Encapsulating 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. Object-orientation supports incremental program development by gradually extending the class hierarchy. Subclassing and late bound method calls allow very flexible reuse of code, thereby avoiding code duplication. Combined with incremental program development, this flexibility poses a challenge for program analysis. The dominant solution to this problem is behavioral subtyping, which avoids re-verification of verified code but requires that all properties of a method are preserved in subclasses. Program analysis becomes incremental, but behavioral subtyping severely restricts code reuse. *Lazy behavioral subtyping* relaxes this restriction to the preservation of properties that are *required* by the call-site usage of methods. Previous work developed corresponding inference systems for languages with single- and multiple-inheritance hierarchies, but although incremental the approach could make it necessary to revisit previously analyzed classes in order to establish new properties. In this paper, we combine the proof system for lazy behavioral subtyping with behavioral interfaces to support incremental reasoning in a modular way. A class may be fully analyzed at development time by relying on interface information for external method calls. Furthermore, this separating classes and interfaces, which encapsulates the objects in a cleaner way, leads to a simplification of the formal reasoning system. The approach is presented using a simple object-oriented language (based on Featherweight Java) with *interfaces* and illustrated by an example using a Hoare-style proof system.

1 Introduction

Object-orientation supports an incremental style of program development, as new classes and subclasses may gradually be added to previously developed class hierarchies; these new subclasses typically extend and specialize existing code from superclasses, potentially overriding existing methods. In that way, the code of a late bound method call depends on the run-time class of the callee object, and so its effects are not statically decidable. Subclassing combined with late binding lead to a very flexible mechanism for code reuse, as illustrated through a plethora of design patterns [14], but pose a challenge for program analysis. The intricacies of late-binding, inheritance, encapsulation and other advanced features found in object-oriented languages spawned a lot of research to clarify the semantical foundations, and especially to capture of such features in a type-safe manner. This led to the development of quite expressive type systems and calculi. One typical representative had been given by Qian and Krieg-Brückner [25],

* 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).

who present a language combining features of object-oriented and functional languages in a unified, typed calculus, where much of the expressive power is needed to capture late-binding and overloading.

While the static, type-theoretic foundations of mainstream object-oriented languages are largely understood, verification support still poses challenges. There are two main approaches in the literature to the verification of class hierarchies with late bound method calls. *Behavioral subtyping* was originally proposed by America [2] and Liskov and Wing [19] and later used in, e.g., Spec♯ [18]. This is an *open world* approach: it facilitates reasoning about programs in an incremental way which fits well with the incremental development style of object-oriented programs. Roughly speaking, the basic idea of behavioral subtyping is that any property of a supertype should also hold for all subtypes. The approach focuses on the *declared properties* of a type, and applied to the object-oriented setting, any property of a superclass should also hold for all subclasses. The idea is appealing, as it provides substitutability not just for the static signatures of objects, as in standard subtyping, but at the behavioral level. Behavioral subtyping, however, imposes severe restrictions on subclassing, limiting how code may be reused in a way which breaks with programming practice [26]. For example, the class hierarchies of Java libraries do not obey the behavioral subtyping discipline. Alternatively, one can accept the practice of unrestricted code reuse and overriding, and capture that in a reasoning system. For instance, Pierik and de Boer [24] have proposed a complete proof system for object-oriented programs which is able to address code reuse in a much more flexible way. However, it is a *closed world* approach: it requires the full class hierarchy to be available at analysis time to ensure that any binding meets the requirements imposed by the usage of values of the type. This means that the approach focuses on the *required properties* of a type. Thus the approach overcomes the limitations of behavioral subtyping, but breaks with incremental reasoning.

Recently, *lazy behavioral subtyping* has been proposed by the authors with the aim to preserve the appealing features of behavioral subtyping, i.e., incremental reasoning, but allow more flexible code reuse in a controlled way. Lazy behavioral subtyping balances the *required* properties reflecting the call-site use of a method with its *provided* properties, and the basic insight is that the properties that need to be preserved depend on the *use* of a method rather than on its declared contract. Previous use, therefore, imposes restrictions on future redefinitions in order to maintain the incremental reasoning property. The approach is supported by an inference system which tracks declaration site specifications and call site requirements for methods in an extensible class hierarchy [12]. This inference system, which is independent from the underlying specific program logic of a given reasoning system, ensures that proven properties are not violated due to method redefinition in new subclasses, and that required properties of a method always hold for redefinitions. The approach has later been extended to deal with multiple inheritance [13].

These previous papers present a slightly simplistic version of lazy behavioral subtyping, with the aim to concentrate on the core mechanisms for flexible code reuse without breaking the principle of incremental reasoning. In particular, we considered a language without interfaces, i.e., classes played the roles of types for objects and of generators of objects instances at the same time. As a consequence, external method

$$
\begin{array}{lll}
P & ::= \overline{K}\,\overline{L}\,\{t\} & K ::= \texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\overline{MS}\} \\
MS & ::= m\,(\overline{x}) : (p,q) & L ::= \texttt{class}\ C\ \texttt{extends}\ C\ \texttt{implements}\ I\ \{\overline{f}\,\overline{M}\ \texttt{inv}\ p\} \\
M & ::= m\,(\overline{x}) : (p,q)\ \{t\} & e ::= f \mid \texttt{this} \mid b \mid \texttt{new}\ C() \mid e.m(\overline{e}) \mid m(\overline{e}) \\
 & & t ::= f := e \mid \texttt{return}\ e \mid \texttt{skip} \mid \texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ t\ \texttt{fi} \mid t;t
\end{array}
$$

Fig. 1. The language syntax, where I, C and, m are interface, class, and method names (of types *Iid*, *Cid*, and *Mid*, respectively), and p and q are assertions. Vector notation denotes lists, as in the expression list $\overline{e}$.

calls could recursively lead to new proof obligations in previously analyzed classes. In this paper, we aim to combine lazy behavioral subtyping with a notion of modularity for external calls, so that lazy behavioral subtyping applies to internal code reuse and additional proof obligations are avoided for classes which have already been analyzed. For this purpose, the type hierarchy will be separated from the class hierarchy, and behavioral interfaces are introduced to type object variables and references. Thus, a class which inherits a superclass need not inherit the type of the superclass, and may thereby reuse code more freely. As this approach can be encoded in the general method of lazy behavioral subtyping, soundness of the proof system of this paper follows directly from the soundness of the pure lazy behavioral subtyping method (see [12]).

