces for traits.
lazyVals	Transforms local lazy vals into vars and initialized bits.
lambdaLift	Moves nested functions to top level.
constructors	Moves field definitions into constructors.
flatten	Gets rid of inner classes.
mixer	Does mixin composition and translates lazy fields.
cleanup	Some platform-specific cleanups
genicode	Generates portable intermediate code.
inliner	Does inlining (optimization).
closureElimination	Gets rid of uncalled closures (optimization).
deadCode	Gets rid of dead code (optimization).
jvm	Generates Java bytecode.

We tried to reuse most of the existing phases because we wanted to focus on the actual code generation and some of the phases already deal with transformations that would needed anyways. Some examples are the "uncurry", "lambdaLift", "constructors" or "mixer" phases. For now, we are not using any of the existing optimization phases to keep things simple. Of course it would be possible to add more optimizations in the future. But this was not our main goal. Instead we added our code generation plugin right after the "mixer" phase. It turned out later, that this causes some unforeseeable issues in the implementation. These will be discussed in more detail in Section 3.4.

3.2 BoogiePL AST Library

In this section we introduce the BoogiePL Abstract Syntax Tree (AST) library that we designed and implemented in Scala, before we started with the implementation of the code generation plugin itself. The idea behind this library was to separate the component, that deals with the low-level intricacies of generating valid and readable BoogiePL code, from the component, that actually performs the high-level transformations on the Scala AST. When we later started working on the code generation plugin, this library turned out to be a huge help, because it allowed us to focus only on the transformations of the Scala AST. At the moment, this is the main purpose of this library, but it could be easily used for other tasks, such as optimization.

We started out with the basic design of the AST. This was guided by the BoogiePL Backus-Naur Form(BNF) from Rustan Leino's "This is Boogie 2" [4] manual. Although we do not use all the features in the BNF description, we added them to the AST design. After all, we could not yet foresee which features we would use exactly.

This rudimentary AST was then extended with methods that produce the actual BoogiePL code or deal with indentation. Getting the indentation right, was not completely trivial and made it necessary to adapt the way we produced the generated code. Initially we had just used a simple string as the output. This had to be changed to lines that could be indented more easily.

Now how would you test something like this? How do you discover bugs in a library when it is not used yet? Writing unit tests for all these possible cases was clearly not an option. And we did not want to use a buggy library in our plugin because we could never be sure what caused an error — the library or the code we were trying to write. To escape this dilemma we decided to use the ScalaCheck testing tool [19] which was originally inspired by QuickCheck for Haskell [20]. It makes it easy to automatically generate hundreds of test cases. For instance, we used it to generate random AST nodes. These AST nodes could then be used to check that some methods satisfy certain properties. We also automatically tested whether the generated BoogiePL code was syntactically correct by running Boogie on it.

This way of testing our library was very useful because it allowed us to eliminate a lot of bugs right away. In fact, we did not discover a single bug in the library while we were developing the compiler plugin. Instead we even found bugs in the Boogie verifier which demonstrates that our tests were quite thorough (see Appendix A for more details on minor problems in Boogie that we discovered). It was just not always easy to track down the parts of our program that made Boogie crash or emit funny error messages.

Still, we continued to add more features to the library while we actually used it in our plugin. The ones that were not directly linked to the AST were put into their own packages. For example, there is a package `Util.Axioms` that contains a lot of axioms that are frequently used in the code generator.

3.3 Code Generation Plugin

Now that we have discussed the BoogiePL AST library, we can demonstrate how it was used in our code generation plugin to translate the Scala code to BoogiePL.

The Scala AST contains a lot of information that is not relevant for our plugin. It is often quite complicated to extract the bits that we need from the AST. We therefore decided to minimize the places in our code that actually work with the Scala AST directly. This will also help in keeping the plugin up to date if the structure of the AST should change. For bigger language constructs, such as classes, traits or methods, we defined our own classes that take the Scala AST as an argument in their constructor.

Most relevant parts of the Scala AST are then extracted by initializing “val fields” with them or defining methods that extract them. Parts of the implementation that need some information from the Scala AST can then use the fields or methods that know how to extract the information. For instance, there exists an abstract class `ScalaTopLevelDefinition` that already extracts a lot of information from the Scala AST. Special top-level definitions, such as classes or traits, are subclasses of class `ScalaTopLevelDefinition` and can therefore reuse a lot of the methods and fields already defined there or override them if necessary. For classes, traits and objects we have to extract a lot of different information from the AST. The most important information are the modifiers (abstract, final, sealed), super types, methods, fields and constructors.

