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 extending the class hierarchy gradually. 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. This paper combines 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. Separating classes and interfaces encapsulates the objects in a cleaner way and leads to a simplification of the formal reasoning system. The approach is presented using a simple object-oriented language (based on Featherweight Java) extended with interfaces and illustrated by an example using a Hoare-style proof system.

1 Introduction

Object-orientation supports an incremental style of program development in which 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 associated with a late bound method call depends on the run-time class of the callee object, and 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 [15], but pose a challenge for program analysis. The intricacies of late-binding, inheritance, encapsulation, and other advanced features of object-oriented languages spawned a lot of research to clarify the semantical foundations, and especially to capture such features in a type-safe manner. This led to the development of expressive type systems and calculi. In this line of research, Qian and Krieg-Brückner developed a language combining

* 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).
features of object-oriented and functional languages in a unified, typed calculus [26],
where much of the expressive power is needed to capture late-binding and overloading.

The static, type-theoretic foundations of mainstream object-oriented languages are
now largely understood, but verification support for object-oriented languages still poses
significant 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 [20] and later used in, e.g., Spec♯ [19].
This is an open world approach: it facilitates reasoning about programs in an incre-
mental 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 super-
class should also hold for all subclasses. The idea is appealing, as it provides substi-
tutability not just for the static signatures of objects, as in standard subtyping, but also
at the behavioral level. Behavioral subtyping, however, imposes severe restrictions on
subclassing, limiting how code may be reused in ways which break with programming
practice [27]. For example, the class hierarchies of Java libraries do not obey the be-
havioral 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 [25] have proposed a complete proof system for object-oriented programs
which is able to address much more flexible code reuse. 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.
The approach focuses on the required properties of a type. It overcomes the limitations
of behavioral subtyping, but breaks with incremental reasoning.

Lazy behavioral subtyping has been proposed by the authors 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 prop-
erties 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 fu-
ture 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 [13]. This inference sys-
tem, which is independent from the underlying specific program logic of a given reason-
ning 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 [14].

These previous papers present a slightly simplistic version of lazy behavioral sub-
typing, concentrating 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 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, the soundness of the proof system of this paper follows directly from the soundness of the pure lazy behavioral subtyping method (see [13]).

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. Section 3 presents the data structures needed to keep track of the different proof obligations, and Section 4 the inference system to analyse a class hierarchy. The method is illustrated on an example in Section 5 and Section 6 discusses the context of this work.

2 The Programming Language

We consider a programming language based on Featherweight Java [18], but extended with (behavioral) interfaces. A program $P$ consists of a set $K$ of interfaces, a set $L$ 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 $I$ of superinterfaces, and declare a set $MS$ 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) $I$. 

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 $\pi$. 

\[ 
\begin{align*}
P & ::= K L \{ t \} \\
MS & ::= m (x) : (p,q) \\
L & ::= \text{class } C \text{ extends } C \text{ implements } I \{ f M MS \text{ inv } p \} \\
M & ::= m (x) \{ t \} \\
e & ::= f \mid \text{this } b \mid \text{new } C() \mid e . m(\pi) \mid m(\pi) \\
t & ::= f := e \mid \text{return } e \mid \text{skip } \mid \text{if } b \text{ then } e \text{ else } f \text{ fi } \mid : t 
\end{align*} \]
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 its behavior. Only the methods declared by I are available for external invocations on references typed by I. 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 a superclass, and define fields f, methods M, and method specifications MS (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 \( \text{inv } p \), 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 \( C \), the class invariant is also hidden. 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. In the imperative language, expressions \( e \) include program variables \( f \) and Boolean expressions \( b \), external calls \( e.m(\overline{x}) \), and self calls \( m(\overline{x}) \). If \( m \) does not return a value, or if the returned value is of no concern, we may use \( e.m(\overline{x}) \) and \( m(\overline{x}) \) directly 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 \( \text{this} \) (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.