The remainder of the paper is structured as follows. Section 2 presents a variant of Featherweight Java as the language we use for our development. In Section 3, we first present the data structures needed to keep track of the different proof obligations, and afterwards in Section 4 the inference system to analyse a class hierarchy. The method is illustrated in Section 5 on an example, and Section 6 discusses the context of this work.

2 The Programming Language

Let us consider a programming language based on Featherweight Java [17], but extended with (behavioral) interfaces. A program P consists of a set $\overline{K}$ of interfaces, a set $\overline{L}$ of classes, and an initial statement t. The syntax is given in Fig. 1 and explained below.

2.1 Behavioral Interfaces

A *behavioral interface* consists of a set of method names with signatures and semantic constraints on the use of these methods. In Fig. 1, an interface I may extend a list $\overline{I}$ of superinterfaces, and declare a set $\overline{MS}$ of method signatures, where behavioral constraints are given as specifications $(pre, post)$ of pre- and postconditions to the signatures. An interface may declare signatures of new methods not found in its superinterfaces, and it may declare additional specifications of methods declared in the superinterfaces. The relationship between interfaces is restricted to a form of behavioral subtyping. An interface may extend several interfaces, adding to its superinterfaces new syntactic and

semantic constraints. We assume that the interface hierarchy conforms with these requirements. The interfaces thus form a type hierarchy: if I' extends I, then I' is a *subtype* of I and I is a *supertype* of I'. Let $\preceq$ denote the reflexive and transitive subtype relation, which is given by the nominal extends-relation over interfaces. Thus, $I' \preceq I$ if I' equals I or if I' extends (directly or indirectly) I.

An object *supports* an interface I if the object provides the methods declared in I and adheres to the specifications imposed by I on these methods. Fields are typed by interfaces; if an object supports I then the object may be referenced by a field typed by I. A class *implements* an interface if its code is such that all instances support the interface. The analysis of the class must ensure that this requirement holds. Objects of different classes may support the same interface, corresponding to different implementations of the same behavior. Note that only the methods declared by I are available for external invocations on references typed by I, but the class may implement additional *auxiliary* methods.

The *substitution principle* for objects applies to the level of interfaces: *an object supporting an interface I may be replaced by another object supporting I or a subtype of I*. A subclass C' of C need not satisfy the interface I of the superclass. If I is not implemented by C', the substitution principle ensures that an instance of C' cannot be used where an object of type I is expected. If a field x is declared with interface I, the actual object referenced by x at run-time will satisfy the behavioral specification of I. However, as object references are typed by interface, the run-time class of a called object is hidden by the behavioral interface of that object. Consequently, all external method calls are late bound.

2.2 The Imperative Language

The imperative part of the language consists of classes which may implement an interface, inherit from one superclass, and define fields $\overline{f}$ and methods $\overline{M}$ (see Fig. 1). The superclass is given by the **`extends`** clause in the class header, and the interface supported by instances of the class is given by the **`implements`** clause. The syntactic parts of a class are referred to by the functions *inh*, *att*, *mtds*, and *impl*, returning the superclass name, attributes, methods, and interface, respectively. Let $\leq$ denote the reflexive and transitive subclass relation, such that $C' \leq C$ if C' equals C, or C' extends (directly or indirectly) C.

For analysis purposes, a class may specify an invariant **`inv`** p, where p is a predicate over the fields of the class (implemented directly or inherited). As the interface of a class C hides the implementation details of C, also the class invariant is hidden. Thus, an external call $x.m()$, where x refers to an instance of class C, cannot assume that the invariant of x holds when the method starts execution. The imperative language constructs are standard. Expressions e include program variables f and Boolean expressions b, external calls $e.m(\overline{e})$, and self calls are written $m(\overline{e})$. If m does not return a value, or if the returned value is of no concern, we may use directly $e.m(\overline{e})$ and $m(\overline{e})$ as statements for simplicity (ignoring the assignment of the return value to a program variable). Note that the list of actual parameter values may be empty and that the formal parameters x and the reserved variable **`this`** (for self reference) are read-only variables. Statements include assignment $f := e$, **`return`** e which returns an expression e to the

caller, conditionals, and sequential composition. For simplicity, all method calls in this language are late bound. Method binding searches the class hierarchy from the actual class of the called object in the usual way.

3 Class Analysis

An essential part of the verification of a class is to ensure that the methods defined by the class supports the behavioral specification of the interface implemented by the class. We assume that methods are defined in the classes in terms of *proof outlines* [22]; i.e., $m(\overline{x}) : (p,q)\{t\}$ such that t is a method body decorated with pre/post requirements on method calls and $\vdash_{\mathrm{PL}} \{p\}\ t\ \{q\}$ is derivable in the given program logic PL if the requirements hold for the method calls in t. Let $body(C,m)$ denote the decorated method body of m in C. The body is either found by a definition of m in C, or inherited (without redefinition) from a superclass of C.

Notation. Given assertions p and q, we let the type *APair* range over assertion pairs (p,q). If p is the pre- and q the postcondition to some method, we call the pair (p,q) a *specification* of that method. For an interface I, let $public(I)$ denote the set of method identifiers supported by I, so $m \in public(I)$ if m is declared by I or by a supertype of I. As a subtype cannot remove methods declared by a supertype, we have $public(I) \subseteq public(I')$ if $I' \preceq I$. If $m \in public(I)$, we let the function $spec(I,m)$ return a set of type $Set[APair]$ with the behavioral specifications supported by m in I, as declared in I or in a supertype of I. The function returns a set since a subinterface may provide additional specifications of methods inherited from superinterfaces; if $m \in public(I)$ and $I' \preceq I$, then $spec(I,m) \subseteq spec(I',m)$. Finally we define entailment (denoted $\twoheadrightarrow$) between sets of assertion pairs.