We created similar classes for methods, constructors and fields, which allowed us to reuse a lot of functionality. This almost meant that we created something close to a simplified Scala AST. But we only did this for bigger constructs. For statements or expressions in method bodies, we used a traversal function that recursively traverses the Scala AST.

This `traverse` function is quite large, much like the commonly used `visit` method in an AST visitor. It is not really complex but it has to deal with quite a lot of different cases, which still looks quite clean thanks to Scala’s pattern matching. Even in the `traverse` function we tried to minimize the accesses to the Scala AST. We usually had quite a few local variables that stored the relevant parts of the AST, so we could easily refer to them later. This turned out to be very useful because the code that actually extracted the important bits from the AST was only used once and could be changed very easily.

To test the plugin itself, we wrote mainly a test, that looks for Scala sources in a testing directory and runs the Scala compiler with our plugin on it. The BoogiePL output is then compared to a reference output. If they are not the same, a "diff" tool is called to visualize the parts that are different. But since there is no definite reference output, we used this mostly as a regression test and to visualize the changes we made to the plugin.

We also run Boogie on the output to check the BoogiePL code for errors. To make this more convenient for the user, we wrote a simple Boogie output processor, that can recognize different kinds of errors by parsing the Boogie error messages.

3.4 Difficulties

In this section we will mention a few difficulties that we encountered during the implementation phase. All of them are related to the existing 'erasure' phase in the Scala compiler. In the following paragraphs we explain the issues that we encountered and show what Scala features are mainly affected by this. Moreover we propose several possible resolution strategies.

First, we would like to add a few general remarks on the way we approached the task of developing the plugin and the inherent difficulties. Since the existing code generation plugins for the JVM or .NET platform were way too specialized and complex, we were not able to reuse parts of them. Sometimes, for lack of documentation, we looked at them in order to understand how the Scala AST was structured. Other than that, we basically started from scratch.

During the design phase, we had already thought about the way we wanted to encode the Scala code, we had written templates and we had used them to encode small programs. What remained to do, was to express the templates and extract all the necessary parameters from the Scala AST. We tested the results on small programs and essentially applied test driven development techniques. We tried hard to minimize the number of features that these programs needed, besides the ones that we were currently working on. But initially, a lot of programs produced huge amounts of Boogie errors, because a lot of features were not yet implemented. This number of errors could only be reduced over time and Boogie did not always display all errors because it checks for errors in multiple phases. We could therefore never be quite sure, that the feature we just implemented actually worked correctly. The result was that a lot of issues did not show up until the plugin could handle most features. Except for a few more intricate issues these could be fixed quite easily though.

All of these turned out to be caused by the 'erasure' phase. The first issue came up when we worked on encoding traits. The Scala AST node for traits contains a field that holds the supertypes of a trait. But we noticed that some supertypes were missing. At first we could not find out what caused this loss of information. Due to the size of the existing code base, the very sparse documentation and the not very fruitful responses from the Scala community, it took us longer than expected to track down this issue. In the end, we were quite sure that the 'erasure' phase was responsible for this loss of information, although this was not confirmed by the Scala developers. Our subsequent experiments in removing the 'erasure' phase or changing at least parts of it, made us realize that other phases depend on the 'erasure' phase and it would not be possible to simply remove it.

Initially, we thought that only a few of the phases that run after the 'erasure' phase, were affected by this. But after more testing we were quite sure that all of them depended on the 'erasure' phase for some reason. Some did not work at all without the 'erasure' phase and some just produced garbage output. The bottom line is that by keeping the 'erasure' phase as-is, our axioms concerning the supertypes are weaker than they could be.

The 'erasure' phase is also responsible for two other issues. Because this phase erases certain types (e.g. `Any`) it has to introduce a couple of extra operations in the AST to compensate for this. For value types it introduces explicit boxing and unboxing operations and additionally it introduces casts.

Summing up, the main issue is that we cannot remove the ‘erasure’ phase because of the phases that depend on it and we cannot keep it, because it removes certain information in the AST and introduces unnecessary operations instead. The next section demonstrates these issues by also showing example programs that do not quite work yet.

There are a few ways in which we could resolve these issues. Unfortunately none of them seems easy enough to be implemented within the remaining time; especially because testing them is rather intricate and time-consuming. The “optimal” solution would be to collaborate with the Scala development team on removing some of the dependencies by modifying the affected phases. But we would have to see whether they are interested in changing things that work for them.

Another solution would be to implement these phases ourselves. This would be a huge effort and would not be easy to maintain because we would have to keep up with the evolution of the Scala language. A more pragmatic solution would be to store the missing information before it is erased and then access it again in our plugin. This would require some bookkeeping but we would not have to mess with the phases that depend on the ‘erasure’ phase.

