Verifying Scala Traits

Malte Schwerhoff

Semester Project Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

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

10th October 2010

Supervised by:
Valentin Wüstholz
Prof. Dr. Peter Müller

Chair of Programming Methodology
inf Informatik Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Contents

1 Introduction

1.1 Overview

This *Research in Computer Science* project is part of an effort to develop an automatic verification system (similar to Spec# [BLS04], ESC/Java [Fla+02] or ESC/Java2 [SC06] and the Java Modelling Language JML [LBR99]) for the Scala [Ode09] programming language. Scala is a strongly typed multi-paradigm language developed at the Ecole Polytechnique Fédérale de Lausanne that runs on the Java Virtual Machine.

The contribution of this project is a specification and verification technique for Scala *traits*, under special consideration of *behavioural subtyping* [LW94]. Traits, also known as *mixins* (see the glossary of [OSV08] for a definition of the two terms in the context of Scala), have been introduced by Moon [Moo86] in 1986 and are seen as a solution to the *diamond* or *linearisation problem* [BC90; Duc+06] of multiple inheritance. So far, mixin-based inheritance is mainly available in untyped dynamic languages, but rarely as a native language feature (e.g. in Ruby). Scala is currently one of the few strongly typed languages that offer native support for traits.

We use Boogie [Bar+05], a static programme verifier developed at Microsoft Research, to verify our examples. Although the syntax of the specification language used in this report has been influenced by Boogie, the resulting proof obligations are nevertheless general in the sense that they are given in first-order logic.

Boogie performs a series of transformations from a source programme written in the Boogie Programming Language (BoogiePL, current version being Boogie 2 [Lei08]) and finally passes the resulting proof obligations to a theorem prover. BoogiePL is a procedural intermediate language that includes method specifications and that is used to verify several high-level languages such as Spec#, C (using VCC [Coh+09]), Dafny [Lei10a] or Chalice [Lei10b] by first encoding the source programme (and corresponding specifications) in BoogiePL and then having Boogie verify this encoding.

We expect the reader to be familiar with object-oriented programming concepts in general, especially inheritance and subtyping, as well as software verification concepts such as pre- and postconditions, specifications and contracts. Basic knowledge of the Scala language and Scala traits is also assumed.

1.2 Motivation

Our motivation to engage in Scala trait verification has been driven by the *stackable modification* example found in [OSV08, p. 222 ff.]. A derived example, which will be used as a running example throughout this report, is shown in Listing 1.1.

Listing 1.1: An example of stackable modifications

```
class IntCell {
  var x: Int = 0

  def get() = this.x

  def set(x: Int) {
    this.x = x
  }
}

trait Doubling extends IntCell {
  override def set(x: Int) {
    super.set(2 * x)
  }
}

trait Incrementing extends IntCell {
  override def set(x: Int) {
    super.set(x + 1)
  }
}

var cid: IntCell = IntCell with Incrementing with Doubling
  /* cid.get() is always odd (or zero) */

var cdi: IntCell = IntCell with Doubling with Incrementing
  /* cdi.get() is always even (or zero) */
```

The concept of stackable modifications gave rise to the following research questions, which have been addressed in this project:

Specification language How to specify methods – e.g. `Doubling.set` – that invoke super-methods, without knowing to which class or trait `super` will be bound by the time the method participates in a mixin?
This question is legitimate not only for traits, but also for classes such as `class A extends B` that invoke super-methods, since when used in a mixin composition, `super` can actually be bound to some trait `T` and not to `B`, as one might expect.

Specification linearisation How to verify that `cdi.get` will always return an even value after calling `cdi.set`, whereas `cid.get` will always return an odd value? Generally speaking, how to cope with the fact that traits can be mixed in different orders into different classes?

Behavioural subtyping How to guarantee a method such as `def m(c: IntCell)` that all cells passed to it show a certain behaviour, e.g. return odd or even values? Utilising Scala's type system is not an option, since it only incorporates a weak notion of behaviour, namely by structural and nominal subtyping. Neither is the straight-forward approach of working with the specifications of the `IntCell` only, since these specifications must be as general as possible in order to not restrict future mixin compositions.

How to enforce behavioural subtyping for classes (and traits), e.g. such that all subtypes of `class OddIntCell with Incrementing with Doubling` return odd values only?

Flexibility Always enforcing behavioural subtyping can significantly reduce the applicability and thereby the benefits of traits. A flexible trait verification technique should therefore grant developers the liberty of deciding themselves where and when to enforce behavioural subtyping.

With respect to our stackable modifications example, enforcing behavioural subtyping in general would imply that the traits `Doubling` and `Incrementing` must both be behavioural subtypes of `IntCell`. This implies that we either give a very weak – and thus possibly meaningless – specification to `IntCell` in order to grant more freedom to the inheriting traits, or that we give a meaningful specification to `IntCell`, thereby constraining the inheriting traits.

Modularity When do trait specifications generate proof obligations: once at declaration-time or every time they occur in a mixin? In case of the latter, is it necessary to inspect the trait body each time it occurs in a mixin, or is it sufficient to consider its specifications only?

1.3 Proceeding

The rest of this report is structured as follows: Section 1.4 lists several use-cases which have been assessed according to the challenges they might pose to verification; Section 2 introduces the elements of the specification language we developed in order to specify Scala traits; Section 3 contains various definitions necessary to finally formulate the proof obligations stated in Section 4. A specified and Boogie-encoded version of our running example is shown in the appendix in Listing 6.1.

1.4 Scala traits

1.4.1 Overview

In the current chapter we briefly describe properties and characteristics of Scala traits that are relevant to behavioural subtyping. More general introductions to Scala and Scala traits can be found in *A Tour of Scala*[1] or in [OSV08].

Inheritance Traits are declared as

$\texttt{trait}\ T\ \texttt{extends}\ B\ \texttt{with}\ T_1\ \texttt{with}\ \ldots\ \texttt{with}\ T_n$

where B is a class or trait and $T_1, \ldots T_n$, $n \geq 0$, are other traits. Just like Scala classes, traits can make super-calls, i.e. invoke the parent class methods they override.

[1] http://www.scala-lang.org/node/104

Traits are used in *mixin class compositions*

```
class A extends B with T_1 with ... with T_n
```

where B is a class or trait and $T_1, \ldots T_n$, $n \geq 0$, are traits.

Traits only have a single constructor taking no arguments. Moreover, that primary trait constructor cannot explicitly invoke its super-constructor.

Linearisation In contrast to Java's class inheritance, where the binding of `super` is already known when the class is declared, `super` is *late-bound* in Scala. By the time a class is instantiated Scala puts all inherited classes and traits into a linear order and binds `super` accordingly. This *linearisation* process is defined formally in [Ode09, p. 52 ff.]. Considering the declaration of `cid` from our running example, we have it that `super` inside of `Incrementing` is bound to `IntCell`, whereas it is bound to `Incrementing` inside of `Doubling`, although both traits are declared to inherit from `IntCell`.

We could thus regard `super` as a generic class argument that is bound not until the class is instantiated. This implies that methods making super-invocations can only be specified in terms of yet unknown super-methods.

Type system Each trait definition also defines a corresponding type, similar to class definitions. Trait mixins are reflected by the type system, but the order in which traits occur in a mixin composition is not taken into account. Hence, we cannot utilise Scala's type system to ensure behavioural subtyping, as shown in Listing 1.2.

Listing 1.2: Scala's type system ignores trait order

```
def m(c: IntCell with Incrementing with Doubling) {...}

m(cid)     // Both invocations
m(cdi)     // are permissible.
```

It is worthwhile to observe that Scala's `with`-keyword serves two related but nonetheless different purposes: declaring classes by mixin composition and declaring *intersection types* [Pie02]. Both applications are illustrated in Listing 1.3.

Listing 1.3: Mixin class composition define intersection types

```
/* Mixin composition */
class M extends Incrementing with Doubling

/* Argument c is of an intersection type */
def m1(c: Incrementing with Doubling) {...}
def m2(c: M) {...}

m1(new M)                                /* Valid */
m2(new M)                                /* Valid */
m1(new Incrementing with Doubling {})    /* Valid */
m2(new Incrementing with Doubling {})    /* Invalid */
```

1.4.2 Use-cases

While the stackable modifications use-case shown in Listing 1.1 has initially motivated us to study behavioural subtyping in the context of Scala traits, we nevertheless collected additional use-cases that may pose challenges to the verification of traits. This section presents several use-cases extracted from various sources, e.g. papers and blogs.

Mimicking Java language constructs

Interfaces Traits are used to mimic Java interfaces: they merely specify the signature of a class without providing any implementation and they cannot be instantiated. Examples of this use-case can be found in [Sca], e.g. `scala.actors.remote.Service` and in the package `scala.collection.interfaces`.

References: [Sue09]

Abstract classes Traits are used to mimic abstract classes in Java: they cannot be instantiated, but unlike interfaces they already contain some implementation. Examples of this use-case can be found in the Scala standard library, e.g. `scala.collection.generic.Sorted`.

