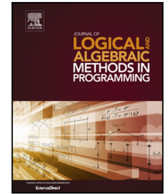
Journal of Logical and Algebraic Methods in Programming 84 (2015) 37–53

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



A proof system for adaptable class hierarchies ☆

Johan Dovland, Einar Broch Johnsen *, Olaf Owe, Ingrid Chieh Yu *

Department of Informatics, University of Oslo, Norway

ARTICLE INFO

Article history:
Received 24 August 2012
Received in revised form 13 August 2014
Accepted 14 September 2014
Available online 21 September 2014

Keywords:
Software evolution
Object orientation
Verification
Proof systems
Class updates
Dynamic code modification

ABSTRACT

The code base of a software system undergoes changes during its life time. For object-oriented languages, classes are adapted, e.g., to meet new requirements, customize the software to specific user functionalities, or refactor the code to reduce its complexity. However, the adaptation of class hierarchies makes reasoning about program behavior challenging; even classes in the middle of a class hierarchy can be modified. This paper develops a proof system for analyzing the effect of operations to adapt classes, in the context of method overriding and late bound method calls. The proof system is *incremental* in the sense that reverification is avoided for methods that are not explicitly changed by adaptations. Furthermore, the possible adaptations are not unduly restricted; i.e., flexibility is retained without compromising on reasoning control. To achieve this balance, we extend the mechanism of *lazy behavioral subtyping*, originally proposed for reasoning about inheritance when subclasses are added to a class hierarchy, to deal with the more general situation of adaptable class hierarchies and changing specifications. The reasoning system distinguishes *guaranteed* method behavior from *requirements* toward methods, and achieves incremental reasoning by tracking guarantees and requirements in adaptable class hierarchies. We show soundness of the proposed proof system.

© 2014 Elsevier Inc. All rights reserved.

1. Introduction

An intrinsic property of software in the real world is that it needs to evolve. This can be during the initial *development* phase, as *improvements* to meet new requirements, or as part of a software *customization* process such as, e.g., feature selection in software product lines or delta-oriented programming [1]. As the code is enhanced and modified, it becomes more complex and drifts away from its original design [2]. For this reason, it may be desirable to redesign the code base to improve its structure, thereby reducing software complexity. For example, the process of *refactoring* in object-oriented software development describes changes to the internal structure of software to make the software easier to understand and cheaper to modify without changing its observable behavior [3]. In this paper, the term *adaptable class hierarchies* covers transformations of classes during the development, improvement, customization, and refactoring of class hierarchies. Adaptations are achieved by means of update operations for adding code to a class such as new fields, method definitions, and implements clauses, by modifying method definitions, and by removing methods (under certain conditions).

Reasoning about properties of object-oriented systems is in general non-trivial due to complications including class inheritance and late binding of method calls. Object-oriented software development is based on an open world assumption;

☆ This work was done in the context of the EU project FP7-610582 *ENVISAGE: Engineering Virtualized Services* (http://www.envisage-project.eu).

* Corresponding author.
E-mail addresses: johand@ifi.uio.no (J. Dovland), einarj@ifi.uio.no (E.B. Johnsen), olaf@ifi.uio.no (O. Owe), ingridcy@ifi.uio.no (I.C. Yu).

http://dx.doi.org/10.1016/j.jlamp.2014.09.001
2352-2208/© 2014 Elsevier Inc. All rights reserved.

i.e., class hierarchies are typically extendable. In order to have reasoning control under such an open world assumption, it is advantageous to have a framework which controls the properties required of method redefinitions. With a *modular reasoning* framework, a new subclass can be analyzed in the context of its superclasses, such that the properties of the superclasses are guaranteed to be maintained. This has the significant advantage that each class can be fully verified at once, independent of subclasses which may be designed later. The best known modular reasoning framework for class hierarchies is *behavioral subtyping* [4]. However, behavioral subtyping has been criticized for being overly restrictive and is often violated in practice [5]. For these reasons it is interesting to explore alternative approaches which allow more flexibility, although this may lead to more proof work.

Incremental reasoning frameworks generalize modular reasoning by possibly generating new verification conditions for superclasses in order to guarantee established properties. Additional properties may be established in the superclasses after the initial analysis, but old properties remain valid. Observe that these frameworks subsume modularity: if the initial properties of the classes are sufficiently strong (for example by adhering to a behavioral contract), it will never be necessary to add new properties later. *Lazy behavioral subtyping* is a formal framework for such incremental reasoning, which allows more flexible code reuse than modular frameworks. The basic idea underlying lazy behavioral subtyping is a separation of concerns between the behavioral *guarantees* of method definitions and the behavioral *requirements* to method calls. Both guarantees and requirements are manipulated through an explicit proof-context: a book-keeping framework controls the analysis and proof obligations in the context of a given class. Properties are only inherited by need. Inherited requirements on method redefinition are as weak as possible while still ensuring soundness.