3.5 Case Studies / Example Programs

This final section presents example programs that demonstrate that our code generator produces meaningful code for the features that we intended to support. Since even short Scala programs produce BoogiePL programs that are rather long, we can usually highlight only a few snippets, that show the encoding of one or two interesting features. Generally we first show a complete Scala example together with parts of its encoding in BoogiePL and a short discussion. These discussions are meant to highlight a few points that were left out before, but are interesting nonetheless.

3.5.1 Classes with Methods and Fields

```
class D

class C extends D {
  val x = 1
  var y = 2

  def m(x: Int) {

  }

  def m(x: Any): Int = {
     1
  }
}
```

```
implementation C..ctor($this: ref) returns ($result: ref)
{
  var $tmp$1$10: ref where $Is($tmp$1$10, scala.Int);
  var $tmp$3$10: ref where $Is($tmp$3$10, scala.Unit);
  var $tmp$2$10: ref where $Is($tmp$2$10, scala.Int);
  var $tmp$4$10: ref where $Is($tmp$4$10, scala.Unit);

    // ----- Constant: 1 -----
    havoc $tmp$1$10;
    assume((!$Heap[$tmp$1$10, $allocated] && ($tmp$1$10 != null
           && ($typeof($tmp$1$10) == scala.Int
           && $ValueFields[$tmp$1$10, $int] == 1))));
    $Heap[$tmp$1$10, $allocated] := true;
    assume(IsHeap($Heap));
    // ----- store field -----
    assert($this != null);
    $Heap[$this, C.x] := $tmp$1$10;
    assume(IsHeap($Heap));
    // Initialize field C.y (left out).
    // ----- call -----
    call $tmp$3$10 := D..ctor($this);
    // ----- Constant: () -----
    $tmp$4$10 := unit;
    // ----- set result -----
    $result := $tmp$4$10;
    return;
}
```

Listing 3.1: **Primary constructor in BoogiePL**

Listing 3.1 shows the almost complete encoding of `C`'s primary constructor. As shown in Section 2.2.2 we first have to initialize the fields, before we call the super constructors. The result of a constructor call is always `unit`. To initialize the fields, we have to create a new integer object. First we assign an arbitrary value to a temporary variable using the `havoc` statement. Then we use the `assume` statement to restrict this arbitrary value, making it an unallocated non-null reference to the heap of type `scala.Int`. Additionally we set the integer's Boogie value to `1`. Then we allocate the reference by setting the `$allocated` field on the Heap to `true` and assume that the resulting heap is valid. This new integer object can then be used to initialize the field `C.x`. The initialization only works like this, if the 'erasure' phase is run, because the 'constructors' phases depends on it.

```
// setter for y
procedure C.y_$eq$$scala.Int(
    $this: ref where ($IsNotNull($this, C) && $Heap[$this, $allocated]),
    x$1: ref where $Is(x$1, scala.Int))
    returns ($result: ref where $Is($result, scala.Unit));

// method m(Any): Int
procedure C.m$$scala.Any(
    $this: ref where ($IsNotNull($this, C) && $Heap[$this, $allocated]),
    x: ref where $Is(x, scala.Any))
    returns ($result: ref where $Is($result, scala.Int));
```

```

// method m(Int): Unit
procedure C.m$$scala.Int(
    $this: ref where ($IsNotNull($this, C) && $Heap[$this, $allocated]),
    x: ref where $Is(x, scala.Int))
    returns ($result: ref where $Is($result, scala.Unit));
```

Listing 3.2: **Classes with methods and fields in BoogiePL**

In Listing 3.2, we demonstrate how the method signatures for overloaded methods and setter-methods are translated. The setter-method follows a special naming convention in Scala. In this case the name of the setter-method is `y_=`, which has to be escaped to `C.y_$eq` in BoogiePL. For overloaded methods we have to append the types of its arguments to make the method names unique.

3.5.2 Objects with methods and fields

The encoding for singleton objects is very similar to the encoding for regular classes. If we rewrite the previous example slightly we can observe only minor changes in the BoogiePL output (see Listing 3.3). The most significant change is that instead of simply using the `$this` reference, we can directly refer to the singleton object using `C$$instance`. Listing 3.4 demonstrates this in a short snippet.

```
class D

object C extends D {
  val x = 1
  var y = 2

  def m(x: Int) {

  }

  def m(x: Any): Int = {
    1
  }
}
```

Listing 3.3: **Objects with methods and fields in Scala**

```
assert(C$$instance != null);
$Heap[C$$instance, C.x] := $tmp$1$10;
assume(IsHeap($Heap));
```

