Contract Monitoring and Call-by-name Evaluation

— Extended Abstract —

Markus Degen, Peter Thiemann, and Stefan Wehr

Institut für Informatik, Universität Freiburg, Georges-Köhler-Allee 079,
79110 Freiburg, Germany
{degen,thiemann,wehr}@informatik.uni-freiburg.de

Abstract. Contracts are a proven tool in software development. They provide specifications for operations that may be statically verified or dynamically validated by contract monitoring.
Contract monitoring for strict languages has an established theoretical basis. For languages with call-by-name evaluation, several styles of contract monitoring are possible. In this article, we study two such styles: *eager monitoring* enforces a contract when it is demanded, possibly evaluating expressions not touched by the user code, whereas *delayed monitoring* only proceeds as far as the user code itself can observe.
In each case, an effect system ensures that contract monitoring does not change the meaning of a program and guarantees that contract monitoring is idempotent. Our formalization brings forward semantic reasons that favor delayed monitoring for a call-by-name language and comes with a Haskell implementation.

1 Introduction

Design by contract [8] is a methodology for constructing correct software. Each operation is associated with a contract that defines two assertions, a *precondition* and a *postcondition*, for the operation. The contract is fulfilled if the usual partial correctness condition is true: If the input meets the precondition and the operation produces output, then the output is obliged to meet the postcondition. A program run *violates* a contract if any of the pre- and postconditions of the operations involved is false. Thus, a contract provides a (partial) specification of an operation that every implementation of the operation must fulfill.

While contracts can be verified statically, in practice they are often enforced dynamically using *contract monitoring* (cf. Eiffel [7, 6], Java [1, 5], Scheme [10], or Haskell [4]): The implementation of an operation with monitoring checks the precondition before performing its computation and checks the postcondition before returning to its caller. If the precondition of the operation is false, then it raises an exception blaming its caller. Conversely, if the postcondition does not hold, then the operation blames itself.

The semantics of contract monitoring is intricate, and its correct and complete implementation is non-trivial [3]. From a practical point of view, contract

monitoring should guarantee at least meaning preservation (MP) and behave idempotently (IP):

MP: If a program run with contract monitoring enabled has no contract violations, then disabling contract monitoring should not change its meaning.
IP: Applying a contract multiple times is equivalent to applying it once.

The MP property ensures that developers may enable contract monitoring for a test version of their software and safely disable contract monitoring for the release version, without running the risk that the test and the release version behave differently. The IP property ensures a meaningful notion of contract composition.

In the context of call-by-value evaluation, there is only one useful and sensible mode of contract monitoring. This mode corresponds to its implementation in Eiffel, Java, and Scheme and is the one we just described.

In the context of call-by-name evaluation, there are at least two options, *eager monitoring* and *delayed monitoring*. Eager monitoring enforces an assertion when it is demanded. That is, it checks the precondition when the function demands its argument and it checks the postcondition when the caller demands the function's result. This eager strategy sometimes leads to undesirable behavior which violates the IP property as pointed out by Hinze et al. [4].

We have developed a formalization which precisely pinpoints where eager monitoring imposes too many restrictions on expressions subject to a contract. Hence, we propose delayed monitoring as an alternative form of contract monitoring for languages with call-by-name evaluation. Delayed monitoring places no restrictions on expressions subject to a contract and enjoys the MP and IP properties. It defers enforcement of an assertion until all values that it depends on are evaluated by user code. Thus, monitoring only proceeds as far as the user code itself can observe. Violations that the user code cannot observe, yet, are considered to be invisible. Perhaps surprisingly, this delayed interpretation has a logical foundation: While call-by-value monitoring checks properties according to classical logic, lazy monitoring relies on a three-valued logic.

Contributions

We have developed a semantic framework for specifying and comparing contract monitoring in functional languages [2]. The basis of the framework is an extended version of Moggi's monadic metalanguage [9] with a fixed monad providing for nontermination, mutable state, and exceptions[1] and an effect system for keeping track of the effects.

We have developed the semantics of two styles of contract monitoring for impure functional languages with call-by-name evaluation, eager and delayed monitoring. Both are defined by translation into the metalanguage.

[1] Usually, call-by-name languages provide non-termination as the only effect. However, there is often a back door that allows other effects to creep in. In Haskell, this back door is called `unsafePerformIO`.

The semantics enables us to formally prove (or disprove) the MP and IP properties. The semantics also explains and helps fixing a problem with eager monitoring observed by Hinze et al. [4]. We define and prove correct a criterion that fixes the problem by imposing a suitable typing discipline.

We also provide a prototype implementation of delayed monitoring in Haskell [2].

References

1. P. Abercrombie and M. Karaorman. jContractor: Design by contract for Java. `http://jcontractor.sourceforge.net/`, 2003.
2. M. Degen, P. Thiemann, and S. Wehr. Contract monitoring and call-by-name evaluation. Technical Report 243, Institut für Informatik, Universität Freiburg, http://proglang.informatik.uni-freiburg.de/projects/contracts/, Oct 2008. Full paper and implementation.
3. R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proc. 16th ACM Conf. OOPSLA*, pages 1–15, Tampa Bay, FL, USA, 2001. ACM Press, New York.
4. R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *Proc. Eighth International Symposium on Functional and Logic Programming FLOPS 2006*, Fuji Susono, Japan, Apr. 2006. Springer.
5. R. Kramer. iContract — the Java design by contract tool. In *TOOLS 26: Technology of Object-Oriented Languages and Systems*, pages 295– 307, Los Alamitos, CA, USA, 1998.
6. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.
7. B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
8. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.
9. E. Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1991.
10. The PLT Group. *PLT MzLib: Libraries Manual*. Rice University, University of Utah, Dec. 2005. Version 300.