References: [Sue09]

Stackable modifications

Several traits extending or restricting themselves to a common base class are stacked (or chained, cascaded) in different orders to yield classes that are composed of the same basic elements (the traits) but show different behaviour. See Listing 1.1 for an example of this use-case.

This is analogous to a filter pipeline where each filter in the pipeline performs a specific operation on its input and where each filter's output is the input to the next filter.

This use-case of traits also resembles the well-known decorator example from [Gam+94], with the important limitation that decorations can only be applied statically at compile-time.

References: [OSV08, p. 222 ff.]

Thin vs. rich interfaces

A *thin interface* has only a few methods and is thereby easier to implement for a service provider. A *rich interface* has a lot of convenience methods and is thereby easier to use for a client.

In Scala such a rich interface would be implemented as a trait that implements all the convenience methods in terms of only a few basic methods. A concrete class now only has to implement the thin interface. Afterwards, it can mixin the rich interface to offer a convenient API to its clients.

Listing 1.4: Declaring thin and rich interfaces

```
class Rational(n: Int, n: Int) extends Ordered[Rational] {
  ⋮

  def compare(that: Rational) =
      (this.num * that.den) - (that.num * this.den)
}

trait Ordered[A] {
  def compare(that: A): Int /* Abstract thin interface */

  /* Rich interface's convenience methods. */
```

```
  def <  (that: A): Boolean = (this compare that) <  0
  def >  (that: A): Boolean = (this compare that) >  0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
}
```

This distinction between thin and rich interfaces (on the implementation level) can also be achieved in Java by using an abstract class implementing all convenience methods, and a concrete class extending the abstract one. The concrete class then only has to implement the basic methods left unimplemented by the abstract class.

The Java approach, however, becomes problematic once we have multiple rich interfaces that are applicable to concrete classes, because we have to combine them in (many) new abstract base classes. This can result in duplicated code or incoherent base classes.

This is not an issue in Scala where a class can inherit from multiple traits.

Listing 1.5: Using rich interfaces

```
trait Ordered[A] {
  // ... Ordered trait as defined before ...
}

trait Iterable[A] {
  def next(): A

  def nextN(n: Int): List[A] = ...
  def everyNth(n: Int): List[A] = ...
  def transform(f: A => A): Iterable[A] = ...
}

class MyString extends Ordered[MyString] with Iterable[String] {
  var content: String = ""

  def compare(that: MyString): Int = ...
  def next(): A = ...
}
```

References: [OSV08, p. 216 ff.; OM09]

Multiple Inheritance, Aggregation

In this very general use-case, traits are used in a mixin class composition simply to aggregate functionality. That is, traits are only used because they enable a class to inherit from multiple other classes. In Java one would probably implement this by aggregation, i.e. by declaring a class with references to other classes providing the required services.

The traits participating in such a mixin composition are not tightly coupled with respect to inheritance relationships, super-calls or aggregation. That is, they do not show particular indicators hinting at the fact that they are finally combined in a mixin composition. In other terms, when studying the traits one by one on their own, it is not self-evident, let alone mandatory, that the traits are eventually combined.

Modules, Packages

A trait can be used in a similar way to a module or package, in the sense that the trait itself contains a coherent (as seen by the developer) set of functions, classes, objects and other traits. The outer trait can therefore be regarded as a module or package. The significant difference, however, is that such a package trait can participate in mixin compositions, just as any other trait can.

Grouping In the grouping use-case, several such package traits are finally mixed into one class that makes use of the thereby inherited functionalities, i.e. of the functions, classes, objects and traits inherited from the different grouping traits. In a sense, the grouping use-case is a generalisation of the multiple inheritance use-case: a rather arbitrary composition of functionality that is not tightly coupled.

Instances of the grouping use-case can be found in the Scala source code, e.g. `scala.tools.nsc.ast.parser.SyntaxAnalyzer` in the version of Scala 2.8.0 RC6.

Two-dimensional extending A special variant of the grouping use-case is the problem of extending a set of data types (first dimension) as well as a set of operations (second dimension) defined on the data types. When adding new data type variants and new operations it should be possible to independently combine the extensions without having to neither change nor duplicate existing code.

[ZO05] gives a detailed introduction into the problem and also suggest two solutions using Scala traits: the first is based on object-oriented decomposition (interpreter pattern), the second on functional decomposition (visitor pattern). Both solutions are equivalent in the sense that they allow extensions to be made and combined independently. However, each solution puts the focus on one dimension and makes it slightly more cumbersome to write extensions in the other dimension.

In contrast to the traits mixed into the `SyntaxAnalyzer`, the package traits used in [ZO05] are strongly coupled by inheritance relationships. This property could be used as a heuristic guiding the decision if a certain package trait is an instance of the very general grouping use-case, or if it represents a more specialised use-case.

1.4.3 Conclusion

After studying the previously listed use-cases we decided to focus on the stackable modifications use-case during our development of a verification technique for Scala traits, since it uses Scala's trait inheritance to full capacity, thereby pushing the boundaries of the technique we seek to develop. Moreover, we assumed that being able to verify instances of the stackable modifications use-case is a prerequisite for the verification of instances of more complex use-cases, such as the two-dimensional extending use-case.

2 Specification language

This chapter subsequently introduces the components of our verification technique, mainly the syntactic elements of the *specification language* we use to declare method pre- and postconditions and the concept of *specification linearisation*. The syntactic elements are presented as an extension of the Scala language specification[1], but hiding them inside comment blocks (as done in JML) might be more appropriate for an initial reference implementation. The proof obligations that must be discharged in order to verify such specifications will be presented in Section 4.

For the sake of simplicity, we presume that classes do not declare auxiliary constructors and thus only implement a primary constructor.

2.1 Method specifications

We specify Scala methods in terms of pre- and postconditions expressed in first-order logic, extended by the possibility of referencing other specifications by means of *specification references*, which are actually simplified *specification functions* as defined by [KM10].

The pre- and postconditions are declared as *requires* and *ensures clauses* directly following the method signature, as done in e.g. Boogie [Lei08]. Note that these clauses are not part of the Scala language, but our own extension. Postconditions support two additional keywords which may not occur in preconditions: `result`, denoting the return value of a method and `old(`e`)`, denoting the evaluation of an expression e in the *old heap*, i.e. the heap as it existed before the method invocation and in which the precondition held. `old(`e`)` can be written as just e if e contains only objects that are not modified during the method execution. The code snippet in Listing 2.1 illustrates the use of the constructs introduced so far.

[1]Which implies that the sources presented in this report will not compile as is

Listing 2.1: Basic method specifications

```
def m(x: Int, y: Int): Int =
  requires: y > 0
  ensures: ∃ r : Int • old(x) = result · sqrt(old(y)) + r ∧ 0 ≤ r ∧ r < sqrt(old(y))
{ x / sqrt(y) }
```

Considering the `Doubling` trait from our running example we quickly see that we cannot specify its `set()` method without referencing the specifications of the `super.set()` method, where `super` is yet unbound. We therefore adopt the concept of super-references to our specification language, as shown in Listing 2.2, and we use the term *abstract specification* to indicate that a specification contains super-references and *concrete specification* to indicate that it does not. A detailed description of the form specification references may have can be found in the appendix.

We also adopt the concept of this-references, where $this.m.pre$ in a specification denotes the composed precondition (i.e. the one resulting from a mixin composition, see Section 2.2) of a method `m`, just like `this.m()` invokes the composed method `m` as resulting from a mixin composition. Such a this-reference is used in Listing 2.2 to specify the constructor of the `IntCell` class in terms of its method `set`, which the constructor invokes. In case of class `M` the postcondition $this.set.post(x)$ is now linearised to $get() = 2x$. If we would also mix in trait `Incrementing`, it would be either $get() = 2x + 1$ or $get() = 2(x + 1)$, depending on the order of the traits.

Listing 2.2: Specifications with super- and this-references

```
class IntCell(x: Int)
  requires: this.set.pre(x)
  ensures:  this.set.post(x)
{
  this.set(x)

  ⋮

  def set(x: Int)
      ensures: get() = x
    {...}
}
```

```
trait Doubling extends IntCell {
  override def set(x: Int)
    requires: super.set.pre(2x)
    ensures:  super.set.post(2x)
  { super.set(2 * x) }
}
```

```
class M(x: Int) extends IntCell(x) with Doubling
```

The precondition of `IntCell.set` in Listing 2.2 has been omitted and thereby defaults to $true$, which is perfectly fine. The precondition of `IntCell`'s constructor, however, has not been omitted, since that would make the verification of the constructor fail. If it would have been omitted it would also default to $true$, which would be ok as long as the precondition of `set` is $true$. Since this cannot be guaranteed – a trait with a precondition other than $true$ might be mixed into the `IntCell` – it is mandatory for the constructor to reference `set`'s precondition.

2.2 Specification linearisation

The operational behaviour of a class resulting from a mixin composition, e.g.

```
class OddCell extends IntCell with Incrementing with Doubling
```

is determined by the composed classes and traits and their linearisation, as mentioned before. Similarly, we need to compose the corresponding specifications in order to determine a valid specification for the resulting class. Since the composition must be done with respect to a given class linearisation, we call it the *specification linearisation* algorithm.