Listing 3.4: **Initializing singleton objects in BoogiePL**

3.5.3 Method Bodies

Our next example demonstrates how method bodies are encoded (see Listing 3.5). It will cover all the important expressions and statements, such as assignments, conditionals, loops and method calls. Besides we will see how simple arithmetic expressions can be translated to BoogiePL.

```
class C {
  def factorial(n: Int): Int = {
    var result = 1
    var i = n

    while (i > 1) {
      result = i * result
      i = i - 1
    }

    return result
  }

  def not(b: Boolean): Boolean = {
    if (b) false else true
  }
}
```

Listing 3.5: **Method bodies in Scala**

```
implementation C.factorial$$scala.Int($this: ref, n: ref)
    returns ($result: ref)
{
  var i$2: ref where $Is(i$2, scala.Int);
  var result$2: ref where $Is(result$2, scala.Int);
  // more local variable declarations

    // result$2 := 1

    i$2 := n;

  while$1:
    // $tmp$2$2 := 1

    call $tmp$3$2 := scala.Int.$greater$$scala.Int(i$2, $tmp$2$2);

    if ($ValueFields[$tmp$3$2, $bool]) {
        call $tmp$1$5 := scala.Int.$times$$scala.Int(i$2, result$2);
        result$2 := $tmp$1$5;

        // $tmp$2$5 := 1

        call $tmp$3$5 := scala.Int.$minus$$scala.Int(i$2, $tmp$2$5);
        i$2 := $tmp$3$5;

        $tmp$1$4 := unit;
        goto while$1;
        $tmp$4$2 := $tmp$1$4;
    } else {
        $tmp$1$6 := unit;
        $tmp$4$2 := $tmp$1$6;
    }

    $result := result$2;
    return;
}
```

Listing 3.6: **`factorial` in BoogiePL**

We had to leave out quite a lot of not so interesting parts from the resulting BoogiePL code (see Listing 3.6). The parts, that were left out, are shown as pseudo code in the comments. We can see how the local variables `i` and `result` are declared and that their names have to be made unique. They are initialized before the loop. The loop itself is encoded exactly like in Section 2.2.2. We only left out the loop invariants. The arithmetic and logical expressions are encoded as procedure calls. The corresponding procedures had to be added to the prelude. Adding all of these procedures would have been quite tedious. Thus, we focused on a few important ones. But adding more of them is straightforward.

This example also illustrates that `unit` is always the result of a while loop. The last important thing to notice here, is that we have to access the `$ValueFields` variable in the conditional expression. This is necessary because the conditional expression has to be of the native Boogie type `bool`.

```
implementation C.not$$scala.Boolean($this: ref, b: ref)
    returns ($result: ref)
{
  var $tmp$1$7: ref where $Is($tmp$1$7, scala.Boolean);
  var $tmp$1$8: ref where $Is($tmp$1$8, scala.Boolean);
  var $tmp$1$9: ref where $Is($tmp$1$9, scala.Boolean);

    if ($ValueFields[b, $bool]) {
        // $tmp$1$8 := false
        $tmp$1$7 := $tmp$1$8;
    } else {
        // $tmp$1$9 := true
        $tmp$1$7 := $tmp$1$9;
    }
    $result := $tmp$1$7;
    return;
}
```

Listing 3.7: **`not` in BoogiePL**

The encoding of the `not` method is interesting because it demonstrates the use of a conditional as an expression. The last expression of each branch is assigned to a temporary variable, which in turn is assigned to the `$result` variable of our method.

3.5.4 Arrays

In this example, we focus on arrays (see Listing 3.8). The example features the three most important operations on arrays — creation, setting an element and getting an element. In Section 2.2.1 on the design of arrays, we already mentioned that arrays are treated just like ordinary objects in Scala. From the programmer's perspective, there is nothing special about them.

The generated BoogiePL code clearly demonstrates that we wanted to use the same principles in our encoding (see Listing 3.9). To make this possible we had to add procedures (e.g. `scala.Array..ctor$$scala.Int`) to the prelude that hide and encapsulate our Boogie-specific encoding of arrays. This is quite convenient, since we do not have to treat arrays as a special case in the code generator.

```
object Test {
  def test {
    val a: Array[Int] = new Array[Int](10)
    a(1) = 1
    a(0)
  }
}
```