Definition 1 (Entailment). *Assume assertion pairs (p,q) and (r,s), and sets $\mathcal{U} = \{(p_i,q_i) \mid 1 \le i \le n\}$ and $\mathcal{V} = \{(r_i,s_i) \mid 1 \le i \le m\}$, and let p' be the assertion p with all fields f substituted by f', avoiding name capture.* Entailment *is defined by*

i) $(p,q) \twoheadrightarrow (r,s) \triangleq (\forall \overline{z}_1 \,.\, p \Rightarrow q') \Rightarrow (\forall \overline{z}_2 \,.\, r \Rightarrow s')$,
where $\overline{z}_1$ and $\overline{z}_2$ are the logical variables in (p,q) and (r,s), respectively

ii) $\mathcal{U} \twoheadrightarrow (r,s) \triangleq (\bigwedge_{1\le i\le n}(\forall \overline{z}_i \,.\, p_i \Rightarrow q'_i)) \Rightarrow (\forall z \,.\, r \Rightarrow s')$.

iii) $\mathcal{U} \twoheadrightarrow \mathcal{V} \triangleq \bigwedge_{1\le i\le m} \mathcal{U} \twoheadrightarrow (r_i,s_i)$.

The relation $\mathcal{U} \twoheadrightarrow (r,s)$ corresponds to Hoare-style reasoning, proving $\{r\}\ t\ \{s\}$ from $\{p_i\}\ t\ \{q_i\}$ for all $1 \le i \le n$, by means of the adaptation and conjunction rules [3]. Entailment is reflexive and transitive, and $\mathcal{V} \subseteq \mathcal{U}$ implies $\mathcal{U} \twoheadrightarrow \mathcal{V}$.

Lazy behavioral subtyping is a method for reasoning about redefined methods and late binding which may be explained as follows. Let m be a method defined in class C. The declared behavior of this method definition is given by the specification set $S(C,m)$, where S is the *specification mapping* taking class and a method name. We assume that for each $(p,q) \in S(C,m)$ there is a proof outline for $body(C,m)$ such that $\vdash_{\mathrm{PL}} \{p\}\ body(C,m)\ \{q\}$. For a self call $\{r\}\ n(\overline{e})\ \{s\}$ in the proof outline, (r,s)

is a *requirement* imposed by C on possible implementations of n to which the call can bind. (Requirements made by external calls are considered below.) Each such requirement, collected during the analysis of C, is included in the set $R(C,n)$, where R is the *requirement* mapping. Lazy behavioral subtyping ensures that all requirements in the set $R(C,n)$ follow from the knowledge of the definition of method n in C; i.e., $S(C,n) \rightarrowtriangle R(C,n)$. If n is later overridden by some subclass D of C, *the same requirements* apply to the new version of n; i.e., $S(D,n) \rightarrowtriangle R(C,n)$ must be proved. This yields an incremental reasoning strategy.

In general, we let $S{\uparrow}(C,m)$ return the accumulated specification set of m in C. If m is defined in C, this is the set $S(C,m)$. If m is inherited, the set is $S(C,m) \cup S{\uparrow}(C.\mathit{inh},m)$. In this manner, a subclass may provide additional specifications of methods that are inherited from superclasses. The requirements toward m that are recorded during the analysis of superclasses are returned by the set $R{\uparrow}(C,m)$ such that $R{\uparrow}(C,m) = R(C,m) \cup R{\uparrow}(C.\mathit{inh},m)$.[1] For each class C and method m defined in C, the lazy behavioral subtyping calculus (see Sec. 4) maintains the relation $S{\uparrow}(C,m) \rightarrowtriangle R{\uparrow}(C,m)$.

If C implements an interface I, the class defines (or inherits) an implementation of each $m \in \mathit{public}(I)$. For each such method, the behavioral specification declared by I must follow from the method specification, i.e., $S{\uparrow}(C,m) \rightarrowtriangle \mathit{spec}(I,m)$. Now consider the analysis of a requirement stemming from the analysis of an external call in some proof outline. In the following, we denote by $x : I.m$ the external call $x.m$ where x is declared with static type I. As the interface hides the actual class of the object referenced by x, the call is analyzed based on the interface specification of m. For an external call $\{r\}\ x : I.m()\ \{s\}$, the requirement (r,s) must follow from the specification of m given by type I, expressed by $\mathit{spec}(I,m) \rightarrowtriangle (r,s)$. Soundness in this setting is given by the following argument. Assume that the call to $x.m$ can bind to m on an instance of class C, and let $I' = C.\mathit{impl}$. Type analysis then ensures that $I' \preceq I$. During the analysis of C, the relation $S{\uparrow}(C,m) \rightarrowtriangle \mathit{spec}(I',m)$ is established. The desired $S{\uparrow}(C,m) \rightarrowtriangle \mathit{spec}(I,m)$ then follows since $\mathit{spec}(I,m) \subseteq \mathit{spec}(I',m)$ when $I' \preceq I$.

The invariant p of a class C is taken as a pre/post specification of each method visible through the supported interface of C. Thus, the invariant is analyzed by proving the specification (p,p) for each such method m. In this manner, the invariant analysis is covered by the general approach of lazy behavioral subtyping as the declaration of an invariant can be considered as an abbreviation of a pre/post specification of each method. Note that this approach does not require that the invariant holds whenever m starts execution; the specification expresses that *if* the invariant holds prior to method execution, then it will also hold upon termination. This approach to invariants works when analyzing extensible class hierarchies [26], even if the invariant of a subclass is different from the superclass invariant. The superclass invariant need not hold in the subclass, but methods defined in the superclass can be inherited by the subclass.