The specification linearisation algorithm applied to a class M yields a so-called *contract* C_M. Contracts will be introduced in Section 2.3, they may be seen as Java interfaces equipped with method specifications. The specification linearisation algorithm performs textual substitution on both pre- and postconditions alike:

- for each method m and for each specification clause $m.s$ (in any order, i.e. there is no need to topologically sort methods or clauses according to their dependencies)
  - recursively replace all super-references inside $m.s$ by the actually referenced specifications, where $super$ is bound to the next class in the class linearisation chain, which is advanced by one at each recursive descent
  - substitute the actual arguments passed to the referenced specification function for the formal arguments of that specification function
  - bind all this-references to C_M

If the specification linearisation algorithm reaches a state where the class linearisation chain has already been traversed completely, i.e. the end of the chain has been reached, but where there still are super-references to resolve, it will terminate with an exception.

Listing 2.3: A mixin composition

```
class B {
  def m(x: Double): Double =
  { return x }
}

trait T2 extends B {
  def m(x: Double): Double =
    requires: x ≥ 0
              ∧ super.m.pre(sqrt(x))
    ensures:  super.m.post(sqrt(x))
  { return super.m(sqrt(x)) }
}

trait T1 extends B {
  def m(x: Double): Double =
    requires: super.m.pre(−x)
    ensures:  super.m.post(−x)
  { return super.m(-x) }
}

trait T3 extends B {
  def m(x: Double): Double =
    requires: x ≠ 0
              ∧ super.m.pre(1/x)
    ensures:  super.m.post(1/x)
  { return super.m(1 / x) }
}


class M1 extends B with T1 with T2 with T3
class M2 extends B with T3 with T1 with T2
class M3 extends B with T2 with T1 with T3
```

The following derivations correspond to the work of the specification linearisation algorithm when applied to the classes `M1`, `M2` and `M3` as declared in Listing 2.3. The triple-bar equality symbol as used in $\phi \equiv \rho$ indicates that ρ follows from ϕ in zero, one or more steps of the specification linearisation algorithm. Zero steps imply that the formula has been simplified or otherwise rewritten.

Class linearisation of `M1`[2]:

$$[M1,\ T3,\ T2,\ T1,\ B]$$

Specification linearisation:

$$\begin{aligned} &M1.m.pre(x) \\ &\quad \equiv T3.m.pre(x) \\ &\quad \equiv x \neq 0 \land T2.m.pre(1/x) \\ &\quad \equiv x \neq 0 \land 1/x \geq 0 \land T1.m.pre(sqrt(1/x)) \end{aligned}$$

[2]Omitting Scala's base classes such as `AnyRef` or `Any`

$$\equiv x \neq 0 \land 1/x \geq 0 \land B.m.pre(-sqrt(1/x))$$
$$\equiv x \neq 0 \land 1/x \geq 0 \land true$$
$$\equiv x \neq 0 \land 1/x \geq 0$$
$$\equiv x > 0$$

Class linearisation of `M2`:

$$[M2,\ T2,\ T1,\ T3,\ B]$$

Specification linearisation:

$$M2.m.pre(x)$$
$$\equiv T2.m.pre(x)$$
$$\equiv x \geq 0 \land T1.m.pre(sqrt(x))$$
$$\equiv x \geq 0 \land T3.m.pre(-sqrt(x))$$
$$\equiv x \geq 0 \land -sqrt(x) \neq 0 \land B.m.pre(1/-sqrt(x))$$
$$\equiv x \geq 0 \land -sqrt(x) \neq 0 \land true$$
$$\equiv x \geq 0 \land -sqrt(x) \neq 0$$
$$\equiv x > 0$$

Class linearisation of `M3`:

$$[M3,\ T3,\ T1,\ T2,\ B]$$

Specification linearisation:

$$M3.m.pre(x)$$
$$\equiv T3.m.pre(x)$$
$$\equiv x \neq 0 \land T1.m.pre(1/x)$$
$$\equiv x \neq 0 \land T2.m.pre(-(1/x))$$
$$\equiv x \neq 0 \land -(1/x) \geq 0 \land B.m.pre(sqrt(-(1/x)))$$
$$\equiv x \neq 0 \land -(1/x) \geq 0 \land true$$
$$\equiv x \neq 0 \land -(1/x) \geq 0$$
$$\equiv x < 0$$

The specification linearisation algorithm can also be used to detect cyclic references (e.g. $m.pre$ (transitively) references $g.pre$ which in turn (transitively) depends on $m.pre$) in the specifications by maintaining a *closed set* of fully-qualified names of the classes, traits and contracts visited while following specification references in the specifications of a method m. Since all references are either static or can at least be resolved statically with respect to a given class linearisation, such a closed set is already sufficient to detect cycles (and thereby to guarantee soundness). If a cycle is detected the linearisation algorithm terminates with a corresponding exception. Note that this simple approach prohibits recursive specifications, but note also that this is a limitation of the implementation presented in this report, not a general limitation of our technique (see also Section 6).

2.3 Contract declarations

For this section we assume a basic understanding of the so-called *refinement relation* between specifications, which will be introduced formally in Section 3.5. Intuitively, if a method specification $(m.pre, m.post)$ *refines* another method specification $(g.pre, g.post)$, invocations of m may be safely substituted for invocations of g without introducing errors to the overall programme. On the class level, if a class `A` refines a class `B`, or more precisely, if the specifications of `A` refine those of `B`, it is safe to substitute objects of type `A` for objects of type `B`. This property is called the *Liskov substitution principle* [LW94].

The specification constructs introduced so far enable developers to directly equip methods with abstract or concrete specifications. Directly in the sense that the specifications are part of the method declaration. Such specifications, however, are not yet sufficient to enforce behavioural subtyping:

1. In order to require some kind of behaviour we must be able to uniquely address a set of specifications, as in `def m(x: IntCell respects C) {...}`, where `C` is a set of specifications for a subset of the methods defined by `IntCell`.

2. Let `cell` hold an object of type `T`, where `T` is a subtype of `IntCell`. If we pass that cell to `m` we must ensure that it behaves according to the set `C` of method specifications, i.e. that it refines them. However, verifying this relation is impossible if the specifications of class `T` or those in `C` are abstract, since the super-references are still unbound.

3. As stated initially, it might not always be desirable to enforce behavioural subtyping. Hence, we need a mechanism to explicitly declare for which classes behavioural subtyping is to be enforced.

We therefore extend our specification technique by a concept we call *contracts*, which is undeniably related to, but not to be confused with contracts as defined by [Mey97]. Our contracts are uniquely named entities consisting of method signatures and corresponding pre- and postconditions only (i.e. no method bodies), somewhat analogous to Java interfaces. Contract method signatures are just like regular class method signatures, they thus have to be unique with respect to the pair ⟨method name, method type⟩.

Unlike the method specifications introduced so far, contracts may only contain concrete specifications, i.e. no super-references. Hence, it is always possible to check for two given contracts if one is a refinement of the other. Listing 2.4 shows two contracts suitable for our running example. The keyword `contract` beginning a contract declaration is not part of the Scala language, but our own extension.

In contrast to method specifications, where this-references are essential in order to reference the specifications as resulting from a mixin composition (see Section 2.2), this-references inside contracts are only syntactic sugar. A this-reference such as $this.m.pre$ inside a contract C can simply be replaced by $C.m.pre$[3] because contract specifications always have to be concrete.

Listing 2.4: Contracts suitable for `IntCells`

```
contract OddGetter {
  def set(x: Int)
    ensures: get() mod 2 = 1
}
```

```
contract EvenGetter {
  def set(x: Int)
    ensures: get() mod 2 = 0
}
```

2.4 Contract annotations

Contracts can now be used to require a certain behaviour from objects passed to methods and also to declare that behavioural subtyping is to be enforced for certain classes (or traits). Listing 2.5 shows such *contract annotations* in the context of our running example. The keyword `respects` declaring a contract annotation is not part of the Scala language, but our own extension.

Listing 2.5: Annotating `IntCell` with a contract

```
class OddCell extends IntCell with Incrementing with Doubling
              respects OddGetter

def m(c: IntCell respects OddGetter) {...}

m(new OddCell) /* Valid */
```

[3]This-references inside contracts can be convenient if the contract name is long or if the contract is renamed often, but they do not increase the expressiveness

Specifications are linearised (see Section 2.2) only when a class is declared to respect a contract `C`, since we then expect the linearised specification to be concrete[4] in order to be able to verify that the linearised specification refines `C`. This implies that a trait declaration containing a respects-clause will be valid only if the trait specification contains no super-references[5].
The linearised specification is discarded once the refinement relation has been verified and only the contracts that a class has been annotated with are used for further verifications.

By definition, annotating a class `B` with a contract `C` enforces behavioural subtyping for all methods specified by the contract, so that all subtypes of `B` must have methods that refine the methods contained in `C`. Listing 2.6 illustrates the enforcement of behavioural subtyping.