Listing 3.8: **Arrays in Scala**

```
implementation Test.test($this: ref) returns ($result: ref)
{
  var a$3: ref where $Is(a$3, scala.Array);
  // more local variable declarations

    // $tmp$1$3 := 10

    // ----- new scala.Array -----
    havoc $tmp$2$3;
    assume((!$Heap[$tmp$2$3, $allocated]
            && ($tmp$2$3 != null && $typeof($tmp$2$3) == scala.Array)));

    call $tmp$3$3 := scala.Array..ctor$$scala.Int($tmp$2$3, $tmp$1$3);

    // ----- initializing a$3 -----
    a$3 := $tmp$2$3;

    // $tmp$4$3 := 1
    // $tmp$5$3 := 1

    call $tmp$6$3 := scala.Array.update$$scala.Int$$scala.Int(
                     a$3, $tmp$4$3, $tmp$5$3);

    // $tmp$1$4 := 0
    call $tmp$2$4 := scala.Array.apply$$scala.Int(a$3, $tmp$1$4);

    // ----- Constant: () -----
    $tmp$3$4 := unit;
    // ----- set result -----
    $result := $tmp$3$4;
    return;
}
```

LISTING 3.9: **Arrays in BoogiePL**

3.5.5 Closures

The example in this section (see 3.10) is just a slight variation of the example in the general discussion about closures from Section 1.3.2. We had to modify it because it used pairs and we do not support them. Since this is one of the examples, that is most affected by the issues that were discussed in Section 3.4, we demonstrate some issues directly in the generated BoogiePL code.

```
object Test {
  def newCounter: () => Int = {
    var c = 0
    def inc(): Int = {
      c += 1; c
    }
    inc
  }

  def test() {
    val c1 = newCounter
    c1() // 1
    c1() // 2

    val c2 = newCounter
    c2() // 1
    c1() // 3
  }
}
```

LISTING 3.10: **`newCounter` in Scala**

Before we look at the generated BoogiePL code, we should look at some of the transformations that are necessary to process this program. The first important transformation is to turn the result value `inc` in `newCounter` into an instantiation of the anonymous class `AnonFun`. This happens in the 'uncurry' compiler phase. The second important transformation lifts the nested function `inc` up to the method level. This happens in the 'lambdaLift' compiler phase. This phase also introduces the `IntRef` wrapper objects that are necessary to support aliasing. Listing 3.11 illustrates both of these transformations.

```
object Test {

  def inc(c: IntRef): Int = {
    c.elem += 1
    c.elem
  }

  class AnonFun(c: IntRef) extends Function0[Int] {
    def apply(): Int = inc(c)
  }

  def newCounter: Function0[Int] = {
    var c = new IntRef(0)

    new AnonFun(c)
  }

  ...
}
```

LISTING 3.11: **`newCounter` in Scala (transformed)**

When we now look at the generated BoogiePL code, we observe that all generics were removed and instead there are new methods and boxing instructions, that compensate for the loss of information. For instance, in Listing 3.12 we can see that there are now two `apply` methods — one returning `scala.AnyRef`, the other returning `scala.Int`. The former acts as a wrapper for the latter and boxes the integer result before it returns it as a `scala.AnyRef` object. This boxing would not be necessary if the 'erasure' phase would replace all generic types with `scala.Any` instead of `scala.AnyRef`. Besides, the 'erasure' phase also seems to remove `scala.Any` completely, which is mainly a problem for code that deals with value types.

```
// method Test$$anonfun$newCounter$1.apply(): scala.AnyRef
procedure Test$$anonfun$newCounter$1.apply(
    $this: ref where ($IsNotNull($this, Test$$anonfun$newCounter$1)
                       && $Heap[$this, $allocated]))
    returns ($result: ref where $Is($result, scala.AnyRef));

// method Test$$anonfun$newCounter$1.apply(): scala.Int
procedure Test$$anonfun$newCounter$1.apply(
    $this: ref where ($IsNotNull($this, Test$$anonfun$newCounter$1)
                       && $Heap[$this, $allocated]))
    returns ($result: ref where $Is($result, scala.Int));
```

LISTING 3.12: **`apply` method in BoogiePL**

3.5.6 Traits

In this last section, we have another look at traits and the Logged-Benchmarked example (see 1.7). We focus particularly on the way super calls are encoded and on the issues with the 'erasure' phase. Listing 3.13 defines a few classes that extend the `Logged` or `Benchmarked` traits.

```
class CLoggedBenchmarked extends C with Logged with Benchmarked
class CBenchmarkedLogged extends C with Benchmarked with Logged
class CLogged extends C with Logged
```