[1] Note that for the language considered in this paper, the set of requirements could be made more fine-grained by removing requirements stemming from redefined method definitions. However, in a language with static calls, this simplification would no longer apply.

$$\mathcal{A} ::= \mathcal{P} \mid \langle C : O \rangle \cdot \mathcal{P} \qquad \mathcal{P} ::= K \mid L \mid \mathcal{P} \cdot \mathcal{P}$$
$$O ::= \varepsilon \mid \mathit{anMtd}(\overline{M}) \mid \mathit{verify}(m, \overline{R}) \mid \mathit{anOutln}(t) \mid \mathit{intSpec}(\overline{m}) \mid \mathit{inv}(p, \overline{m}) \mid O \cdot O$$

Fig. 2. Syntax for analysis operations.

4 The Inference System

Classes and interfaces are analyzed with regard to a *proof environment*. The proof environment tracks the specifications and requirements for the different classes, and the interface specifications that each class must adhere to. Let *Cid*, *Iid*, and *Mid* denote the types of class, interface, and method names, respectively.

Definition 2 (Proof environments). *A proof environment $\mathcal{E}$ of type Env is a tuple $\langle L, K, S, R \rangle$ where $L : Cid \rightarrow Class$, $K : Iid \rightarrow Interface$ are partial mappings and $S, R : Cid \times Mid \rightarrow Set[APair]$ are total mappings.*

Subscript are used to refer to a specific environment; e.g., $S_{\mathcal{E}}$ is the S-mapping of $\mathcal{E}$. Now, *environment soundness* is defined. The definition is adapted from [12] by taking interfaces into account. Condition 3 in the definition captures interface implementations, requiring that each method satisfies the behavioral specification given by the interface.

Definition 3 (Sound environments). *A proof environment $\mathcal{E}$ is sound if it satisfies the following conditions for each $C : Cid$ and $m : Mid$.*

1. $\forall (p,q) \in S_{\mathcal{E}}(C,m) \;.\; \exists \mathit{body}_{\mathcal{E}}(C,m) \;.\; \vdash_{\mathrm{PL}} \{p\}\, \mathit{body}(C,m)\, \{q\}$
   $\wedge\; \forall \{r\}\; n\; \{s\} \in \mathit{body}_{\mathcal{E}}(C,m) \;.\; R_{\mathcal{E}}(C,n) \twoheadrightarrow (r,s)$
   $\wedge\; \forall \{r\}\; x : I.n\; \{s\} \in \mathit{body}_{\mathcal{E}}(C,m) \;.\; \mathit{spec}(I,n) \twoheadrightarrow (r,s)$
2. $S{\uparrow}_{\mathcal{E}}(C,m) \twoheadrightarrow R{\uparrow}_{\mathcal{E}}(C,m)$
3. $\forall n \in \mathit{public}(I) \;.\; S{\uparrow}_{\mathcal{E}}(C,n) \twoheadrightarrow \mathit{spec}(I,n)$, *where* $I = C.\mathit{impl}$.

There are four operations to update a proof environment; these load a new class L or interface K, and extend the commitment and requirement mappings with a pair (p,q) for a given method m and class C. We define an operator $\oplus : Env \times Update \rightarrow Env$, where the first argument is the current proof environment and the second argument is the environment update, as follows:

$$\begin{aligned}
\mathcal{E} \oplus \mathit{extL}(C,D,I,\overline{f},\overline{M}) &= \langle L_{\mathcal{E}}[C \mapsto \langle D,I,\overline{f},\overline{M}\rangle], K_{\mathcal{E}}, S_{\mathcal{E}}, R_{\mathcal{E}}\rangle \\
\mathcal{E} \oplus \mathit{extK}(I,\overline{I},\overline{MS}) &= \langle L_{\mathcal{E}}, K_{\mathcal{E}}[I \mapsto \langle \overline{I},\overline{MS}\rangle], S_{\mathcal{E}}, R_{\mathcal{E}}\rangle \\
\mathcal{E} \oplus \mathit{extS}(C,m,(p,q)) &= \langle K_{\mathcal{E}}, K_{\mathcal{E}}, S_{\mathcal{E}}[(C,m) \mapsto S_{\mathcal{E}}(C,m) \cup \{(p,q)\}], R_{\mathcal{E}}\rangle \\
\mathcal{E} \oplus \mathit{extR}(C,m,(p,q)) &= \langle L_{\mathcal{E}}, K_{\mathcal{E}} S_{\mathcal{E}}, R_{\mathcal{E}}[(C,m) \mapsto R_{\mathcal{E}}(C,m) \cup \{(p,q)\}]\rangle
\end{aligned}$$

In the calculus, judgments have the form $\mathcal{E} \vdash \mathcal{A}$, where $\mathcal{E}$ is the proof environment and $\mathcal{A}$ is a sequence of *analysis operations* (see Fig. 2). The main inference rules are given in Fig. 3. The operations and the calculus are discussed below. We emphasize on the differences wrt. the calculus in [12], which correspond to the introduction of interfaces and class invariants. In the rules, $I \in E$ and $C \in \mathcal{E}$ denote that $K_{\mathcal{E}}(I)$ and $L_{\mathcal{E}}(C)$ are

$$\frac{I \notin \mathcal{E} \qquad \overline{I} \neq nil \Rightarrow \overline{I} \in \mathcal{E} \qquad \mathcal{E} \oplus extK(I, \overline{I}, \overline{MS}) \vdash \mathcal{P}}{\mathcal{E} \vdash (\texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\overline{MS}\}) \cdot \mathcal{P}}\ (\textsc{NewInt})$$

$$\frac{\begin{array}{c} I \in \mathcal{E} \qquad C \notin \mathcal{E} \qquad D \neq nil \Rightarrow D \in \mathcal{E} \\ \mathcal{E} \oplus extL(C, D, I, \overline{f}, \overline{M}) \vdash \langle C : anMtd(\overline{M}) \cdot inv(p, public_{\mathcal{E}}(I)) \cdot intSpec(public_{\mathcal{E}}(I)) \rangle \cdot \mathcal{P} \end{array}}{\mathcal{E} \vdash (\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I\ \{\overline{f}\ \overline{M}\ \texttt{inv}\ p\}) \cdot \mathcal{P}}\ (\textsc{NewClass})$$