A user given specification \( m(\overline{x}) : (p,q) \) applies to the definition of \( m \) in \( C \) or to the inherited definition of \( m \) (without redefinition) from the superclass of \( C \). Thus, a subclass may provide additional specifications of methods inherited from superclasses. For convenience, we use the syntax \( m(\overline{x}) : (p,q) \{ t \} \) to both define and specify \( m \).
3 Class Analysis

An essential part of the verification of a class is to ensure that the methods defined by the class support the behavioral specification of the interface implemented by the class. Methods are verified in terms of proof outlines \(^{23}\), i.e., \(m(x) : \{(p,q)\} t\) such that \(t\) is a method body decorated with pre/post requirements on method calls and \(\vdash_{PL} \{(p) t \{q\}\}\) 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 in a definition of \(m\) in \(C\), or inherited from a superclass of \(C\).

**Notation.** Given assertions \(p\) and \(q\), the type \(APair\) ranges over assertion pairs \((p, q)\). If \(p\) is the pre- and \(q\) the postcondition to some method, the pair \((p, q)\) is a specification of that method. For an interface \(I\), \(public(I)\) denotes the set of method identifiers supported by \(I\); thus, \(m \in public(I)\) if \(m\) is declared by \(I\) or by a supertype of \(I\). A subtype cannot remove methods declared by a supertype, so \(public(I) \subseteq public(I')\) if \(I' \subseteq I\). If \(m \in public(I)\), the function \(spec(I, m)\) returns 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 \(I' \subseteq I\), then \(spec(I, m) \subseteq spec(I', m)\). Consequently, entailment (denoted \(\rightarrow\)) is defined between sets of assertion pairs.

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

\[
\begin{align*}
\text{i)} & \quad (p, q) \rightarrow (r, s) \triangleq (\forall z. \ p \Rightarrow q) \Rightarrow (\forall y. \ r \Rightarrow s), \\
& \quad \text{where } z \text{ and } y \text{ are the logical variables in } (p, q) \text{ and } (r, s), \text{ respectively} \\
\text{ii)} & \quad U \rightarrow (r, s) \triangleq (\bigwedge_{1 \leq i \leq n} (\forall z_i. \ p_i \Rightarrow q'_i)) \Rightarrow (\forall y. \ r \Rightarrow s') \\
\text{iii)} & \quad U \rightarrow V' \triangleq \bigwedge_{1 \leq i \leq m} U \rightarrow (r_i, s_i).
\end{align*}
\]

The relation \(U \rightarrow (r, s)\) corresponds to Hoare-style reasoning, proving \(\{r\} t \{s\}\) from \(\{p_i\} t \{q_i\}\) for all \(1 \leq i \leq n\), by means of the adaptation and conjunction rules \(^3\). Entailment is reflexive and transitive, and \(V' \subseteq U\) implies \(U \rightarrow V'\).

**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 a 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 a class and a method name as arguments. We assume that for each \((p, q) \in S(C, m)\) there is a proof outline for \(body(C, m)\) such that \(\vdash_{PL} \{(p) body(C, m) \{q\}\}\). For a self call \(\{r\} n(\bar{\pi}) \{s\}\) in the proof outline, the assertion pair \((r, s)\) 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 our knowledge of the definition of method \(n\) in \(C\); i.e., \(S(C, n) \rightarrow R(C, n)\). If \(n\) is later overridden by some subclass \(D\) of \(C\), the same requirements apply to the new version of \(n\); i.e., \(S(D, n) \rightarrow R(C, n)\) must be proved. This yields an incremental reasoning strategy.
In general, we let $S(C, m)$ 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 $S(C, m) \cup S(C, \text{inh}, m)$, which captures that 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 $R(C, m)$ such that $R(C, m) = R(C, m) \cup R(C, \text{inh}, m)$.\footnote{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.} For each class $C$ and method $m$ defined in $C$, the lazy behavioral subtyping calculus (see Sec. 3) maintains the relation $S(C, m) \Rightarrow R(C, m)$.