Listing 2.6: Behavioural subtyping for an `IntCell`

```
/* Let OddCell be declared as before. */

/* Valid, since GoodCell is a behavioural subtype of OddCell. */
class GoodCell extends OddCell {
  override def set(x: Int)
    requires: super.set.pre(-x)
    ensures:  super.set.post(-x)
  { super.set(-x) }
}

/* Invalid, since BadCell is not a behavioural subtype of OddCell,
 * i.e. BadCell does not respect contract OddGetter.
 */
class BadCell extends OddCell {
  override def set(x: Int)
    ensures: get() = x
  { this.x = x }
}
```

Multiple contracts $C_1, \ldots, C_n$ may be used conjointly to express that a class respects all of them, as shown in Listing 2.7. The example also demonstrates that the contracts do not override each others specifications.

[4] At least for the methods specified by `C`
[5] Again, only for the methods specified by that contract

Listing 2.7: Annotating `IntCells` with multiple contracts

```
val cell: IntCell respects OddGetter, PositiveGetter = ...

def m(c: IntCell respects OddGetter) {...}
def g(c: IntCell respects PositiveGetter) {...}

m(cell) /* Valid */
g(cell) /* Valid */

m(new IntCell with Incrementing with Doubling)    /* Invalid*
```

Contracts can also declare constructors and fields, just like any regular Scala class. When annotating a class or trait with a contract the verifier has to check that the annotated class declares at least the fields declared by the contract. An example of a contract specifying a constructor can be found in the appendix in Listing 6.3.

Given method `m` as defined in Listing 2.7, it would be possible to allow an invocation such as

```
m(new IntCell with Incrementing with Doubling)
```

by having the system automatically verify that the resulting object respects the required contract `OddGetter`. Such an automatisation would result in less code by relieving developers from the need of explicitly declaring respects-clauses. However, this might result in programmes that verify although the developer did not want them to verify. We therefore require developers to always explicitly declare respects-clauses (and also refines-clauses introduced in Section 2.6).

Contracts may appear in specification references, as illustrated in Listing 2.8.

Listing 2.8: Referencing contracts

```
contract PositiveSetter {
  def set(x: Int)
    requires: x > 0
}

def m(var cell: IntCell respects PositiveSetter, x: Int)
  requires: PositiveSetter.set.pre(x)
{ cell.set(x) }
```

2.5 Contract casts

When working with third-party libraries situations like the one illustrated in Listing 2.9 can occur, where the developer is not able to modify the declarations of classes from that library in order to add certain respects-clauses.

Listing 2.9: 3rd-party libraries and contracts

```
/* Let ExternalClass and ECFactory be declared by a third-party
 * library.
 * Let contract MyContract be defined in the client code.
 */

def m(var ec: ExternalClass respects MyContract) { ... }

val factory = new ECFactory()

val ec: ExternalClass = factory.create()

m(ec) /* Invalid, since ExternalClass is not declared to
       * respect MyContract.
       */

val ec2: ExternalClass respects MyContract = factory.create()
  /* Invalid for the same reason. */
```

To be able to cope with such situations we extend Scala with *contract casts*, which enable developers to declare that a given object is of a type respecting a certain contract. This is demonstrated in Listing 2.10.

Listing 2.10: Using contract casts

```
/* Let ExternalClass, ECFactory and MyContract be declared
 * as before
 */

def m(var ec: ExternalClass respects MyContract) { ... }

val factory = new ECFactory()

val ec: ExternalClass respects MyContract
    = (MyContract) factory.create()   /* Valid */
```

```
m(ec)                              /* Valid */
m((MyContract) factory.create())   /* Valid */
```

Unlike type casts[6], whose validity must be checked during run-time, contract casts only consider statically declared types and thus can be checked at compile-time by a static verifier.

2.6 Refinement declarations

Considering Listing 2.5 again, we might want to pass objects of a class

```
class MyCell extends IntCell with MyTrait respects MyContract
```

to `m` (as declared in Listing 2.7), where contract `MyContract` is a refinement of contract `OddGetter` required by `m`. In order to address such situations developers can explicitly declare refinement relations between contracts by means of *refinement declarations*. Once established, these declarations enable the system to immediately decide by a simple look-up whether or not two objects (i.e. their types) are compatible in the sense of the refinement relation. Listing 2.11 illustrates the use of refinement relations. The keyword `refines` denoting refinement declarations is not part of the Scala language, but our own extension.

Listing 2.11: Declaring refinement relations I

```
contract MyGetter refines OddGetter {
  def set(x: Int)
    ensures: get() mod 2 == 1 ∧ get() > 77
}

val cell: IntCell respects MyGetter = ...

/* cid and m as before */
m(cid)  /* Valid */
m(cell) /* Valid */
```

A declaration such as `contract C refines D` is only valid if the set of methods declared by contract `D` is a subset of the set of methods declared by contract `C`.

[6]Type down-casts, that is

In general, it is not sufficient to only support refinement relation declarations when declaring a new contract, since developers might want to establish a refinement relation between contracts that are declared in external libraries. Developers may use the `contract` keyword without a contract body, as shown in Listing 2.12, to establish a refinement relation between two (or more) classes under such circumstances.

Listing 2.12: Declaring refinement relations II

```
/* Let C, D, E and F be contracts defined in an external library. */

/* Declares that the already declared contract C refines the
 * contracts D and E.
 */
contract C refines D, E

/* Fails, since C is already declared. */
contract C refines F {
  ⋮
}
```

3 Definitions

This chapter introduces the toolset necessary to formalise the proof obligations generated by the Scala language extensions we previously established:

- several utility functions that are needed to formalise the proof obligations
- specification functions, corresponding to the specifications given to methods in terms of pre- and postconditions
- the refinement relation between contracts

3.1 Types

We use the following types to state the signatures of the functions defined in following sections:

- $Class$, $Trait$ and $Object$, denoting Scala classes, traits and objects, respectively
- $Type$, denoting Scala types; types are of form T or $T_1 \times \ldots \times T_n \rightarrow TR$
- $Heap$, denoting programme heaps
- $Bool$, denoting Boolean values
- $Ident$, denoting identifier names
- $Contract$, denoting both explicitly declared and linearised contracts
- $Method$, denoting Scala methods; every method has a type T: $Type$, a name N: $Ident$ and a declaring contract C: $Contract$

Moreover, we use $set[T]$ and $list[T]$ to denote a set or a list, respectively, containing elements of type T.

We do not yet address the framing problem, but note that our verification technique is independent of a particular framing technique. Consequently, we leave our heap model abstract for as long as possible, until it is eventually necessary to refine it (e.g. as done in the examples Listing 6.2 and Listing 6.4 in the appendix).

3.2 Utility functions

This sections introduces several utility functions that are necessary in order to formulate the proof obligations generated by our trait verification technique. We usually proceed by defining a function over single items, e.g. contracts, and then take the liberty to generalise (to overload, if one likes) the function to also accept sets of the same type as the single items.

To indicate that the argument to a function $f(x)$ can be of one out of several types, e.g. of type $Class$ or of type $Trait$, we make use of *union types* [Pie02], by declaring $f\colon Class \sqcup Trait$.

We write $T_1 <: T_2$ to denote that T_1 is a subtype of T_2.

We define

$$name\colon Method \rightarrow Ident$$
$$contract\colon Method \rightarrow Contract$$

as the name and the declaring contract, respectively, of a method, and

$$type\colon Method \sqcup Object \rightarrow Type$$

as the type of a method or of an object, where the type is the static type in the latter case.

We define

$$meth\colon Contract \sqcup Class \sqcup Trait \rightarrow set[Method]$$

as the set of methods (including the constructors) declared by a contract, class or trait, and we generalise it to

$$meth\colon set[Contract \sqcup Class \sqcup Trait] \to set[Method]$$
$$meth(G_1,\ \dots,\ G_n) \Leftrightarrow meth(G_1) \cup \dots \cup meth(G_n)$$

for multiple contracts, classes and traits.

Analogously, we define (and generalise)

$$const\colon Contract \sqcup Class \sqcup Trait \to Method$$

as the constructor declared by a contract, class or trait.

We define

$$matches\colon Method \times Method \to Bool$$
$$matches(m,\ m') \Leftrightarrow name(m) = name(m') \wedge type(m) = type(m')$$

to hold whenever two given methods m and m' have the same name and signature. Observe that m' is always unique, since contracts may not declare more than one method with the same name and signature.

We define

$$classlin\colon Class \sqcup Trait \to list[Class \sqcup Trait]$$

as the linearisation of a class or trait. Consult [Ode09, p. 52 ff.] for a definition of Scala's class linearisation algorithm.

We define

$$speclin\colon Class \sqcup Trait \to Contract$$

as the linearisation of the specifications of a class or trait M with respect to M's class linearisation, as described in Section 2.2. The specification linearisation function depends on the class linearisation function, as apparent from the referenced description.

Finally, we define

$$con\colon Class \sqcup Trait \to set[Contract]$$

as the set of contracts that a given class or trait respects, and we generalise it to

$$con\colon set[Class \sqcup Trait] \to set[Contract].$$

The set returned by $con(M)$ for a class or trait M is transitively closed with respect to the inheritance relation, i.e. the set contains all contracts that M or one of M's super-classes (or super-traits) has been annotated with. See Listing 3.1 for an example.

