`

Extensible Code Contracts for Scala

Master Thesis Description

Rokas Matulis

1. Introduction

Code contracts is a way to provide formally verifiable specifications for software components. Eiffel was the pioneering language in the field, which introduced Design by Contract [1], lately Microsoft has built Code Contracts library [2] which enables developers to write specifications on the .NET platform using regular library calls.
The goal of this thesis is to implement a similar code contracts library for the Scala programming language, which would enable Scala programmers to provide formal specifications for their programs. Code contracts for Scala is a starting point for further research and should enable easy extendibility to provide more specific contracts and to build static analysis tools that would reason about programs based on these formal specifications and that would also provide compile time results.

2. Goals

This project consists of several steps and concentrates more on building the whole chain of components for the tool to work, than on supporting a wide range of specification features right away, so restrictions are necessary.

2.1. Library

The first part consists of designing and implementing the core part of a code contracts library, which are pre- and post-conditions for functions. It should be possible to extend it with more specifications such as object and loop invariants, access permissions or modification clauses.
Pre- and post-conditions will take a boolean expression which might include pure functions, for which appropriate annotations will be necessary. Regarding forall and exists expressions over lists, sets, ranges, etc., these are already supported by the Scala collections library.

2.2. Rewriting

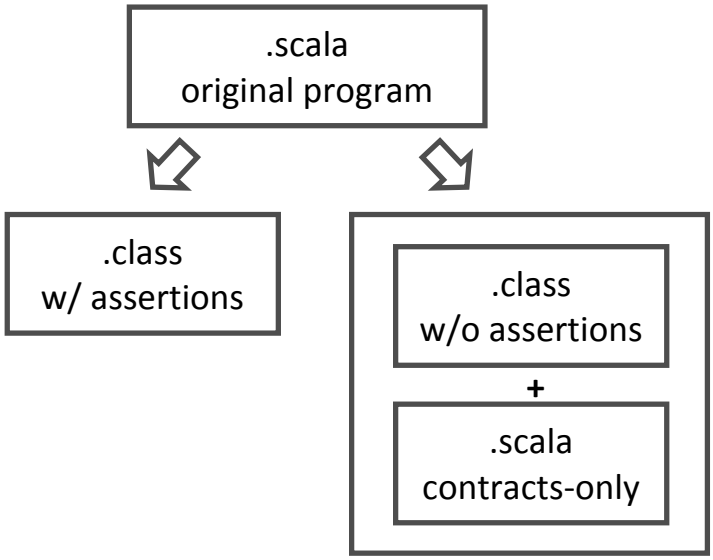
The rewriting part is done on Scala code, not on JVM bytecode, and consists of developing a Scala compiler plug-in which will in essence modify the AST at a specific stage and based on selected option will output either of the two results (as shown in Figure 1):

1.1 An AST which includes inserted assertions and should be passed on to subsequent compilation phases, the final output will thus be compiled .class files with runtime assertions (which could be used for debug builds).

2.1 Separate contracts-only files, which would contain only the contracts for a certain class (this enables providing separate contracts for already written and compiled libraries)

2.2 An AST without any contracts for further compilation; the final output would be compiled .class files without any runtime assertions (which could be used for release builds).

`



2.3.Well definedness

Specifications should be well defined, for instance, they must comply with the following rules:

1. Private fields cannot be used in preconditions
2. Only pure functions can be used (but purity and termination is not checked, since these are orthogonal issues)

2.4.Scala subset

The aim of the project is to concentrate more on the entirety of the working tool rather than full support of all language features, so only a subset of Scala will be used, namely classes, inheritance, methods and recursion, however, no traits and no high-order functions.

2.5.Source code

All submitted source code is well-documented and thoroughly tested.

3. Possible Extensions

1. New specifications could be introduced by subclassing from the main contracts class:
    a. loop invariants,
    b. permissions
2. The subset of Scala supported by the library could be extended, for instance, by supporting exceptions
3. The compiler plug-in can be extended to support options (provided by the user), which would define the behavior of the rewriting. For example, one could specify whether access permissions should only be checked statically or at runtime as well.

4. Project Plan

1. **1 month:** Analysis and design
2. **3 months:** Core implementation
3. **2 months:** Extensions and write up

5. References

1. "Applying Design by Contract," B. Meyer, IEEE Computer, pp. 40-51, October 1992.
2. http://research.microsoft.com/en-us/projects/contracts/