If $C$ implements an interface $I$, the class defines (or inherits) an implementation of each $m \in \text{public}(I)$. For each such method, the behavioral specification declared by $I$ must follow from the method specification, i.e., $S(C, m) \Rightarrow \text{spec}(I, m)$. 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 $\text{spec}(I, m) \Rightarrow (r, s)$. 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 $I' = C.\text{impl}$. Type analysis then ensures that $I' \preceq I$. During the analysis of $C$, the relation $S(C, m) \Rightarrow \text{spec}(I', m)$ is established. The desired $S(C, m) \Rightarrow \text{spec}(I, m)$ then follows since $\text{spec}(I, m) \subseteq \text{spec}(I', m)$ 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 for 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 \footnote{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.}, 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.

4 The Inference System

Classes and interfaces are analyzed in the context a proof environment. The proof environment tracks the specifications and requirements for the different classes, and the interface specifications to which each class must adhere. Let $\text{Cid}, \text{Iid}$, and $\text{Mid}$ denote the types of class, interface, and method names, respectively.

Definition 2 (Proof environments). A proof environment $E$ of type $\text{Env}$ is a tuple $(L, K, S, R)$ where $L : \text{Cid} \rightarrow \text{Class}$, $K : \text{Iid} \rightarrow \text{Interface}$ are partial mappings and $S, R : \text{Cid} \times \text{Mid} \rightarrow \text{Set}[\text{APair}]$ are total mappings.
Subscripts are used to refer to a specific environment; e.g., $S_E$ is the $S$-mapping of $E$. Now, environment soundness is defined. The definition is adapted from [13] by taking interfaces into account. Condition below captures interface implementations, requiring that each method satisfies the behavioral specification given by the interface.

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

1. $\forall (p, q) \in S_E(C, m) \cdot \exists body_E(C, m) . \vdash_{PL} \{ p \} body(C, m) \{ q \}$
   $\land \forall \{ r \} n \{ s \} \in body_E(C, m) . R_E(C, n) \Rightarrow (r, s)$
   $\land \forall \{ r \} x: I, n \{ s \} \in body_E(C, m) . spec(I, n) \Rightarrow (r, s)$
2. $S^+_E(C, m) \Rightarrow R^+_E(C, m)$
3. $\forall n \in public(I) . S^+_E(C, n) \Rightarrow spec(I, n)$, where $I = C.impl$.

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:

$$E \oplus extL(C, D, I, \overline{T}, \overline{M}) = \langle L_E(C \Rightarrow \langle D, I, \overline{T}, \overline{M} \rangle), K_E, S_E, R_E \rangle$$
$$E \oplus extK(I, \overline{L}, \overline{S}) = \langle L_E, K_E[I \Rightarrow \langle \overline{L}, \overline{S} \rangle], S_E, R_E \rangle$$
$$E \oplus extS(C, m, (p, q)) = \langle L_E, K_E, S_E[(C, m) \Rightarrow S_E(C, m) \cup \{(p, q)\}], R_E \rangle$$
$$E \oplus extR(C, m, (p, q)) = \langle L_E, K_E, S_E, R_E[(C, m) \Rightarrow R_E(C, m) \cup \{(p, q)\}]$$

In the calculus, judgments have the form $E \vdash \mathcal{A}$, where $E$ is the proof environment and $\mathcal{A}$ is a sequence of analysis operations (see Fig. 2). The main inference rules are given in Fig. 2. The operations and the calculus are discussed below. We emphasize the differences wrt. the calculus in [13], which reflect the introduction of interfaces and class invariants. In the rules, $I \in E$ and $C \in E$ denote that $K_E(I)$ and $L_E(C)$ are 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 $anReq(\overline{M})$ (see the premise of (NEWCLASS)) generalizes the analysis of a single method done in (NEWMID). The omitted rules may be found in [13] and are similar to the decomposition rules (DECOMPINT) and (DECOMPENV) 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 overriding. 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
I \notin \mathcal{E} \quad I \neq \text{nil} \Rightarrow I \in \mathcal{E} \quad \mathcal{E} \oplus \text{extK}(I, 1, \mathcal{M}_S) \vdash \mathcal{P} \quad \text{(NEWINT)}