$$\frac{\mathcal{E} \vdash \langle C : verify(m, \{(p,q)\} \cup R\!\uparrow_{\mathcal{E}}(P_{\mathcal{E}}(C).\mathsf{inh}, m)) \cdot O \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : anMtd(m(\overline{x}) : (p,q)\ \{t\}) \cdot O \rangle \cdot \mathcal{P}}\ (\textsc{NewMtd})$$

$$\frac{S\!\uparrow_{\mathcal{E}}(C, m) \rightarrowtriangle (p,q) \qquad \mathcal{E} \vdash \langle C : O \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : verify(m, (p,q)) \cdot O \rangle \cdot \mathcal{P}}\ (\textsc{ReqDer})$$

$$\frac{\begin{array}{c} \vdash_{\mathrm{PL}} \{p\}\ body_{\mathcal{E}}(C, m)\ \{q\} \\ \mathcal{E} \oplus extS(C, m, (p,q)) \vdash \langle C : anOutln(body_{\mathcal{E}}(C, m)) \cdot O \rangle \cdot \mathcal{P} \end{array}}{\mathcal{E} \vdash \langle C : verify(m, (p,q)) \cdot O \rangle \cdot \mathcal{P}}\ (\textsc{ReqNotDer})$$

$$\frac{\mathcal{E} \oplus extR(C, m, (p,q)) \vdash \langle C : verify(m, (p,q)) \cdot O \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : anOutln(\{p\}\ m\ \{q\}) \cdot O \rangle \cdot \mathcal{P}}\ (\textsc{LateCall})$$

$$\frac{I \in \mathcal{E} \qquad spec_{\mathcal{E}}(I, m) \rightarrowtriangle (p,q) \qquad \mathcal{E} \vdash \langle C : O \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : anOutln(\{p\}\ x : I.m\ \{q\}) \cdot O \rangle \cdot \mathcal{P}}\ (\textsc{ExtCall})$$

$$\frac{S\!\uparrow_{\mathcal{E}}(C, m) \rightarrowtriangle spec_{\mathcal{E}}(C.\mathsf{impl}, m) \qquad \mathcal{E} \vdash \langle C : O \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : intSpec(m) \cdot O \rangle \cdot \mathcal{P}}\ (\textsc{IntSpec})$$

$$\frac{\mathcal{E} \vdash \langle C : verify(m, (p,p)) \cdot O \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : inv(p, m) \cdot O \rangle \cdot \mathcal{P}}\ (\textsc{Inv})$$

$$\frac{\mathcal{E} \vdash \langle C : inv(p, \overline{m}_1) \cdot inv(p, \overline{m}_2) \cdot O \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : inv(p, \overline{m}_1 \cup \overline{m}_2) \cdot O \rangle \cdot \mathcal{P}}\ (\textsc{DecompInv})$$

$$\frac{\mathcal{E} \vdash \langle C : intSpec(\overline{m}_1) \cdot intSpec(\overline{m}_2) \cdot O \rangle \cdot \mathcal{P}}{\mathcal{E} \vdash \langle C : intSpec(\overline{m}_1 \cup \overline{m}_2) \cdot O \rangle \cdot \mathcal{P}}\ (\textsc{DecompInt})$$

$$\frac{\mathcal{E} \vdash \mathcal{P}}{\mathcal{E} \vdash \langle C : \varepsilon \rangle \cdot \mathcal{P}}\ (\textsc{EmpClass})$$

Fig. 3. The inference system, where $\mathcal{P}$ is a (possibly empty) sequence of classes and interfaces. To simplify the presentation, we let m denote a method call including actual parameters. Let *nil* denote the empty list.

defined, respectively. For brevity, we elide a few straightforward rules which formalize a lifting from single-elements to sets or sequences of elements. For example, the rule for $anMtd(\overline{M})$ (which occurs in the premise of (NewClass)), generalizes the analysis of a single method which is done in (NewMtd). The omitted rules may be found in [12]

and are similar to the decomposition rules (DECOMPINT) and (DECOMPINV) for interface and invariant requirements.

A program is analyzed as a sequence of interfaces and classes. For simplicity, we require that superclasses appear before subclasses and that interfaces appear before they are used. This ordering ensures that requirements imposed by superclasses are verified in an incremental manner on subclass overridings. Rules (NEWINT) and (NEWCLASS) extend the environment with new interfaces and classes; the introduction of a new class leads to an analysis of the class. The specification and requirement mappings are extended based on the internal analysis of each class. We assume that programs are well-typed. Especially, if a field x is declared with type I and there is a call to a method m on x, then m is assumed to be supported by I. Rule (NEWCLASS) generates an operation of the form $\langle C : O \rangle$, where O is a sequence of analysis operations to be performed for class C. Note that (NEWINT) and (NEWCLASS) cannot be applied while a $\langle C : O \rangle$ operation is analyzed, which ensures that $\langle C : O \rangle$ is analysed before a new class is analyzed. A successful analysis of C yields an operation $\langle C : \varepsilon \rangle$ which is discarded by (EMPCLASS).

For a class C implementing an interface I, (NEWCLASS) generates three initial operations *anMtd*, *inv*, and *intSpec*. For each method m defined in C, *anMtd* collects the inherited requirements toward m and any user given specification of the method, analyzed in (NEWMTD). Rule (INV) analyses the class invariant as a pre/post specification of each externally visible method in C. Finally, (INTSPEC), ensures that the implementation of C satisfies the behavioral specification of I.