Listing 3.13: **Logged-Benchmarked Example (revised)**

The generated code for the class constructors (see listing 3.14) looks exactly like we designed it in Section 2.2.2. It demonstrates again, that the order of the traits is significant.

```
implementation CBenchmarkedLogged..ctor($this: ref)
    returns ($result: ref)
{
    ...

    call $tmp$1$35 := C..ctor($this);

    call $tmp$2$35 := Benchmarked..ctor($this);

    call $tmp$3$35 := Logged..ctor($this);

    ...
}

implementation CLoggedBenchmarked..ctor($this: ref)
    returns ($result: ref)
{
    ...

    call $tmp$1$39 := C..ctor($this);

    call $tmp$2$39 := Logged..ctor($this);

    call $tmp$3$39 := Benchmarked..ctor($this);

    ...
}
```

Listing 3.14: **Super calls in constructors**

In Listing 3.15 we can observe that the same holds for methods as well. Both implementations of the `Benchmarked$$super$m` method call different super methods, because the order of the traits is different.

```
// in CLoggedBenchmarked
implementation Benchmarked$$super$m($this: ref) returns ($result: ref)
{
  var $tmp$1$3: ref where $Is($tmp$1$3, scala.Unit);

    call $tmp$1$3 := Logged.m($this);

    $result := $tmp$1$3;
    return;
}

// in CBenchmarkedLogged
implementation Benchmarked$$super$m($this: ref) returns ($result: ref)
{
  var $tmp$1$8: ref where $Is($tmp$1$8, scala.Unit);

    call $tmp$1$8 := C.m($this);

    $result := $tmp$1$8;
    return;
}
```

Listing 3.15: **Super calls in methods**

The only issue that we found in this example is due to the ‘erasure’ phase. The generated code verifies just fine, although the second axiom is obviously too weak (see Listing 3.16). The correct axiom would be “`axiom (Logged <:  C);`” instead. Sometimes this fact might be needed, and in these cases, the ‘erasure’ phase will insert casts. But now the verifier does not know that the corresponding cast is in fact an up-cast and hence legitimate.

```
axiom (Logged <: scala.AnyRef);

axiom (Logged <: scala.ScalaObject);
```

Listing 3.16: **Issue with the ‘erasure’ phase**

Chapter 4

Conclusions

4.1 Related Work

Software verification is a very broad and active field of research, that produced a lot of different approaches and methodologies over the last decades. The existing tool that had the most profound influence on this project is certainly the Spec# verification system, which was itself strongly influenced by the “Extended Static Checker for Java” (ESC/Java) [21] and its Java Modeling Language (JML) [22]. Our prelude was itself inspired by the Spec# prelude and we also studied the way certain language features are encoded by the Spec# system.

Müller and Ruskiewicz worked on a verification methodology for C# delegates [16]. Later Nordio, Calcagno, Meyer and Müller proposed another verification methodology for function objects that focuses on Eiffel’s agents [23]. Both C# delegates and agents in Eiffel are already quite similar to Scala’s closures. But closures are more general because they may capture local variables in their lexical environment. There even exists a proposal to add closures in Java by Bracha, Gafter, Gosling and von der Ahé [24]. Our encoding of closures is different from the encoding of agents by Nordio, Calcagno, Meyer and Müller. The main reasons are that they have specifications in Eiffel and closures in Scala are more general than agents in Eiffel, because closures can capture local variables.

4.2 Future Work

Like we said in the introduction to this report, this project was only a first step in the direction of creating a static program verifier for Scala. Hence there are still plenty more things to do until this goal is reached. In this section we propose a few next steps and extensions for this project.

The first step would be to sort out the remaining issues from Section 3.4. We already sketched a few possible resolution scenarios and the most promising at this point seems to be the one, that stores the AST before the ‘erasure’ phase, so that the original information is not lost completely. Adding the necessary bookkeeping and testing the resulting implementation are the major difficulties.

To test more and larger programs it will be necessary to extend the prelude significantly. For example, we had to add a lot of operations on value types (e.g. addition for integer values). So far, we added these things only incrementally.

At some point, a specification methodology has to be added of course. Without this, it will not be possible to verify any interesting programs. The question of how to address the frame problem for Scala methods and closures in particular will be of great importance in this context.

Especially the implicit setter- and getter-methods in Scala will be difficult to verify without precise framing specifications. Adding specifications to variables that are captured by one or more closures will also require some work.

Another problem that we have not addressed yet, is the initialization of singleton objects. For now, we simply encode the constructors of singleton objects. Everywhere the singleton object is used, we should be able to assume that its constructor has already been executed before.

Since the Scala programming language allows the programmer to access Java classes, it might also be interesting to look into whether this might make it necessary to adapt the prelude or how this influences the verification of larger Scala programs that make use of Java classes and interfaces.