Listing 3.1: Respected contracts of a class

```
/* Let C, D, E and F be contracts defined in an external library. */

contract G refines F {...}

class B respects C, D {...}
trait T1 respects G {...}
trait T2 extends T1 {...}
class M extends B with T2 respects E

/* con(M) = {C, D, G, E}
 * Note that F is not part of con(M)!
 */
```

3.3 Specification functions

Following [KM10], we associate a *specification function* with each method specification clause, i.e. with each pre- and postcondition stated as `requires` and `ensures` clauses. Specification functions are global functions (one function per method type) determining whether the corresponding condition holds in a given state.

A precondition of any n-ary (possibly nullary) method of type S is an $(n+3)$-ary boolean function

$$pre_S\colon Method \times Heap \times TT \times T_1 \times \ldots \times T_n \rightarrow Bool$$

where $Heap$ represents the current heap, TT is the type of the receiver object (the this-object) and where the T_i's are the types of the n method arguments.
Recall, that we defined objects of type $Method$ such that they contain the method name, the declaring contract and the method type.

Similarly, a postcondition of any n-ary method is an $(n+5)$-ary boolean function

$$post_S\colon Method \times Heap \times Heap \times TT \times T_1 \times \ldots \times T_n \times TR \rightarrow Bool$$

where $Heap \times Heap$ represents the old heap (in which the precondition held) and the current heap, i.e. the heap resulting from the method invocation, and where TR is the method's return type. If the method does not return anything, its return type will be `Unit` and we assume that `()` is the only value of type `Unit`.

For the sake of readability we will use specification functions as $m.pre(\ldots)$ instead of $pre_{type(m)}(m, \ldots)$, where m is a method, and likewise for postconditions.

3.4 Special contracts

Let $C^{\top}$ be a contract that declares no methods. According to our definition of contract refinement (Section 3.5), $C^{\top}$ is the *top contract*, i.e. the contract that does not refine any other contract, but itself is refined by all other contracts.

Let $C^{\perp}$ be a contract such that

$$\begin{array}{l} \forall\, m\colon Method\ \exists\, m' \in meth(C^{\perp}) \bullet \\ \quad \forall\, p_1\colon T_1,\ \ldots,\ p_n\colon T_n,\ result\colon TR \bullet \\ \qquad matches(m,\ m') \wedge m'.pre(p_1,\ \ldots,\ p_n) = true \\ \qquad \wedge\, m'.post(p_1,\ \ldots,\ p_n,\ result) = false \end{array}$$

$C^{\perp}$ is the *bottom contract*, i.e. the contract that is not refined by any other contract, but itself refines all other contracts.

Given a method m declared by some contract D, i.e. $m \in meth(D)$, we define the notation $C.m$ such that

$$(C.m \in meth(C) \wedge matches(m,\ C.m)) \vee C.m = C^{\perp}.m$$

That is, $C.m$ either denotes a method m' that is declared by a contract C and that matches m or it denotes a corresponding method declared by the bottom contract. Thus, $C.m$ is always well-defined for any contract C and any method m.

An obvious optimisation for an implementation of our verification technique would be to not generate proof obligations corresponding to $m \sqsubseteq C^{\perp}.m$, but rather to abort immediately, since the refinement relation never holds anyway.

3.5 Refinement relation

The refinement relation that has been introduced intuitively in Section 2.3 is formally defined over methods as

$$\sqsubseteq : Method \times Method \rightarrow Bool$$

$$\begin{aligned} & m \sqsubseteq m' \Leftrightarrow \\ & \quad \forall\, h,\ h' : Heap,\ obj : TT,\ p_1 : T_1,\ \ldots,\ p_n : T_n,\ r : TR \bullet \\ & \qquad (m'.pre(h,\ obj,\ p_1,\ \ldots,\ p_n) \Rightarrow m.pre(h,\ obj,\ p_1,\ \ldots,\ p_n)) \\ & \qquad\quad \wedge \\ & \qquad (m'.pre(h,\ obj,\ p_1,\ \ldots,\ p_n) \\ & \qquad\quad \Rightarrow (m.post(h,\ h',\ obj,\ p_1,\ \ldots,\ p_n,\ r) \Rightarrow m'.post(h,\ h',\ obj,\ p_1,\ \ldots,\ p_n,\ r))) \end{aligned}$$

where

- m and m' are methods such that $type(m) = type(m')$
- obj is a receiver object
- h and h' are the old and the current heap, respectively
- $T_1, \ldots, T_n$ are the types of the n (possibly zero) arguments of m and m'
- TR is the return type of m and m'

Consequently, we generalise the refinement relation to range over contracts as a whole and then to range over sets of contracts:

$$\begin{aligned} & \sqsubseteq : Contract \times Contract \rightarrow Bool \\ & C \sqsubseteq D \Leftrightarrow \forall\, m \in meth(D) \bullet C.m \sqsubseteq D.m \end{aligned}$$

$$\begin{aligned} & \sqsubseteq : Contract \times set[Contract] \rightarrow Bool \\ & C \sqsubseteq D_1,\ \ldots,\ D_n \Leftrightarrow \bigwedge_{i=1}^{n} C \sqsubseteq D_i \end{aligned}$$

Note that the above definition of the refinement relation corresponds to the proof obligations generated by a refinement declaration such as `contract C refines D`. Thus, the `refines` keyword is the syntactic equivalent of the relation $C \sqsubseteq D$.

4 Proof obligations

This chapter presents all proof obligations arising from our verification technique. Since the majority of proof obligations corresponds directly to the verification of refinement relations as presented in Section 3.5, being familiar with the relation is crucial for the understanding of this section.

We will not consider well-known verification aspects that are not affected by traits and class mixin compositions, e.g. verifying that the precondition of a method m holds at a call-site.

4.1 Method specifications

Let m be a Scala method (or constructor) with n parameters and a method body $body_m$, a precondition P and a postcondition Q. This corresponds to a Hoare-style [Hoa69] triple $\{P\}\ body_m\ \{Q\}$ which must be proven to hold.

Considering our verification framework, we thus have to show that

$$wp(body_m,\ m.post) \Rightarrow m.pre$$

where $m.pre$ and $m.post$ are the specification functions corresponding to P and Q, respectively, and where $wp(\ldots)$ represents Hoare's weakest precondition.
This is the only proof obligation requiring access to the method body. Once the proof obligation is discharged we only need the pre- and postconditions to verify any future use of method m.

4.2 Refinement declarations

Proof obligation 1: A mere contract declaration such as `contract` C `{...}` does not generate proof obligations, since there is nothing to verify yet. Refinement declarations such as `contract` C `refines` D, on the other hand, generate proof obligations, since we must verify that the declared refinement relation actually holds. The generated proof obligations directly correspond to the verification of $C \sqsubseteq D$.

4.3 Class annotations

Let B be a Scala class, let $T_1, \ldots, T_n$ be Scala traits and let C be a contract. We create a new contract-annotated class M by declaring

`class` $M(\vec{p})$ `extends` $B(f(\vec{p}))$ `with` T_1 `with` ... `with` T_n `respects` C `{...}`

where p is the formal argument vector (possibly of length zero) and f is a pure function returning a vector.

4.3.1 Refining an annotated contract

Let us at first consider the relation $speclin(M) \sqsubseteq C$ only, i.e. the proof obligations that must be discharged in order to verify that the linearised contract of M refines the contract C that M is annotated with.

Methods

Proof obligation 2: The proof obligations arising from the methods (excluding the constructors!) are all of the form

$$\forall\, m \in meth(C) \setminus const(C) \bullet speclin(M).m \sqsubseteq C.m$$

i.e. each method declared by a contract C generates a proof obligation directly corresponding to the refinement relation.

Constructors

The proof obligations generated by a constructor specification are more intricate, since Scala subsequently (following the class linearisation order) invokes all constructors participating in the mixin.