Specifications are verified by (REQDER) or (REQNOTDER). If a method specification follows from previously proven specifications of the method, the specification is discarded by (REQDER). Otherwise, (REQNOTDER) leads to the analysis of a proof outline for the method. In such proof outlines, external calls $\{r\}\ x : I.m()\ \{s\}$ are handled by (EXTCALL), which ensures that (p,q) follows from the specification $spec(I,m)$ of m in I, and internal calls by (LATECALL), which ensures that the method definitions to which the call may be bound satisfy the requirement (p,q).

5 Example

In this section we illustrate our approach by a small account system implmented by two classes: *PosAccount* and a subclass *FeeAccount*. The example illustrates how interface encapsulation and the separation of class inheritance and subtyping facilitate code reuse. Class *FeeAccount* reuses the implementation of *PosAccount*, but the type of *PosAccount* is not supported by *FeeAccount*. Thus *FeeAccount* does not represent a behavioral subtype of *PosAccount*.

A system of communication components can be specified in terms of the observable interaction between the different components [7, 16]. In the object-oriented setting with interface encapsulation, the observable interaction of an object is described by the *communication history*, which is a sequence of invocation and completion messages of the methods declared by the interface. At any point in time, the communication history abstractly captures the system state. Previous work [11] illustrates how the observable interaction and the internal implementation of an object can be connected. Expressing pre- and postconditions to methods declared by an interface in terms of the communi-

cation history allows abstract specifications of objects supporting the interface. For this purpose, we assume an auxiliary variable h of type $Seq[Msg]$, where Msg ranges over invocation and completion (return) messages to the methods declared by the interface. For the below example, however, it suffices to consider only completion messages. A history h is constructed as a sequence of completion messages by the empty (ε) and right append ($\vdash$) constructor. We write completion messages on the form $\langle o, m(\bar{x}, r)\rangle$ where m is a method completed on object o, $\bar{x}$ is the actual parameter values for this method execution, and r is the return value. For reasoning purposes, such a completion message is implicitly appended to the history at each method termination, as a side effect of the return statment.

5.1 Class *PosAccount*

Interface *IPosAccount* supports the three methods *deposit*, *withdraw*, and *getBalance*. The current balance of the account is abstractly captured by the function $Val(h)$ defined below, and the three methods maintain $Val(h) \geq 0$. Method *deposit* deposits an amount as specified by the parameter value and returns the current balance after the deposit, and method *getBalance* returns the current balance. Method *withdraw* returns *true* if the withdrawal succeeded, and *false* otherwise. A withdrawal succeeds only if it leads to a non-negative balance. In postconditions we let **return** denote the returned value.

```
interface IPosAccount {
  int deposit(nat x) : (Val(h) ≥ 0,  return = Val(h) ∧ return ≥ 0)
  bool withdraw(nat x) : (Val(h) ≥ 0 ∧ h = h0,  return = Val(h0) ≥ x ∧ Val(h) ≥ 0)
  int getBalance() : (Val(h) ≥ 0,  return = Val(h) ∧ return ≥ 0)
}
```

where

$$
\begin{array}{ll}
Val(\varepsilon) & \triangleq 0 \\
Val(h \vdash \langle o, deposit(x,r)\rangle) & \triangleq Val(h) + x \\
Val(h \vdash \langle o, withdraw(x,r)\rangle) & \triangleq \textbf{if } r \textbf{ then } Val(h) - x \textbf{ else } Val(h) \textbf{ fi} \\
Val(h \vdash \textbf{others}) & \triangleq Val(h)
\end{array}
$$

This interface is implemented by class *PosAccount* given below. The balance is maintained by a variable *bal*, and the corresponding invariant expresses that the balance equals $Val(h)$ and remains non-negative. Notice that the invariant $bal = Val(h)$ connects the state of *PosAccount* objects to their observable behavior, and is needed in order to ensure the postconditions declared in the interface.

```
class PosAccount implements IPosAccount {
  int bal = 0;
  int deposit(nat x) : (true,  return = bal) {update(x); return bal}
  bool withdraw(nat x) : (bal = b0,  return = b0 ≥ x) {
   if (bal >= x) then update(-x); return true else return false fi}
  int getBalance() : (true,  return = bal) {return bal}
  void update(int v) : (bal = b0 ∧ h = h0,  bal = b0 + v ∧ h = h0) {bal := bal + v}
  inv bal = Val(h) ∧ bal ≥ 0 }
```

Notice that the method *update* is hidden by the interface, which means that this method is not available to the environment, it is used internally only. Also note that

the following simple definition of *withdraw* maintains the invariant of the class as it preserves $bal = Val(h)$:

```
bool withdraw(int x) {return false}
```

However, this implementation does not meet the interface specification which requires that the method must return *true* if the withdrawal can be performed without resulting in a non-negative balance. Next we consider the verification of class *PosAccount*.

Pre- and postconditions. The pre- and postconditions in the definition of *PosAccount* lead to the following extensions of the S mapping:

$$(true,\ \ \mathbf{return} = bal) \in S(PosAccount, deposit) \quad (1)$$

$$(bal = b_0,\ \ \mathbf{return} = b_0 \geq x) \in S(PosAccount, withdraw) \quad (2)$$

$$(true,\ \ \mathbf{return} = bal) \in S(PosAccount, getBalance) \quad (3)$$

$$(bal = b_0 \wedge h = h_0,\ \ bal = b_0 + v \wedge h = h_0) \in S(PosAccount, update) \quad (4)$$

These specifications are trivially verified over their respective method bodies.

Invariant analysis. Rule (INV) of Fig.3 initiates the analysis of the class invariant wrt. the methods *deposit*, *withdraw* and *getBalance*. By (REQNOTDER), the invariant is remembered as a specification of these methods:

$$(bal = Val(h) \wedge bal \geq 0,\ \ bal = Val(h) \wedge bal \geq 0) \in S(PosAccount, m), \quad (5)$$

for $m \in \{deposit, withdraw, getBalance\}$. Methods *deposit* and *withdraw* perform self calls to *update*, which result in the following two requirements:

$$\begin{array}{l} R(PosAccount, update) = \{ \\ \quad (bal = Val(h) \wedge bal \geq 0 \wedge v \geq 0,\ \ bal = Val(h) + v \wedge bal \geq 0), \\ \quad (bal = Val(h) \wedge v \leq 0 \wedge bal + v \geq 0,\ \ bal = Val(h) + v \wedge bal \geq 0)\} \end{array} \quad (6)$$

These requirements are proved by entailment from equation (4).