I \in \mathcal{E} \quad C \notin \mathcal{E} \quad D \neq \text{nil} \Rightarrow D \in \mathcal{E} \quad \mathcal{E} \oplus \text{extL}(C, D, I, 1, \mathcal{M}_S) \vdash \langle C : \text{anSpec(\mathcal{M}_S) \cdot anReq(\mathcal{M}) \cdot inv(p, public_E(I)) \cdot intSpec(public_E(I)) \cdot \mathcal{P} \rangle \quad \text{(NEWCLASS)}

\mathcal{E} \vdash \langle \text{class C extends D implements I \{f \mathcal{M}_S \text{ inv p} \}} \rangle \cdot \mathcal{P} \quad \text{(NEWCLASS)}

\mathcal{E} \vdash \langle \text{verify(m, Rmp_{\mathcal{E}}(C, \text{inh} m))} \rangle \cdot \mathcal{P} \quad \text{(NEWMTD)}

\mathcal{E} \vdash \langle C : \text{verify(m, (p, q))} \rangle \cdot \mathcal{P} \quad \text{(NEWSPEC)}

\mathcal{E} \vdash \langle C : \text{verify(m, (p, q))} \rangle \cdot \mathcal{P} \quad \text{(REQDER)}

\mathcal{E} \vdash \langle \text{verify(m, (p, q))} \rangle \cdot \mathcal{P} \quad \text{(REQNOTDER)}

\mathcal{E} \vdash \langle C : \text{verify(m, (p, q))} \rangle \cdot \mathcal{P} \quad \text{(LATECALL)}

\mathcal{E} \vdash \langle C : \text{anOutln(p, m, \{q\})} \rangle \cdot \mathcal{P} \quad \text{(EXTCALL)}

\mathcal{E} \vdash \langle C : \text{intSpec(m, \{q\})} \rangle \cdot \mathcal{P} \quad \text{(INTSPEC)}

\mathcal{E} \vdash \langle C : \text{intSpec(m, \{q\})} \rangle \cdot \mathcal{P} \quad \text{(INV)}

\mathcal{E} \vdash \langle C : \text{inv(p, m, \{q\})} \rangle \cdot \mathcal{P} \quad \text{(DECOMPINV)}

\mathcal{E} \vdash \langle C : \text{inv(p, m, \{q\})} \rangle \cdot \mathcal{P} \quad \text{(DECOMFIN)}

\mathcal{E} \vdash \langle C : \text{intSpec(m, \{q\})} \rangle \cdot \mathcal{P} \quad \text{(DECOMFIN)}

\mathcal{E} \vdash \mathcal{P} \quad \mathcal{E} \vdash \langle C : \epsilon \rangle \cdot \mathcal{P} \quad \text{(EMPCLASS)}

Fig. 3. The inference system, where \( \mathcal{P} \) is a (possibly empty) sequence of classes and interfaces. For simplicity, \( m \) denotes a method call including actual parameters and \( \text{nil} \) the empty list.

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
⟨C : O⟩, where O is a sequence of analysis operations to be performed for class C. Note that (NEWINT) and (NEWCLASS) cannot be applied while a ⟨C : O⟩ operation is analyzed, which ensures that ⟨C : O⟩ is analyzed before a new class is analyzed. A successful analysis of C yields an operation ⟨C : ε⟩ which is discarded by (EMPCLASS).