Finally, the Scala programming language offers many interesting features that we have not been able to address in this project. The following seem to be particularly noteworthy:

- Unchecked exceptions
- Concurrency (e.g. Actors model)
- Type system (e.g. generics, abstract types, variance annotations, existential types, path-dependent types)
- Pattern matching
- Implicit conversions
- Curried functions
- Lazy values

4.3 Conclusions

The main goal of this project was the design and implementation of a tool that translates a subset of the Scala programming language to the BoogiePL programming language.

To achieve this goal, we first designed a formal model of the Scala heap structure and its type hierarchy. This model was encoded in BoogiePL and is part of our Scala prelude. Next, we designed a encoding for a subset of the Scala programming language in BoogiePL. Initially, this encoding was done “by hand” and only tested on small example programs to ensure its practicality.

Once we had enough confidence in this encoding, we began to design and implement a tool that can do this for us. The main component of this tool is a Scala compiler plugin, that transforms the Scala AST to our own BoogiePL AST. It makes heavy use of the BoogiePL AST library, which produces the actual BoogiePL code and makes sure that the code is easily readable, as well as syntactically correct. The syntactic correctness of the produced code was extensively tested using the ScalaCheck testing tool.

The Scala compiler plugin supports a usable subset of the Scala programming language. This includes support for traditional object-oriented programming constructs (classes, methods and fields) and more Scala specific constructs, such as traits, closures or singleton objects. During the implementation, we discovered certain problems with the existing Scala compiler phases that we were not able to work around due to the limited amount of time that we had. But these are merely technical issues that can be resolved and do not put into question the practicality of our encoding. We even sketched several possible solutions in Section 3.4.

List of Figures

1.1 Scala's type hierarchy 14
1.2 Chain of super calls 19
1.3 Program verification with Boogie 21

2.1 `Array`'s supertypes 32

List of Tables

3.1 Existing compiler phases 44

Listings

1.1 Type inference . . . 14
1.2 `map` . . . 15
1.3 Closures . . . 16
1.4 `map` using pattern matching . . . 16
1.5 Case classes . . . 17
1.6 Shapes Example . . . 17
1.7 Logged-Benchmarked Example . . . 18
1.8 Elvis the Singleton . . . 19
1.9 `type` declarations . . . 21
1.10 `const` declarations . . . 22
1.11 `function` declarations . . . 22
1.12 `axiom` declarations . . . 22
1.13 `var` declarations . . . 22
1.14 `procedure` declarations . . . 23
1.15 `implementation` declarations . . . 23
2.1 Encoding top types in BoogiePL . . . 27
2.2 Encoding bottom types in BoogiePL . . . 29
2.3 Encoding `ScalaObject` in BoogiePL . . . 29
2.4 Encoding value types in BoogiePL . . . 30
2.5 Encoding of fields and the heap in BoogiePL . . . 31
2.6 Encoding of “val fields” in BoogiePL . . . 31
2.7 Encoding `Array` and `Array0` in BoogiePL . . . 33
2.8 Encoding Array access in BoogiePL . . . 33
2.9 Classes, Traits and Singleton Objects in Scala . . . 34
2.10 Classes in BoogiePL . . . 35
2.11 Traits in BoogiePL . . . 35
2.12 Singleton Objects in BoogiePL . . . 35
2.13 Axioms from the Prelude . . . 36
2.14 Class with two fields in Scala . . . 36
2.15 Class with two fields in BoogiePL . . . 36
2.16 Constructors in Scala . . . 37
2.17 Primary constructors in BoogiePL . . . 37
2.18 Auxiliary constructors in BoogiePL . . . 38

2.19 Super calls in Scala 39
2.20 Super calls in BoogiePL 39
2.21 Loop invariants in Scala 40
2.22 Loop invariants in BoogiePL 40
2.23 Transforming functions 41
2.24 Counters 41
2.25 Counters (Transformation) 42
3.1 Primary constructor in BoogiePL 47
3.2 Classes with methods and fields in BoogiePL 48
3.3 Objects with methods and fields in Scala 49
3.4 Initializing singleton objects in BoogiePL 49
3.5 Method bodies in Scala 50
3.6 `factorial` in BoogiePL 50
3.7 `not` in BoogiePL 51
3.8 Arrays in Scala 51
3.9 Arrays in BoogiePL 52
3.10 `newCounter` in Scala 52
3.11 `newCounter` in Scala (transformed) 53
3.12 `apply` method in BoogiePL 53
3.13 Logged-Benchmarked Example (revised) 53
3.14 Super calls in constructors 54
3.15 Super calls in methods 54
3.16 Issue with the ‘erasure’ phase 55
A.1 Nullary functions 69
A.2 `free modifies` clauses 69
A.3 Final types 70

Bibliography

[1] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004), volume 3362 of LNCS, pages 49–69. SV, 2005. 11, 20