Interface specifications. At last, we must verify that the implementation of each method defined by interface *IPosAccount* satisfies the corresponding interface specification, according to (INTSPEC). For *getBalance*, it can be proved that the method specification, as given by (3) and (5), entails the interface specification

$$(Val(h) \geq 0,\ \ \mathbf{return} = Val(h) \wedge \mathbf{return} \geq 0)$$

Verification of the other two methods follows the same outline, and this concludes the verification of class *PosAccount*.

5.2 Class *FeeAccount*

Interface *IFeeAccount* resembles *IPosAccount*, as the same methods are supported. However, *IFeeAccount* takes an additional *fee* for each successful withdrawal, and the balance is not guaranteed to be non-negative. For simplicity we take *fee* as a (read-only) parameter of the interface and of the class (which means that it can be used directly in the definition of *Fval* below).

```
interface IFeeAccount(nat fee) {
 int deposit(nat x) : (N(h), return = Fval(h) ∧ N(h))
 bool withdraw(nat x) : (N(h) ∧ h = h0, return = Fval(h0) ≥ x ∧ N(h))
 int getBalance() : (N(h), return = Fval(h) ∧ N(h)) }
```

where

$$
\begin{array}{ll}
N(h) & \triangleq Fval(h) \geq -fee \\
Fval(\varepsilon) & \triangleq 0 \\
Fval(h \vdash \langle o, deposit(x,r)\rangle) & \triangleq Fval(h) + x \\
Fval(h \vdash \langle o, withdraw(x,r)\rangle) & \triangleq \textbf{if } r \textbf{ then } Fval(h) - x - fee \textbf{ else } Fval(h) \textbf{ fi} \\
Fval(h \vdash \textbf{others}) & \triangleq Fval(h)
\end{array}
$$

Note that *IFeeAccount* is not a subtype of *IPosAccount*: a class that implements *IFeeAccount* will not implement *IPosAccount*. Informally, this can be seen from the postcondition of *withdraw*. For both interfaces, *withdraw* returns true if the parameter value is less or equal to the current balance, but *IFeeAccount* takes an additional fee in this case, which possibly decreases the balance to $-fee$.

Given that the implementation provided by class *PosAccount* is available, it might be feasible to reuse the code of this class when implementing *IFeeAccount*. In fact, only method *withdraw* needs reimplementation, which is illustrated by class *FeeAccount* below. This class implements *IFeeAccount* and extends the implementation of *PosAccount*, which means that the interface supported by the superclass is not supported by the subclass. Typing restrictions will prohibit that methods on an instance of *FeeAccount* is called through the superclass interface *IPosAccount*.

```
class FeeAccount(int fee) extends PosAccount implements IFeeAccount {
 bool withdraw(nat x) : (bal = b0, return = b0 ≥ x) {
  if (bal >= x) then update(-(x+fee)); return true else return false fi}
 inv bal = Fval(h) ∧ bal ≥ −fee
}
```

Pre- and postconditions. As the methods *deposit* and *getBalance* are inherited without redefinition, the specifications of these methods can be relied on also when reasoning about these methods. Especially, specifications (1), (3), and (4) are still valid. For *withdraw*, the declared specification can be proved:

$$(bal = b_0,\ \ \textbf{return} = b_0 \geq x) \in S(FeeAccount, withdraw) \tag{7}$$

Invariant verification. The subclass invariant can be proved over the inherited methods *deposit* and *getBalance* in addition to the overridden method *withdraw*. For *deposit*, the following requirement on *update* is included in the requirement mapping:

$$(bal = Fval(h) \wedge bal \geq -fee \wedge v \geq 0,\ \ bal = Fval(h) + v \wedge bal \geq -fee) \in R(FeeAccount, update)$$

This requirement is entailed by the already proven specification (4) of *update*. Analysis of *withdraw* gives the following requirement, which is also entailed by (4):

$$(bal = Fval(h) \wedge bal \geq x \wedge x \geq 0 \wedge v = -(x+fee),\ bal = Fval(h) - (x+fee) \wedge bal \geq -fee) \in R(FeeAccount, update)$$

Interface specification. Consider again the method *getBalance*. After analysis of subclass invariant, the specification of *getBalance*, given by $S{\uparrow}(\mathit{FeeAccount}, \mathit{getBalance})$, is as follows:

$$\begin{array}{l} S{\uparrow}(\mathit{FeeAccount}, \mathit{getBalance}) = \\ \quad \{(\mathit{bal} = \mathit{Val}(h) \wedge \mathit{bal} \geq 0, \;\; \mathit{bal} = \mathit{Val}(h) \wedge \mathit{bal} \geq 0), \\ \quad\; (\mathit{true}, \;\; \texttt{return} = \mathit{bal}), \\ \quad\; (\mathit{bal} = \mathit{Fval}(h) \wedge \mathit{bal} \geq -\mathit{fee}, \;\; \mathit{bal} = \mathit{Fval}(h) \wedge \mathit{bal} \geq -\mathit{fee})\} \end{array} \tag{8}$$

which is the specification set that can be assumed to prove the interface specification

$$(\mathit{Fval}(h) \geq -\mathit{fee}, \;\; \texttt{return} = \mathit{Fval}(h) \wedge \mathit{Fval}(h) \geq -\mathit{fee}) \tag{9}$$

Specification (9) can be proved by entailment from (8) using (IntSpec). Note that the superclass invariant is not established by the precondition of (9), which means that the inherited invariant cannot be assumed when establishing the postcondition of (9). The other inherited specification is however needed, expressing that **return** equals *bal*. Verification of the interface specifications for *deposit* and *withdraw* then follows the same outline.

