Research in Computer Science

Title: Verifying Scala Traits

Group: Chair of Programming Methodology

Student: Malte Schwerhoff <scmalte@ethz.ch>

Supervisors: Valentin Wüstholz <valentin.wuestholz@inf.ethz.ch>

Prof. Peter Müller <peter.mueller@inf.ethz.ch>

Date: 2010-03-10

1 Introduction

This *Research in Computer Science* project is part of an effort to develop an automatic verification system (similiar to Spec# [SpecSharp] or the Java Modelling Language JML [JML]) for the Scala [Scala] 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 focus of this project is on the verification of Scala *traits*. Traits, also known as *mixins* (see the glossary of [Odersky08] for a definition of the two terms in the context of Scala), have been introduced by Moon [Moon] in 1986 and are seen as a solution to the *diamond* or *linearisation problem* ([Bracha90a], [Ducasse06a]) 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.

The verification will be done with Boogie [Boogie], a programme verifier developed at Microsoft Research. Boogie performs a series of transformations from a source programme written in the Boogie Programming Language (BoogiePL, current version being Boogie 2 [Boogie2]) and eventually feeds the result to a theorem prover.

BoogiePL is a procedural intermediate language that includes contracts and that is used to verify several high-level languages such as Spec# or C# by first encoding the source programme in BoogiePL and then having Boogie verify this encoding.

To the best of my knowledge there currently exists no research project with a similar goal. One reason for this might be that the interest in static verification is naturally limited in the communities of untyped dynamic languages where traits originate from.

2 Motivation

The following Scala snippet is adopted from [Odersky08] and shall work as a motivational example to point out possible challenges regarding the verification of traits.

```
abstract class IntCell {
  def get(): Int
  def set(x: Int)
}

class BasicIntCell extends IntCell {
  private var x: Int = 0
  def get() = x
  def set(x: Int) { this.x = x }
}

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

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

val cdi = new BasicIntCell with Increment with Doubling
  /* Double first, then increment. */
val cid = new BasicIntCell with Doubling with Increment
  /* Increment first, then double. */

cdi.set(1); cdi.set(2); cdi.set(3);
  /* cdi.get() would return 3, 5, 7, respectively. */
cid.set(1); cid.set(2); cid.set(3);
  /* cid.get() would return 4, 6, 8, respectively. */
```

This simple example already gives rise to several interesting questions:

Contract language expressiveness
How do we specify a postcondition for `Doubling.set` without knowing (inside the declaration of `Doubling`) to which class or trait `super.set` will be bound when the trait is applied? Can we specify the postcondition by parametrising the postcondition of `super.set`? Can this already be done (conveniently) with established contract languages such as Spec#, JML or Eiffel [Eiffel]?

Contract inheritance
Can we verify that `cdi.get` will always return an odd value after calling `cdi.set`, whereas `cid.get` will always return an even value? Generally speaking, how do we cope with the fact that traits can be mixed into arbitrary classes or objects in an arbitrary order?

Modularity
Does it make sense to verify `Doubling` and `Increment` only once (locally, on their own) or do we need to reverify traits again anyway whenever they are mixed into a class or object? Even if it is reasonable, is it also sufficient to verify them locally?

3 Project Plan

1. Developing a formalism that captures every possible application of traits and thereby allows the verification of traits in an arbitrary scenario is a goal too demanding for this research project. Thus, a reasonable subset (3 to 5) of use-cases has to be identified which capture the most popular (and, from the point of good software engineering, desired) uses of traits.

   The obvious resources for such an identification are books about Scala, the Scala mailing lists and adequately sized software written in Scala, e.g. the Scala compiler, the Scala libraries (collections, actors) and the Lift framework.

2. Exemplifying each of the identified use-cases in its own Scala programme that is short but still points out the challenges it imposes on static verification.

3. Sketching a contract language that is at least expressive enough to be applicable to the implemented examples and also justifies the assumption that it is (easily) extendible to cover Scala in general.

   I expect the pivotal question to be whether it is possible to specify non-trivial contracts for traits that are nonetheless permissive enough, so that the traits can still be used in as many desirable scenarios as possible.

4. Specifying the examples with contracts by adding contract, so that the correctness of the examples can be verified.

5. Finally, the annotated Scala programmes have to be manually encoded in BoogiePL, such that Boogie is able to verify the correctness of the examples.

6. (Optional) Implementing an automatic encoder from contract-annotated Scala traits to BoogiePL, probably by extending the Master's thesis of Valentin Wüstholz [Wüstholz].

4 Deliverables

1. An initial presentation on the project that is to be conducted and a final presentation on the project's results

2. A project report

3. Programme implementations in Scala and in BoogiePL, as well as the results of the verification with Boogie

5 Work Load Estimation

Phase	Work load
Identifying use-case	10%
Exemplifying use-cases	5%
Sketching contract language	15%
Specifying examples	15%
Encoding in BoogiePL	30%
Presentations and report	25%

6 Bibliography

Scala http://www.scala-lang.org/

SpecSharp Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart devices* (CASSIS 2004), volume 3362 of LNCS, 2005, Springer

JML G. Leavens, A. L. Baker, and C. Ruby. JML: a notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, 1999, Springer

Moon David A. Moon. Object-Oriented Programming with Flavors. *OOPSLA Proceedings*, 1986, ACM Press

Boogie 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* 2005, volume 4111 of LNCS, 2006, Springer

Boogie2 K. Rustan M. Leino. This is Boogie 2. (Draft). http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf

Odersky08 Martin Odersky, Lex Spoon and Bill Venners. Programming in Scala. 2008, Artima Inc.

Ducasse06a Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *Transactions on Programming Languages and Systems* 28, 2006

Bracha90a Gilad Bracha and William Cook. Mixin-based inheritance. *OOPSLA Proceedings*, 1990, ACM Press

Eiffel Eiffel: Analysis, Design and Programming Language. Standard ECMA-367 (2nd edition), 2006, Ecma International

Wüstholz Valentin Wüstholz. Encoding Scala Programs for the Boogie Verifier. Master's Project Report, 2009, ETH Zurich