[2] http://research.microsoft.com/en-us/projects/specsharp/. 11, 20

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, volume 4111 of LNCS, pages 364–387. SV, 2006. 11, 20

[4] K. Rustan M. Leino. This is Boogie 2. (Draft). 11, 20, 21, 28, 44

[5] http://www.scala-lang.org/. 11, 13

[6] Martin Odersky. The Scala Language Specification 2.7. École polytechnique fédérale de Lausanne, Switzerland, January 2009. 11, 13

[7] Gilad Bracha and William R. Cook. Mixin-based Inheritance. In OOPSLA/ECOOP, pages 303–311, 1990. 13, 17

[8] Martin Odersky, Lex Spoon, and Bill Venners. Programming in Scala - A comprehensive step-by-step guide. artima, 2008. 13

[9] Martin Odersky. Scala By Example. http://www.scala-lang.org/docu/files/ScalaByExample.pdf, April 2009. 13

[10] Philipp Haller Michel Schinz. A Scala Tutorial for Java programmers. http://www.scala-lang.org/docu/files/ScalaTutorial.pdf, March 2009. 13

[11] David A. Moon. Object-Oriented Programming with Flavors. In OOPSLA, pages 1–8, 1986. 17

[12] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable Units of Behaviour. In ECOOP 2003 – Object-Oriented Programming, pages 327–339. Springer, 2003. 17

[13] Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In Proc. FOOL 12, January 2005. http://homepages.inf.ed.ac.uk/wadler/fool. 18

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. Addison-Wesley Professional, January 1995. 19

[15] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In IJCAI, pages 235–245, 1973. 19

[16] P. Müller and J. N. Ruskiewicz. A modular verification methodology for C# delegates. In U. Glässer and J.-R. Abrial, editors, Rigorous Methods for Software Construction and Analysis, 2007. To appear. 20, 57

[17] http://research.microsoft.com/en-us/projects/vcc/. 20

[18] Rob DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005. 20

[19] http://code.google.com/p/scalacheck/. 45

[20] Koen Claessen and John Hughes. Quickcheck: A Lightweight Tool for Random Testing of Haskell Programs. In ACM SIGPLAN Notices, pages 268–279. ACM Press, 2000. 45

[21] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, pages 234–245, New York, NY, USA, 2002. ACM. 57

[22] G. Leavens, A. L. Baker, and C. Ruby. JML: a notation for detailed design. In I. Kilov, B. Rumpe, and I. Simmonds, editors, Behavioral Specifications of Businesses and Systems, pages 175–188. Kluwer, 1999. 57

[23] M. Nordio, C. Calcagno, B. Meyer, and P. Müller. Reasoning about Function Objects. Technical Report 615, ETH Zurich, 2009. 57

[24] Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé. Closures for the Java Programming Language (v0.5). http://www.javac.info/closures-v05.html. 57

[25] Ioannis T. Kassios. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In FM, pages 268–283, 2006.

[26] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In ACM Conference on Generative Programming and Component Engineering (GPCE'08). ACM, 2008.

[27] Adriaan Moors, Frank Piessens, and Martin Odersky. Towards equal rights for higher-kinded types. In 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages, 2007.

[28] K. R. M. Leino and P. Müller. Modular verification of static class invariants. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, Formal Methods (FM), volume 3582 of Lecture Notes in Computer Science, pages 26–42. Springer-Verlag, 2005.

[29] Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with Classes, Mixins, and Traits. In Naoki Kobayashi, editor, APLAS, volume 4279 of Lecture Notes in Computer Science, pages 270–289. Springer, 2006.

[30] Martin Odersky et al. An Overview of the Scala Programming Language. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne, 2006.

Appendix A

Minor Problems with Boogie

While developing and testing the BoogiePL AST library we discovered a few minor problems with Boogie. We list them in the following sections together with a short description and a program that demonstrates them. They were also reported to the Boogie developers.

A.1 Boogie crashes for nullary functions

```
function f() returns (r: t)
{
    x
}
```

Listing A.1: **Nullary functions**

This turned out to be a bug. It was fixed very quickly.

A.2 `free modifies` clause results in a syntax error

```
var x: int;

procedure p();
free modifies x; // Doesn't work.
modifies x;      // Works.
```

Listing A.2: **`free modifies` clauses**

Error message: `syntax error: invalid SpecPrePost`

This had not yet been implemented. It has been added to the Boogie Issue Tracker.

A.3 Final types are no longer supported

```
type finite t1; // Doesn't work.
type t1;        // Works.
type finite t2;
```

Listing A.3: **Final types**

Error message: `Error: more than one declaration of type name: finite`

It turned out that final types are no longer supported.