Lazy Behavioral Subtyping ⋆

Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen

Dept. of Informatics, University of Oslo, Norway
{johand,einarj,olaf,msteffen}@ifi.uio.no

Abstract. Late binding allows flexible code reuse but complicates formal reasoning significantly, as a method call's receiver class is not statically known. This is especially true when programs are incrementally developed by extending class hierarchies. This talk presents a novel method to reason about late bound method calls. In contrast to traditional behavioral subtyping, reverification is avoided without restricting method overriding to fully behavior-preserving redefinition. The approach ensures that when analyzing the methods of a class, it suffices to consider that class and its superclasses. Thus, the full class hierarchy is not needed, and *incremental* reasoning is supported. We formalize this approach as a calculus which lazily imposes context-dependent subtyping constraints on method definitions. The calculus ensures that all method specifications required by late bound calls remain satisfied when new classes extend a class hierarchy. The calculus does not depend on a specific program logic, but the examples use a Hoare-style proof system. We show soundness of the analysis method.

1 Motivation

Late binding of method calls is a central feature in object-oriented languages and contributes to flexible code reuse. A class may extend its superclasses with new methods, possibly overriding the existing ones. This flexibility comes at a price: It significantly complicates reasoning about method calls as the binding of a method call to code cannot be statically determined; i.e., the binding at run-time depends on the actual class of the called object. In addition, object-oriented programs are often designed under an *open world assumption*: Class hierarchies are extended over time as subclasses are gradually developed and added. In general, a class hierarchy may be extended with new subclasses in the future, which will lead to new potential bindings for overridden methods.

To control this flexibility, existing reasoning and verification strategies impose restrictions on inheritance and redefinition. One strategy is to ignore openness and assume a "closed world"; i.e., the proof rules assume that the complete inheritance tree is available at reasoning time (e.g., [9]). This severely restricts the applicability of the proof strategy; for example, libraries are designed to be extended. Moreover, the closed world assumption contradicts inheritance as an object-oriented design principle, which is intended to support incremental development and analysis. If the reasoning relies on the world being closed, extending the class hierarchy requires a costly reverification.

⋆ This research is partially funded by the EU project IST-33826 CREDO: Modeling and analysis of evolutionary structures for distributed services (http://credo.cwi.nl).

An alternative strategy is to reflect in the verification system that the world is open, but to constrain how methods may be redefined. The general idea is that to avoid reverification, any redefinition of a method through overriding must *preserve* certain properties of the method being redefined. An important part of the properties to be preserved is the method's contract; i.e., the pre- and postconditions for its body. The contract can be seen as a description of the promised behavior of all implementations of the method as part of its interface description, the method's *commitment*. Best known as *behavioral subtyping* (e.g, [1,7,8,10]), this strategy achieves incremental reasoning by limiting the possibilities for code reuse. Once a method has committed to a contract, this commitment may not change in later redefinitions. That is overly restrictive and often violated in practice [11]; e.g., it is not respected by the standard Java library definitions.

2 Contribution

In this work, we relax the property preservation restriction of behavioral subtyping, while embracing the open world assumption of incremental program development. The basic idea is as follows: given a method m declared with p and q as the method's pre- and postcondition, there is no need to restrict the behavior of methods overriding m and require that these adhere to that specification. Instead it suffices to preserve the "part" of p and q actually *used to verify* the program at the current stage. Specifically, if m is used in the program in the form of a method call $\{r\}\ e.m()\ \{s\}$, the pre- and postconditions r and s at that call-site constitute m's *required* behavior and it is those weaker conditions that need to be preserved to avoid reverification. Thus, we distinguish declaration-site specifications, which need not be enforced on redefinitions, from call-site requirements, which are in fact enforced on redefinitions. This distinction leads to behavioral subtyping "by need". We call the corresponding analysis strategy *lazy behavioral subtyping*. This strategy may serve as a blueprint for integrating a flexible system for program verification of late bound method calls into object-oriented program development and analysis tools environments [2–4].

The presentation uses an object-oriented kernel language, based on Featherweight Java [6], and Hoare-style proof outlines. We formalize lazy behavioral subtyping as a syntax-driven inference system in which the analysis of a class is done in the context of a *proof environment* constructed during the analysis. The proof environment keeps track of the context-dependent requirements on method definitions, derived from late bound calls. The strategy is incremental; for the analysis of a class C, only knowledge of C and its superclasses is needed. Proofs derived in the context of superclasses are never violated by later extensions to the class hierarchy. We show the soundness of the proposed analysis strategy. The talk builds on previously published work by the authors [5], but extends this work with methodological aspects and applications in the context of multiple inheritance.

References

1. P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 60–90. Springer, 1991.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Intl. Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.
4. L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, , and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Proceedings of FMICS '03*, volume 80 of *ENTCS*. Elsevier Science Publishers, 2003.
5. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. In J. Cuellar and T. Maibaum, editors, *Proc. 15th Intl. Symposium on Formal Methods (FM'08)*, volume 5014 of *LNCS*, pages 52–67. Springer, May 2008.
6. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
7. G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20a, Department of Computer Science, Iowa State University, Ames, Iowa, 2006.
8. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.
9. C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 343(3):413–442, 2005.
10. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *8th European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999.
11. N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. Fifth International Conference on Software Reuse (ICSR5)*, pages 206–215. IEEE Computer Society Press, 1998.