To be able to follow the proof obligation presented below, it is crucial to know that primary constructors in Scala cannot call a super-constructor directly, e.g.

```
class A(x: Int) extends B
  { super.this(-x) /*NOT valid Scala code */}
```

because that call is already made "inside" the class declaration, e.g.

```
class A(x: Int) extends B(-x)
```

Valid constructor specifications therefore do not contain super-references, but only contract references and this-references. For class M as defined in Section 4.3, let

- $L = classlin(M)$ be the class linearisation of M (of length n)
- L_i be the i-th element of L
- $C_M = speclin(M)$ be the specification linearisation of M
- S_i be the specification of L_i where each this-references is bound to C_M

Proof obligation 3: To ensure that the invocation sequence conforms to the constructor specifications of C, the following proof obligation has to be discharged:

$$\begin{aligned}
&\forall\, h,\ h'\colon Heap,\ t\colon TT,\ \vec{p}\colon T_C \bullet C.pre(h,\ f_1(\vec{p}))\\
&\quad \Rightarrow S_1.pre(h,\ t,\ f_1(\vec{p})) \wedge\\
&\qquad (\forall\, h_1\colon Heap \bullet S_1.post(h,\ h_1,\ t,\ f_1(\vec{p}))\\
&\qquad\quad \Rightarrow S_2.pre(h_1,\ t,\ f_2(\vec{p})) \wedge\\
&\qquad\qquad (\forall\, h_2\colon Heap \bullet S_2.post(h_1,\ h_2,\ t,\ f_2(\vec{p}))\\
&\qquad\qquad\quad \Rightarrow S_3.pre(h_2,\ t,\ f_3(\vec{p})) \wedge\\
&\qquad\qquad\qquad (\forall\, h_3\colon Heap \bullet S_3.post(h_2,\ h_3,\ t,\ f_3(\vec{p}))\\
&\qquad\qquad\qquad\quad \Rightarrow S_4.pre(h_3,\ t,\ f_4(\vec{p})) \wedge\\
&\qquad\qquad\qquad\qquad \vdots\\
&\qquad\qquad\qquad\qquad\ \Rightarrow S_n.pre(h_{n-1},\ t,\ f_n(\vec{p})) \wedge\\
&\qquad\qquad\qquad\qquad\qquad (S_n.post(h_{n-1},\ h',\ t,\ f_n(\vec{p})) \Rightarrow C.post(h,\ h',\ t,\ f_1(\vec{p})))\ldots)))
\end{aligned}$$

where

- T_C is the type of the argument vector of C's constructor
- f_i are pure functions that return the arguments as passed to the i-th constructor (if a constructor does not take arguments, f_i returns nothing and e.g. $S_i.pre(h, f_i(\vec{p}))$ actually is just $S_i.pre(h)$)
- h_i are intermediate heaps

To illustrate how the constructor argument functions f_i look like, let us consider the declarations

```
class A1(x: Int, b: Boolean)
class A2(z: Int) extends A1(2*z, true)
trait T extends A2
class A3(y: Int, s: String) extends A2(-y) with T
```

resulting in a class linearisation of $classlin(A3) = [A3, T, A2, A1]$ and corresponding constructor argument functions such that

$$f_1(y,\ s) = (y,\ s)$$
$$f_2(y,\ s) = ()$$
$$f_3(y,\ s) = (-y)$$
$$f_4(y,\ s) = (-2y,\ true)$$

Observe that f_1 will always be the identity function and hence could be removed from the above formula, but we nevertheless keep it for the sake of uniformity.

It is important to note that the intricate constructor proof obligation presented above is always and exclusively generated when verifying that a linearised specification refines a given contract, i.e. when verifying that $speclin(M) \sqsubseteq C$. We will therefore continue to use the refinement relation $\sqsubseteq$ to express proof obligations, but the reader must be aware of the different forms of proof obligations generated for the verification of the refinement relation.

4.3.2 Refining inherited contracts

Proof obligation 4: Let us examine the declaration of M in Section 4.3 again (omitting constructor arguments, since they are of no interest here):

```
class M extends B with T1 with ... with Tn respects C
```

According to Scala's type system, we have it that $M <: B$, $M <: T_1$, ... and $M <: T_n$. If B and T_i have been annotated with contracts in order to enforce behavioural subtyping, we have to ensure that $speclin(M)$ is a behavioural subtype of each of them. Thus, we also have to verify that $speclin(M) \sqsubseteq con(B, T_1, \ldots, T_n)$.

4.4 Multiple class annotations

Proof obligation 5: Generalising the above declaration of M by annotating M with multiple contracts, as in

```
class M extends B with T1 with ... with Tn respects C1,...,Cm
```

is straight forward and generates the following proof obligations:

1. $speclin(M) \sqsubseteq \{C_1, \ldots, C_m\}$
2. $speclin(M) \sqsubseteq con(B, T_1, \ldots, T_n)$

It is interesting to observe that, if we consider contracts similar to types, a class such as M is of union type $B \sqcup T_1 \sqcup \ldots \sqcup T_n$ and of "union contract" $C_1 \sqcup \ldots \sqcup C_m$.

4.5 Trait annotations

Proof obligation 6: It is also possible to enforce behavioural subtyping for traits, e.g. by a declaration such as

```
trait T extends B with T1 with ... with Tn respects C
```

which generates the same proof obligations as in the case of the so far considered class M. As already mentioned in Section 2.2, T may not contain any super-references in specifications of methods that are also specified by C, since the class linearisation of T is undefined and the verification would therefore fail while trying to resolve the super-references.

4.6 Variable declarations

A variable declaration such as

```
var x: B with T respects C
```

does not generate proof obligations, since it is not necessary that the corresponding empty class declared as

```
class M_x extends B with T
```

already respects contract C, but rather, that an object obj that x eventually points to respects C.

4.7 Assignments

Let a: A, b: B be two variables, where $A <: B$. An assignment such as $b = a$ is only valid if the system has already verified each refinement relation occurring in

$$\forall\, C \in con(B)\; \exists\, C' \in con(A) \bullet C' \sqsubseteq C.$$

Such contributions to the system's refinement knowledge base are either made explicitly with respects- and refines-clauses or implicitly when verifying class declarations where a contract-annotated class is extended.

Hence, assignments do not create additional proof obligations that have to be discharged, but rather trigger look-ups during the verification of specification-enriched Scala code, in order to see whether the refinement relations of interest have already been verified. That is, the verifier needs to keep track of the refinement relations for which the corresponding proof obligations have been generated. See Section 2.4 for an explanation why we do not generate proof obligations for assignments on the fly, i.e. without an explicit request from a developer.

4.8 Methods

Method declarations

The verification of a method declaration such as

```
def m(x: B with T respects C₁): M respects C₂ {...}
```

has two aspects to consider: the formal argument type and the return type. However, analogously to variable declarations already considered in Section 4.6, both aspects do not create proof obligations.

Return statements

The verification of a return statement such as

```
def m(x: B with T respects C₁): M respects C₂ {
  ⋮
  return y
}
```

where $type(y) <: M$ is equivalent to the verification of

```
var result: M respects C₂ = y
```

and thus is already covered by Section 4.7, which deals with assignments.

Method invocations

A method invocation such as `m(x)` also does not generate proof obligations because the rules for assignments from Section 4.7 apply once again.

4.9 Contract casts

Proof obligation 7: The verification of a contract cast such as

```
val x: B respects C = (C) y
```

where $type(y) <: B$ and where the system does not already know[1] that

$$\exists\, C' \in con(type(y)) \bullet C' \sqsubseteq C$$

holds, generates proof obligations corresponding to the verification of

$$speclin(type(y)) \sqsubseteq C.$$

[1]Which would render the cast redundant

5 Conclusion

We have introduced a specification and verification technique for Scala traits that has been designed with automated and responsive verification in mind. The proof obligations have been formalised in first-order logic and are generated in a conservative manner, i.e. triggered by explicit requests from developers.

Our main tools are specification references and specification functions enabling developers to specify methods in terms of other methods, especially in terms of yet unknown super-methods. The concepts of specification super-references and specification linearisations enable developers to compose different specifications to yield new specifications, just as Scala's class mixin composition yields new operations.

Our technique distinguishes between (possibly abstract) method specifications and concrete, type-invariant behaviour declared as contracts and enforced by behavioural subtyping rules. Regarding their behaviour, the former enables developers to leave classes and traits open as long as possible, whereas the latter enables them to close the behaviour if necessary.

We have manually encoded two stackable modification instances in Boogie and successfully verified them, thus showing that our technique can be applied to concrete instances of the stackable modifications use-case.

Reviewing our initial research questions from Section 1.2, we conclude that we were able to successfully address the question of a suitable specification language and that of a specification linearisation algorithm. Our technique incorporates behavioural subtyping but does not force developers to weigh meaningful specifications against implementational freedom.

The answer to the question of modularity is a mixed blessing: each method body has to be inspected only once, but the specification linearisation algorithm has to be executed for each mixin composition. This results in the need to reverify each mixin composition in which a given class or trait participates whenever the specifications of that class or trait have been changed.

6 Future work

We did not yet implement our verification technique, e.g. as an extension of the Scala compiler. An **implementation** would enable us to test our verification technique with more complex specifications and to challenge the interaction with the underlying theorem prover, i.e. Boogie.

For the sake of simplicity we did not yet consider **invariants** and **information hiding** (or **data abstraction**) (for example by means of abstraction functions [KM10], but we regard both as orthogonal extensions of our technique.

Auxiliary constructors have also not yet been considered. Since Scala – unlike Java – requires all auxiliary constructors to (indirectly) invoke the primary constructor as their first statement, we assume that extending our technique with auxiliary constructors is a straight-forward technicality.

In order to specify the examples presented in this report we did not need to deal with the heap and hence we did not incorporate any **framing technique**, e.g. *dynamic frames* [Kas06] in our verification technique. Again, we assume that such an extension is orthogonal to our technique.

Another limitation of the current state of our technique – but not of the technique itself – is the strict prohibition of cyclic specifications, which inhibits **recursive specifications** (see Section 2.2). Once again, we assume that an extension with recursive specifications and recursion variants is an orthogonal matter.

A worthwhile extension of our current specification language would be to support **contract inheritance**. A contract `E` declared as `contract E extends C,D` would inherit and thereby reuse the specifications from the contracts `C` and `D`, but could also override them and add new specifications, provided that `E` stays a refinement of `C` and of `D`.