For a class C implementing an interface I, (NEWCLASS) generates four initial operations anSpec, anReq, inv, and intSpec. Operation anSpec is analyzed by (NEWSPEC), initiating a verification of the user given specifications with regard to their respective implementations. For each method m defined in C, anReq collects the inferred requirements toward m which are analyzed in (NEWMD). 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 new proof outline for the method. In such proof outlines, external calls \{r \in I.m \} \{s\} are handled by (EXITCALL), which ensures that \(p, q\) follows from the specification spec(I,m) of m in I. Internal calls are handled 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 implemented 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 communicating components can be specified in terms of the observable interaction between the different components \[?\[?\]. In an object-oriented setting with interface encapsulation, the observable interaction of an object may be 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 \[12\] 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 communication history allows abstract specifications of objects supporting the interface. For this purpose, we assume an auxiliary variable h of type \textit{Seq}[Msg] (sometimes called a mythical program variable \[?\], where \textit{Msg} ranges over invocation and completion (return) messages to the methods declared by the interface. However, for the example below it suffices to consider only completion messages. Thus, a history h will be constructed as a sequence of completion messages by the empty (ε) and right append (·) constructor. We write completion messages on the form ⟨o,m(\(\tau\),r)⟩ where m is a method completed on object o, \(\tau\) 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 statement.
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) ≥ 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

\[
\begin{align*}
Val(ε) & \triangleq 0 \\
Val(h \cdot (o, deposit(x, r))) & \triangleq Val(h) + x \\
Val(h \cdot (o, withdraw(x, r))) & \triangleq \text{if } r \text{ then } Val(h) - x \text{ else } Val(h) \text{ fi} \\
Val(h \cdot \text{others}) & \triangleq Val(h)
\end{align*}
\]

This interface is implemented by a 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 internal 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. This means that the method is not available to the environment, but only for internal use. 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:

\[
\begin{align*}
    & (true, \text{ return } = bal) \in S(\text{PosAccount, deposit}) \quad (1) \\
    & (bal = b_0, \text{ return } = b_0 \geq x) \in S(\text{PosAccount, withdraw}) \quad (2) \\
    & (true, \text{ return } = bal) \in S(\text{PosAccount, getBalance}) \quad (3) \\
    & (bal = b_0 \land h = h_0, \ bal = b_0 + v \land h = h_0) \in S(\text{PosAccount, update}) \quad (4)
\end{align*}
\]

These specifications are easily verified over their respective method bodies.

**Invariant analysis.** Rule \( (\text{INV}) \) of Fig[3] initiates the analysis of the class invariant wrt. the methods deposit, withdraw, and getBalance. By \( (\text{REQNOTDER}) \), the invariant is treated as a specification of these methods:

\[
(bal = Val(h) \land bal \geq 0, \ bal = Val(h) \land bal \geq 0) \in S(\text{PosAccount, } m), \quad (5)
\]

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

\[
R(\text{PosAccount, update}) = \\
\{ \ (bal = Val(h) \land bal \geq 0 \land v \geq 0, \ bal = Val(h) + v \land bal \geq 0), \quad (6) \\
(bal = Val(h) \land v \leq 0 \land bal + v \geq 0, \ bal = Val(h) + v \land bal \geq 0) \ \}
\]

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 \( (\text{INTSPEC}) \). For getBalance, it can be proved that the method specification, as given by Specifications[3] and [5] entails the interface specification

\[
(Val(h) \geq 0, \text{ return } = Val(h) \land \text{ return } \geq 0)
\]

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).

```java
interface IFeeAccount(nat fee) { 
    int deposit(nat x) : (N(h), return = Fval(h) \land N(h)) 
    bool withdraw(nat x) : (N(h) \land h = h_0, return = Fval(h_0) \geq x \land N(h)) 
    int getBalance() : (N(h), return = Fval(h) \land N(h)) 
}
```
where

\[
\begin{align*}
N(h) & \triangleq Fval(h) \geq -fee \\
Fval(\epsilon) & \triangleq 0 \\
Fval(h \cdot (o, deposit(x, r))) & \triangleq Fval(h) + x \\
Fval(h \cdot (o, withdraw(x, r))) & \triangleq \text{if } r \text{ then } Fval(h) - x - fee \text{ else } Fval(h) \text{ fi} \\
Fval(h \cdot \text{others}) & \triangleq Fval(h)
\end{align*}
\]