6 Discussion

The notion of *behavioral* subtyping, i.e., to require substitutability not just for static object signatures but also for object behavior, goes back to America [2] and Liskov and Wing [19]. It both has been adopted and developed futher and at the same time criticised as being too restrictive to reflect the situation in actual class hierarchies. For example, Wehrheim has studied variations of behavioral subtyping characterized by diffent notions of testing in the context of CSP processes [28]. Recent advances in program development platforms [5, 8] and in theorem proving technology for program verification [6] make the development of more fine-grained systems for incremental reasoning interesting, as a tool is able to collect and often automatically discharge proof obligations during program development.

Related to the concept of behavioral subtyping is the notion of *refinement*. In an object-oriented setting, Back, Mikhajlova, and von Wright propose class refinement and use the refinement calculus to reason about substitutability for object-oriented programs [4]. Similarly, Utting [27] and Mikhajlova and Sekerinsky [21] deal with modular reasoning for object-oriented programs using the refinement calculus.

Putting the emphasis not on how to avoid reverification on the client-side of a method call, but for the designer of a derived class, Soundarajan and Fridella [26] separate two different specifications for each class, an *abstract* specification for the clients of the class and a *concrete* specification for the derived subclasses. Like the current work, they aim at relaxing behavioral subtyping and especially separating subclassing/inheritance from behavioral subtyping to gain flexibility while maintaining proof reuse. Lately, incremental reasoning, both for single and multiple inheritance, has been considered in the setting of *separation logic* [20, 9, 23]. These approaches support a distinction between static specifications, given for each method implementation, from

dynamic specifications that are used to verify late bound calls. The dynamic specifications are given at the declaration site, in contrast to our work where late bound calls are verified based on call-site requirements. Ideas from behavioral subtyping have also been used to support modular reasoning for aspect-oriented programming [10] and for active objects [15].

We propose *lazy* behavioral subtyping to relax some of the restrictions for incremental development of classes imposed by behavioral subtyping, namely to require preservation only for those properties actually needed for client-side verification of methods. The system is syntax-driven and should be possible to integrate in program development platforms. In this paper, lazy behavioral subtyping has been integrated with interface encapsulation, allowing code reuse internally while relying on behavioral interfaces for external method calls. This combination ensures not only that analysis is incremental but also that analysis is modular; i.e., a class needs only be considered once during the program analysis. Revisiting a class for further analysis, which was necessary in previous work on lazy behavioral subtyping, is thereby avoided.

We have illustrated the approach by a simple bank example where the subclass does not represent a behavioral subtype of the superclass. Reuse of code and reuse of proofs are demonstrated. At the same time, client-side reasoning may fully exploit the different properties of the two classes, due to the presence of behavioral interfaces. In future work we plan to investigate the possibilities of letting interfaces also influence the reasoning of self calls in a more fine-grained manner, with the aim of obtaining even weaker requirements to redefinitions. The extension to multiple inheritance could follow the approach of [13].

Acknowledgements. The authors are indepted to Bernd Krieg-Brückner, Ole-Johan Dahl, and Neelam Soundarajan for cooperation and inspiration, the two latter for their insight in object-orientation and formal methods, and Bernd Krieg-Brückner for his insight on program specification and progam verification, and for championing the importance of this field.

References

1. ACM. *37th Annual Symposium on Principles of Programming Languages (POPL)*, Jan. 2008.
2. 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.
3. K. R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.
4. R.-J. Back, A. Mikhajlova, and J. von Wright. Class refinement as semantics of correct object substitutability. *Formal Aspects of Computing*, 12(1):18–40, 2000. Also as Turku Center of Computer Science, TUCS Technical Report No. 333, March 2000.
5. 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.

6. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.
7. M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Monographs in Computer Science. Springer, 2001.
8. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
9. W.-N. Chin, C. David, H.-H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In POPL'08 [1], pages 87–99.
10. C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *SPLAT 2003: Software engineering Properties of Languages for Aspect Technologies at AOSD 2003*, Mar. 2003. Available as Computer Science Technical Report TR03-01a from ftp//:ftp.cs.iastate.edu/pub/techreports/TR03-01/TR.pdf.
11. J. Dovland, E. B. Johnsen, and O. Owe. Observable Behavior of Dynamic Systems: Component Reasoning for Concurrent Objects. *Electronic Notes in Theoretical Computer Science*, 203(3):19–34, 2008.
12. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. In J. Cuellar and T. Maibaum, editors, *Proc. 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *LNCS*, pages 52–67. Springer, May 2008.
13. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Incremental reasoning for multiple inheritance. In M. Leuschel and H. Wehrheim, editors, *Proc. 7th International Conference on Integrated Formal Methods (iFM'09)*, LNCS. Springer, Feb. 2009. To appear.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.
15. N. Hameurlain. Behavioural subtyping and property preservation for active objects. In B. Jacobs and A. Rensink, editors, *Proceedings of the Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, volume 209, pages 95–110. Kluwer, 2002.
16. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.
17. 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.
18. 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.
19. 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.
20. C. Luo and S. Qin. Separation logic for multiple inheritance. *ENTCS*, 212:27–40, 2008.
21. A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97. Industrial Benefits of Formal Methods*, volume 1313 of *LNCS*, pages 82–101. Springer, 1997.
22. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.
23. M. J. Parkinson and G. M. Biermann. Separation logic, abstraction, and inheritance. In POPL'08 [1].
24. C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 343(3):413–442, 2005.
25. Z. Qian and B. Krieg-Brückner. Typed object-oriented functional programming with late binding. In P. Cointe, editor, *Proc. 10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *LNCS*, pages 48–72. Springer, 1996.

26. 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.
27. M. Utting. *An Object-oriented Refinement Calculus with Modular Reasoning*. PhD thesis, University of New South Wales, Australia, 1992.
28. H. Wehrheim. Behavioral subtyping relations for active objects. *Form. Methods Syst. Des.*, 23(2):143–170, 2003.