Another extension worth considering might be **contract combinations**. The developer could declare that the combination of several existing contracts refines another existing contract, as in `contract C ⊕ D refines E`.

Bibliography

[BLS04] Mike Barnett, Rustan Leino and Wolfram Schulte. 'The Spec# Programming System: An Overview'. In: *CASSIS*. Ed. by Gilles Barthe et al. Vol. 3362. Lecture Notes in Computer Science. Springer, 2004. ISBN: 3-540-24287-2.

[Fla+02] Cormac Flanagan et al. 'Extended static checking for Java'. In: *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. New York, NY, USA: ACM, 2002, pp. 234–245. ISBN: 1-58113-463-0.

[SC06] Aleksy Schubert and Jacek Chrzaszcz. 'ESC/Java2 as a Tool to Ensure Security in the Source Code of Java Applications'. In: *SET*. Ed. by Krzysztof Sacha. Vol. 227. IFIP. Springer Boston, 2006, pp. 337–348. ISBN: 978-0-387-39387-2.

[LBR99] Gary T. Leavens, Albert L. Baker and Clyde Ruby. 'JML: A notation for detailed design'. In: *Behavioral Specifications of Businesses and Systems*. Kluwer Academic Publishers, 1999, pp. 175–188.

[Ode09] Martin Odersky. *The Scala Language Specification: Version 2.7*. Mar. 2009. URL: http://www.scala-lang.org/docu/files/ScalaReference.pdf.

[LW94] Barbara H. Liskov and Jeanette M. Wing. 'A Behavioral Notion of Subtyping'. In: *ACM Transactions on Programming Languages and Systems* 16 (1994), pp. 1811–1841.

[OSV08] Martin Odersky, Lex Spoon and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. 1st. Artima Inc, Nov. 2008. ISBN: 0981531601.

[Moo86] David A. Moon. 'Object-oriented programming with flavors'. In: *SIGPLAN Not.* 21.11 (1986), pp. 1–8. ISSN: 0362-1340.

[BC90] Gilad Bracha and William R. Cook. 'Mixin-based Inheritance'. In: *OOPSLA/ECOOP*. 1990, pp. 303–311.

[Duc+06] Stéphane Ducasse et al. 'Traits: A mechanism for fine-grained reuse'. In: *ACM Trans. Program. Lang. Syst.* 28.2 (2006), pp. 331–388. ISSN: 0164-0925.

[Bar+05] Michael Barnett et al. 'Boogie: A Modular Reusable Verifier for Object-Oriented Programs'. In: *FMCO*. Ed. by Frank S. de Boer et al. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 364–387. ISBN: 3-540-36749-7.

[Lei08] K. Rustan M. Leino. *This is Boogie 2. (Draft)*. June 2008. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf.

[Coh+09] Ernie Cohen et al. 'VCC: A Practical System for Verifying Concurrent C'. In: *TPHOLs*. Ed. by Stefan Berghofer et al. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 23–42. ISBN: 978-3-642-03358-2.

[Lei10a] K. Rustan M. Leino. *Dafny: An Automatic Program Verifier for Functional Correctness*. Tech. rep. Microsoft Research, 2010. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml203.pdf.

[Lei10b] K. Rustan M. Leino. 'Verifying Concurrent Programs with Chalice'. In: *VMCAI*. Ed. by Gilles Barthe and Manuel V. Hermenegildo. Vol. 5944. Lecture Notes in Computer Science. Springer, 2010, p. 2. ISBN: 978-3-642-11318-5.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Feb. 2002, p. 645. ISBN: 0262162091.

[Sca] *Scala 2.8.0 RC Standard Library*. 17th June 2010. URL: http://www.scala-lang.org/archives/rc-api/index.html (visited on 29/06/2010).

[Sue09] Josh Suereth. *How should traits in Scala be used?* Feb. 2009. URL: http://suereth.blogspot.com/2009/02/how-should-traits-in-scala-be-used.html (visited on 17/06/2010).

[Gam+94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612.

[OM09] Martin Odersky and Adriaan Moors. 'Fighting bit Rot with Types (Experience Report: Scala Collections)'. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*. Ed. by Ravi Kannan and K Narayan Kumar. Vol. 4. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009, pp. 427–451. ISBN: 978-3-939897-13-2.

[ZO05] Matthias Zenger and Martin Odersky. 'Independently Extensible Solutions to the Expression Problem'. In: *Proc. FOOL 12*. Jan. 2005.

[KM10] I. T. Kassios and P. Müller. *Specification and Verification of Closures*. Tech. rep. Swiss Federal Institute of Technology Zurich, 2010. URL: http://people.inf.ethz.ch/lehnerh/pm/publications/getpdf.php?bibname=Own&id=KassiosMueller10.pdf.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. 2. Upper Saddle River, NJ: Prentice Hall, 1997. ISBN: 978-0-13-629155-8.

[Hoa69] C. A. R. Hoare. 'An axiomatic basis for computer programming'. In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[Kas06] Ioannis T. Kassios. 'Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions'. In: *FM*. Ed. by Jayadev Misra, Tobias Nipkow and Emil Sekerinski. Vol. 4085. Lecture Notes in Computer Science. Springer, 2006, pp. 268–283. ISBN: 3-540-37215-6.

[Wüs09] Valentin Wüstholz. *Encoding Scala Programs for the Boogie Verifier*. Tech. rep. Swiss Federal Institute of Technology Zurich, 2009. URL: http://www.pm.inf.ethz.ch/education/theses/student_docs/Valentin_Wuestholz/MA_report.pdf.

Appendix

Specification references

Specification references may be of the following form:

- (full-qualified) method specification references $p.n.m.s$, where
  - p is a Scala package, e.g. $Scala.utils$ or $ch.ethz.dinfk$
  - n is the name of a contract, e.g. $OddGetter$; p may be omitted if n can be resolved uniquely
  - m is the name of a method, e.g. set
  - s is either pre or $post$
- (full-qualified) constructor specification references $p.n.s$
- this-references $this.m.s$ and $this.s$, respectively
- super-references $super.m.s$ (but not $super.s$, since Scala does not support the invocation of super-constructors inside sub-constructors)

Examples

The examples presented in this section have been encoded manually, their linearised specifications have also been manually computed. They serve illustrative purposes only and do not comprise all aspects of our verification technique. Note especially, that they do not contain an encoding of the Scala method bodies in Boogie, which is necessary to initially verify and thereby establish the pre- and postconditions. A possible encoding of Scala programmes for the Boogie verifier can be found in [Wüs09].

The proof obligations – presented in Section 4 as closed formulae with nested quantifiers – are encoded in a sequence of `assert` and `assume` pairs. This in a sense more operational encoding makes it easier to analyse failing verifications: in the case of a closed formula encoding, Boogie's error messages contain only the number of the formula's first line, whereas they contain the exact line number of the failing assertion in the case of the sequential encoding.

Both examples have been verified without errors or warnings using Boogie 2.0 (with the `/smoke` option) and Z3 2.4.

The first example encoded in Boogie is our running example.

Listing 6.1: `IntCells` with specifications

```
class IntCell {
  var v: Int = 0

  def get() = this.v /* A pure function */

  def set(x: Int)
    /* ensures: get() = x */
  { this.v = x }
}

trait Doubling extends IntCell {
  override def set(x: Int)
    /* ensures: super.set.post(2x) */
  { super.set(2 * x) }
}

trait Incrementing extends IntCell {
  override def set(x: Int)
    /* ensures: super.set.post(x + 1) */
  { super.set(x + 1) }
}

/*
  contract OddGetter {
    def set(x)
      ensures: get() mod 2 == 1
  }

  contract EvenGetter {
```

```
    def set(x)
      ensures: get() mod 2 == 0
  }
*/

class OddCell extends IntCell
              with Incrementing with Doubling
              /* respects OddGetter */

class EvenCell extends IntCell
               with Doubling with Incrementing
               /* respects EvenGetter */
```