Note that IFeeAccount is not a behavioral 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 withdraw needs reimplementation, which is illustrated by the 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 prohibit that methods on an instance of FeeAccount are called through the superclass interface IPosAccount.

```java
class FeeAccount(int fee)
    extends PosAccount implements IFeeAccount {
    bool withdraw(nat x) : (bal = b0, return = b0 \geq x) {
        if (bal >= x) then update(-(x+fee));
        return true else return false fi
    }

    inv bal = Fval(h) \land bal \geq -fee
}
```

Pre- and postconditions. As the methods deposit and getBalance are inherited without redefinition, the specifications of these methods can be relied on in the context of the subclass. Especially, Specifications (1) and (3) above remain valid. For withdraw, the declared specification can be proved:

\[
(bal = b0, \text{ return } = b0 \geq x) \in S(FeeAccount, withdraw) \tag{7}
\]

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:

\[
(bal = Fval(h) \land bal \geq -fee \land v \geq 0, \ bal = Fval(h) + v \land bal \geq -fee) \\
\in R(FeeAccount, update)
\]

This requirement follows from Specification (4) of update. Analysis of withdraw gives the following requirement, which also follows from Specification (4):

\[
(bal = Fval(h) \land bal \geq x \land x \geq 0 \land v = -(x+fee), \ bal = Fval(h) - (x+fee) \land bal \geq -fee) \\
\in R(FeeAccount, update)
\]
Interface specification. Now reconsider \( \text{getBalance} \). After analyzing the subclass invariant, the specification of \( \text{getBalance} \) is given by \( S^\uparrow(\text{FeeAccount, getBalance}) \), defined as follows:

\[
S^\uparrow(\text{FeeAccount, getBalance}) = \\
\{( \text{bal} = \text{Val}(h) \land \text{bal} \geq 0, \text{bal} = \text{Val}(h) \land \text{bal} \geq 0), \\
(\text{true, return} = \text{bal}), \\
(\text{bal} = \text{Fval}(h) \land \text{bal} \geq -\text{fee}, \text{bal} = \text{Fval}(h) \land \text{bal} \geq -\text{fee})\}
\]

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

\[
(\text{Fval}(h) \geq -\text{fee}, \text{return} = \text{Fval}(h) \land \text{Fval}(h) \geq -\text{fee})
\]

Specification \( 9 \) can be proved by entailment from Specification \( 8 \) using (INTSPEC). Note that the superclass invariant is not established by the precondition of Specification \( 9 \), which means that the inherited invariant cannot be assumed when establishing the postcondition of Specification \( 9 \). 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 [20]. Since then, it has both been adopted and developed further, and at the same time criticised for being too restrictive to reflect the situation in actual class hierarchies. For example, Wehrheim has studied variations of behavioral subtyping characterized by different notions of testing in the context of CSP processes [29]. 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 [28] and Mikhajlova and Sekerinsky [22] 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 [27] separate two 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 [9, 21, 24]. These approaches support a distinction between static specifications, given for each method implementation, and dynamic specifications
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 [16].

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 combined 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 need only be considered once during the program analysis. Revisiting a class for further analysis, which could happen in previous work on lazy behavioral subtyping, is thereby avoided.

To use our system, the programmer is expected to provide specifications of interfaces and classes (by means of invariants or pre/post pairs), such that for each class the specifications given entail the corresponding interface specifications. All other specifications and requirements are generated by the system. We have illustrated the approach by a simple bank example where the subclass does not represent a behavioral subtype of the superclass. Both the reuse of code and the reuse of proofs are demonstrated in the example. 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. Furthermore, the extension of the work presented in this paper to multiple inheritance could follow the approach of [14].

Acknowledgements. The authors are indebted 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 in program specification and program verification, and for championing the importance of formal methods in computer science.

References

Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04), volume 3362 of LNCS, pages 49–69. Springer, 2005.


15. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Mass., 1995.


18. 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.