Lazy behavioral subtyping seems well-suited for the incremental reasoning style desirable for object-oriented software development, and can be adjusted to different mechanisms for code reuse. It was originally developed for single inheritance class hierarchies [6], but has later been extended to multiple inheritance [7] and to trait-based code reuse [8].

Adaptable class hierarchies add a level of complexity to proof systems for object-oriented programs, as superclasses in the middle of a class hierarchy can change. Unrestricted, such changes may easily violate previously verified properties in both sub- and superclasses. The management of verification conditions becomes more complicated than in the case of extending a class hierarchy at the bottom with new subclasses. This paper extends the approach of lazy behavioral subtyping to allow incremental reasoning about adaptable class hierarchies.

The approach is presented using a small object-oriented language and a number of *basic update operations* for adapting class definitions. We consider a series of “snapshots” of a class hierarchy during a development and adaptation process. The developer applies basic update operations and analysis steps to the class hierarchy between these snapshots. Based on the lazy behavioral subtyping framework, the specified and required properties of method definitions and method calls are tracked through these adaptation steps.

The paper is structured as follows: Section 2 presents the language considered and provides a motivating example for update operations. Section 3 presents our program logic for classes, based on proof outlines. Section 4 presents lazy behavioral subtyping and the verification environments needed for analyzing class adaptations. We explain the proof system for the basic update operations in Section 5 and finally, related work is discussed in Section 6 before we conclude the paper in Section 7.

2. The programming language

We consider an object-oriented kernel language akin to Featherweight Java [9], in which pointers are typed by behavioral interfaces and methods are annotated with pre/postconditions, The syntax for classes and interfaces in the language is given in Fig. 1. A program consists of a set of interfaces and class definitions, followed by a method body. A behavioral interface I extends a list $\bar{I}$ of superinterfaces and declares a set $\overline{MS}$ of method signatures with behavioral annotations. We refer to the behavioral annotation occurring in a method signature as a *guarantee* for the method. These guarantees reflect semantic constraints on the use of the declared methods, and are given as pairs (p, q) of pre- and postconditions to the signatures. An interface may introduce additional constraints to methods already declared in superinterfaces. A class may inherit from a single superclass, adding fields $\overline{f}$, methods $\overline{M}$, and method guarantees $\overline{MS}$.

Method parameters are read-only, and we assume that programs are well-typed. A method body consists of a sequence of standard statements followed by **return** e, where e is the value returned by the method. Methods of type Void need no return statements and are called without assignment to actual parameters.

$$
\begin{array}{llll}
T ::= I \mid \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Nat} \mid \mathsf{Double} & K & ::= & \textbf{interface}\ I\ \textbf{extends}\ \bar{I}\ \{\overline{MS}\} \\
M ::= MS\ \{\overline{T}\ \bar{x}; t; \textbf{return}\ e\} & L & ::= & \textbf{class}\ C\ \textbf{extends}\ C\ \textbf{implements}\ \bar{I}\ \{\overline{T}\ \overline{f}; \overline{M}\ \overline{MS}\} \\
v ::= f \mid x & MS & ::= & [T \mid \mathsf{Void}]\ m\ (\overline{T}\ \bar{x}) : (a, a) \\
 & t & ::= & t; t \mid v := rhs \mid e.m(\bar{e}) \mid \textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ t\ \textbf{fi} \mid \textbf{skip} \\
 & rhs & ::= & \textbf{new}\ C(\,) \mid e.m(\bar{e}) \mid m(\bar{e}) \mid e
\end{array}
$$

Fig. 1. The language syntax. C is a class name, and I an interface name. Variables v are fields (f) or local variables (x), and e denotes side-effect free expressions over the variables, b expressions of Boolean type, and a is an assertion. Vector notation denotes collections, as in the expression list $\bar{e}$, interface list $\bar{I}$ (with a slight abuse of notation, we write $\overline{T}\ \bar{x}$ to denote a list of variable declarations), and in sets such as $\overline{K}, \overline{L}, \overline{MS}$ and $\overline{M}$. Let *nil* denote the empty list. To distinguish assignments from equations in specifications and expressions, we use := and = respectively.

Download English Version:

https://daneshyari.com/en/article/432253

Download Persian Version:

https://daneshyari.com/article/432253

Daneshyari.com