Listing 6.2: Boogie encoding of the `IntCells`

```
/*
 * Preamble
 */

/* The modulo operation */
axiom (forall x: int, y: int :: {x % y} {x / y}
  x % y == x - (x / y) * y
);

axiom (forall x: int, y: int :: {x % y}
  (0 < y ==> 0 <= x % y && x % y < y) &&
  (y < 0 ==> y < x % y && x % y <= 0)
);

/* Heap model */
type IntCell;
type Field _;
type Heap = <beta>[IntCell, Field beta] beta;

/*
 * IntCell's fields and pure functions
 */

const v: Field int; /* Field IntCell.v */

/* Pure function IntCell.get */
function get(h: Heap, this: IntCell) returns (int);
```

```
axiom (forall h: Heap, this: IntCell :: { get(h, this) }
  get(h, this) == h[this, v]
);

/*
 * Contracts
 */

/* Contract OddGetter */
function OddGetter.set.pre(h: Heap, this:IntCell, x: int)
    returns (bool) { true }

function OddGetter.set.post(oh: Heap, h: Heap, this:IntCell, x: int)
    returns (bool) {
  get(h, this) % 2 == 1
}

/* Contract EvenGetter */
function EvenGetter.set.pre(h: Heap, this:IntCell, x: int)
    returns (bool) { true }

function EvenGetter.set.post(oh: Heap, h: Heap, this:IntCell, x: int)
    returns (bool) {
  get(h, this) % 2 == 0
}

/*
 * Linearised specifications
 */

/* Linearised specifications of the OddCell */
function OddCell.set.pre(h: Heap, this:IntCell, x: int)
    returns (bool) { true }

function OddCell.set.post(oh: Heap, h: Heap, this:IntCell, x: int)
    returns (bool) {
  get(h, this) == 2 * x + 1
}

/* Linearised specifications of the EvenCell */
function EvenCell.set.pre(h: Heap, this:IntCell, x: int)
```

```
    returns (bool) { true }

function EvenCell.set.post(oh: Heap, h: Heap, this:IntCell, x: int)
    returns (bool) {
  get(h, this) == 2 * (x + 1)
}

/*
 * Refinement verifications
 */

/* Verify that the linearised specifications of the OddCell refine
 * contract OddGetter
 */
procedure OddCell_set_refines_contract_OddGetter() {
  var oh, h: Heap;
  var x: int;
  var cell: IntCell;

  /* OddGetter.set.pre ⇒ OddCell.set.pre */
  havoc oh, x, cell;
  assume OddGetter.set.pre(oh, cell, x);
  assert OddCell.set.pre(oh, cell, x);

  /* OddCell.set.post ⇒ OddGetter.set.post -- Requires the
   * previous steps
   */
  havoc h;
  assume OddCell.set.post(oh, h, cell, x);
  assert OddGetter.set.post(oh, h, cell, x);
}

/* Verify that the linearised specifications of the EvenCell refine
 * contract EvenGetter
 */
procedure EvenCell_set_refines_contract_EvenGetter() {
  var oh, h: Heap;
  var x: int;
  var cell: IntCell;

  /* EvenGetter.set.pre ⇒ EvenCell.set.pre */
  havoc oh, x, cell;
```

```
  assume EvenGetter.set.pre(oh, cell, x);
  assert EvenCell.set.pre(oh, cell, x);

  /* EvenCell.set.post ⇒ EvenGetter.set.post -- Requires the
   * previous steps */
  havoc h;
  assume EvenCell.set.post(oh, h, cell, x);
  assert EvenGetter.set.post(oh, h, cell, x);
}
```

The second example is a modified version of the *IntCell* illustrating the use of constructor specifications. It is purely artificial and rather short, but it is already non-trivial to linearise and verify the specifications manually.

Listing 6.3: `AddCell` with specifications

```
class Store(z: Int)
  /* requires: this.add.pre(z)
   * ensures:  this.add.post(z)
   */
{
  var x: Int = 0
  add(z)

  def add(y: Int)
    /* ensures:  x = old(x) + y */
  { x += y }
}

trait Sqrt extends Store
  /* requires: x ≥ 0 this.add.pre(sqrt(x))
   * ensures:  this.add.post(sqrt(x))
   */
{
  add(sqrt(x))

  override def add(y: Int)
    /* requires: y ≥ 0 ∧ super.add.pre(sqrt(y))
     * ensures:  super.add.post(sqrt(y))
     */
  { super.add(sqrt(y)) }
}
```

```
trait Neg extends Store {
    /* requires: this.add.pre(-x)
     * ensures:  this.add.post(-x)
     */
  add(-x)

  override def add(y: Int)
    /* requires: super.add.pre(-y)
     * ensures:  super.add.post(-y)
     */
  { super.add(-y) }
}

/*
  contract ContractZero(z: Int) {
    requires: z = 0
    ensures:  x = 0 // Annotated class must have such a field

    def add(y: Int)
      requires: y ≤ 0
      ensures:  old(x) ≤ x
  }
*/

var ssn = new Store(0) with Sqrt with neg /* respects ContractZero */
```

Listing 6.4: Boogie encoding of the `AddCell`

```
/*
 * Preamble
 */

/* Heap model */
type AddCell;
type Field _;
type Heap = <beta>[AddCell, Field beta] beta;

/* A discrete root function (rounding down) */
function sqrt(v: int) returns (int);

axiom (forall v: int :: { sqrt(v) }
```

```
  v >= 0
    ==>
  sqrt(v) >= 0 && sqrt(v)*sqrt(v) <= v && v < (sqrt(v)+1)*(sqrt(v)+1)
);

/*
 * IntCell's fields and pure functions
 */

const x: Field int;

/*
 * Contracts
 */

/* Contract ContractZero */
function ContractZero.pre(h: Heap, this: AddCell, z: int)
    returns (bool) {
  z == 0 // Everything else should fail
}

function ContractZero.post(oh: Heap, h: Heap, this: AddCell, z: int)
    returns (bool) {
  h[this, x] == 0 // Everything else should fail
}

function ContractZero.add.pre(h: Heap, this: AddCell, y: int)
    returns (bool) {
  y <= 0
}

function ContractZero.add.post(oh: Heap, h: Heap, this: AddCell,
    y: int) returns (bool) {
  oh[this, x] <= h[this, x]
}

/*
 * Linearised specifications
 */

function ssn.add.pre(h: Heap, this: AddCell, y: int)
    returns (bool) {
```

```
  -y >= 0
}

function ssn.add.post(oh: Heap, h: Heap, this: AddCell, y: int)
    returns (bool) {
  h[this, x] == oh[this, x] + sqrt(-y)
}

/*
 * Constructors
 */

function Store.pre(h: Heap, this: AddCell, z: int)
    returns (bool) {
  ssn.add.pre(h[this, x := 0], this, z)

  /* h[this, x := 0] denotes that the initial value of the field
   * 'x' is zero.
   */
}

function Store.post(oh: Heap, h: Heap, this: AddCell, z: int)
    returns (bool) {
  ssn.add.post(oh[this, x := 0], h, this, z)
}

function Sqrt.pre(h: Heap, this: AddCell) returns (bool) {
  h[this, x] >= 0 && ssn.add.pre(h, this, sqrt(h[this, x]))
}

function Sqrt.post(oh: Heap, h: Heap, this: AddCell) returns (bool) {
  ssn.add.post(oh, h, this, sqrt(oh[this, x]))
}

function Neg.pre(h: Heap, this: AddCell) returns (bool) {
  ssn.add.pre(h, this, -h[this, x])
}

function Neg.post(oh: Heap, h: Heap, this: AddCell) returns (bool) {
  ssn.add.post(oh, h, this, -oh[this, x])
}
```

```
/*
 * Method refinement verifications
 */

procedure ssn_add_refines_ContractZero() {
  var oh, h: Heap;
  var y: int;
  var cell: AddCell;

  /* ContractZero.add.pre ⇒ ssn.add.pre */
  havoc oh, y, cell;
  assume ContractZero.add.pre(oh, cell, y);
  assert ssn.add.pre(oh, cell, y);

  /* ssn.add.post ⇒ ContractZero.add.post -- Requires the previous
   * steps
   */
  havoc h;
  assume ssn.add.post(oh, h, cell, y);
  assert ContractZero.add.post(oh, h, cell, y);
}

procedure ssn_add_refines_ContractZero_alternative() {
  /* Equivalent closed-formula encoding, presented for illustrative
   * purposes only.
   */
  assert (
    forall oh: Heap, cell: AddCell, y: int ::
      ContractZero.add.pre(oh, cell, y)
        ==> ssn.add.pre(oh, cell, y) &&
          (forall h: Heap ::
            ssn.add.post(oh, h, cell, y)
              ==> ContractZero.add.post(oh, h, cell, y)
          )
  );
}

/*
 * Constructor refinement verifications
 */

procedure ssn_refines_ContractZero() {
```

```
  var oh,h: Heap;
  var h',h'': Heap;
  var z: int;
  var cell: AddCell;

  /* ContractZero.pre ⇒ pre */
  havoc oh, z, cell;
  assume ContractZero.pre(oh, cell, z);
  assert Store.pre(oh, cell, z);

  havoc h';
  assume Store.post(oh, h', cell, z);
  assert Sqrt.pre(h', cell);

  havoc h'';
  assume Sqrt.post(h', h'', cell);
  assert Neg.pre(h'', cell);

  /* post ⇒ ContractZero.post -- Requires the previous steps */
  havoc h;
  assume Neg.post(h'', h, cell);
  assert ContractZero.post(oh, h, cell, z);
}

procedure ssn_refines_ContractZero_alternative() {
  /* Equivalent closed-formula encoding, presented for illustrative
   * purposes only.
   */
  assert (
    forall oh: Heap, cell: AddCell, z: int ::
      ContractZero.pre(oh, cell, z)
        ==> Store.pre(oh, cell, z) &&
          (forall h': Heap :: Store.post(oh, h', cell, z)
            ==> Sqrt.pre(h', cell) &&
              (forall h'': Heap :: Sqrt.post(h', h'', cell)
                ==> Neg.pre(h'', cell) &&
                  (forall h: Heap ::
                    Neg.post(h'', h, cell)
                      ==> ContractZero.post(oh, h, cell, z))
              )
          )
  ); }
```