9-3-2006

Behavioral Subtyping, Specification Inheritance, and Modular Reasoning

Gary T. Leavens
Iowa State University

David A. Naumann
Iowa State University

Behavioral Subtyping, Specification Inheritance, and Modular Reasoning

Abstract
Behavioral subtyping is an established idea that enables modular reasoning about behavioral properties of object-oriented programs. It requires that syntactic subtypes are behavioral refinements. It validates reasoning about a dynamically-dispatched method call, say E.m(), using the specification associated with the static type of the receiver expression E. For languages with references and mutable objects the idea of behavioral subtyping has not been rigorously formalized as such, the standard informal notion has inadequacies, and exact definitions are not obvious. This paper formalizes behavioral subtyping and supertype abstraction for a Java-like sequential language with classes, interfaces, exceptions, mutable heap objects, references, and recursive types. Behavioral subtyping is proved sound and semantically complete for reasoning with supertype abstraction. Specification inheritance, as used in the specification language JML, is formalized and proved to entail behavioral subtyping.

Keywords
Behavioral subtyping, supertype abstraction, specification inheritance, modularity, specification, verification, state transformer, dynamic dispatch, invariants, Eiffel language, JML language

Disciplines
Software Engineering

Behavioral Subtyping, Specification Inheritance, and Modular Reasoning

Gary T. Leavens and David A. Naumann

TR #06-20b
July 21, 2006, revised Aug 4, Sept. 3 2006

Keywords: Behavioral subtyping, supertype abstraction, specification inheritance, modularity, specification, verification, state transformer, dynamic dispatch, invariants, Eiffel language, JML language.

2006 CR Categories: D.2.2 [*Software Engineering*] Design Tools and Techniques — Object-oriented design methods; D.2.3 [*Software Engineering*] Coding Tools and Techniques — Object-oriented programming; D.2.4 [*Software Engineering*] Software/Program Verification — Class invariants, correctness proofs, formal methods, programming by contract, reliability, tools, Eiffel, JML; D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement — Documentation; D.3.1 [*Programming Languages*] Formal Definitions and Theory — Semantics; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; D.3.3 [*Programming Languages*] Language Constructs and Features — classes and objects, inheritance; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, logics of programs, pre- and post-conditions, specification techniques;

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Behavioral Subtyping, Specification Inheritance, and Modular Reasoning

Gary T. Leavens *
Iowa State University
Ames, IA 50011 USA
leavens@cs.iastate.edu

David A. Naumann †
Stevens Institute of Technology
Hoboken, NJ 07030 USA
naumann@cs.stevens.edu

Abstract

Behavioral subtyping is an established idea that enables modular reasoning about behavioral properties of object-oriented programs. It requires that syntactic subtypes are behavioral refinements. It validates reasoning about a dynamically-dispatched method call, say $E.m()$, using the specification associated with the static type of the receiver expression E. For languages with references and mutable objects the idea of behavioral subtyping has not been rigorously formalized as such, the standard informal notion has inadequacies, and exact definitions are not obvious. This paper formalizes behavioral subtyping and supertype abstraction for a Java-like sequential language with classes, interfaces, exceptions, mutable heap objects, references, and recursive types. Behavioral subtyping is proved sound and semantically complete for reasoning with supertype abstraction. Specification inheritance, as used in the specification language JML, is formalized and proved to entail behavioral subtyping.

1. Introduction

In object-oriented (OO) programming, subtyping and dynamic dispatch are both useful and problematic. They are useful because supertypes can abstract away details in the specifications of their subtypes, thus allowing variations in data structures and algorithms to be handled uniformly. They are problematic for modular reasoning because a dynamically-dispatched method call such as $E.m()$ seems to require a case analysis to deal with all possible dynamic types of E's value. The basic idea of modular reasoning, which we call supertype abstraction [22], is clear. It is a generalization of typechecking: reasoning about an invocation, say $E.m()$, is based on the specification associated with the static type of E, and constraints are imposed on implementations of m at all subtypes. While modular type safety conditions for dynamically-dispatched methods are well-known [12], a straightforward translation into conditions on overriding method specifications, while sound, is more restrictive than necessary. The translation also gives no help in reasoning about object invariants, which need to be strengthened in subtypes. Hence, for modular reasoning one needs a behavioral notion of subtyping.

Remarkably, there is no mathematically rigorous account of behavioral subtyping and its connection with modular reasoning about specifications and programs in conventional OO programming languages — although there has been much study [1, 2, 3, 11, 14, 15, 19, 22, 29, 30, 47] (see [21] for a survey). Some of the current understanding of behavioral subtyping is embodied in program logics [31, 40, 42, 43, 46] but is difficult to disentangle from various other complications. Some of the current understanding is embodied languages and tools such as Eiffel [30], JML [20], ESC/Java [16], and Spec# [8, 7]. But these have unsoundnesses and incompletenesses, some by engineering design and some for lack of adequate theory and methodology. A key source of unsoundness is naive treatment of object invariants, because many important design patterns invalidate the simple hierarchical notion of encapsulation on which the standard treatment [17] is based.

On one hand, behavioral subtyping has been rigorously studied in restrictive programming models (e.g., [19, 47]). On the other hand, various embodiments have been implemented in static and runtime verification tools and logics that apply to rich specification and programming languages such as Java and JML [20] and Eiffel [30]. Our goal is to close the gap by providing a rigorous analysis on which can be based more specialized assessments and justifications of specific tools and logics. (With the ultimate aim of high assurance for verification tools, we have undertaken to machine-check our results.)

We believe the gap has remained because it was far from clear how to formalize a general theory that pertains directly to reasoning about code in languages of practical interest. The semantic intricacies of the languages —and of current methodologies for sound reasoning about invariants, heap encapsulation and locality of effects, etc.— are daunting. The details of the OO language are important, because some language features, such as reflection, allow programs to make observations that can distinguish between supertype and subtype objects. The achievements closest to our aim are soundness and completeness proofs for logics of Java fragments that embody supertype abstraction in some form (e.g.,[43]). But these assess the reasoning power of a proof system, rather than assessing and explicating the connection between behavioral subtyping and supertype abstraction; and they are somewhat removed from the axiomatic semantics of some widely used verifiers (which simply postulate soundness of behavioral subtyping in some form).

The key insight that led to our results is a purely semantic formulation of supertype abstraction using two denotational semantics: in one, method calls are statically dispatched. On this basis we are able to give a formal treatment in a language with many constructs of sequential OO languages, including classes, interfaces, mutable heap objects, assignment, exceptions, inheritance, visibility, reference equality, type tests, and recursive types. Even with effective definitions in hand, it was nontrivial to find the right induction hypothesis and technique to prove the main lemma.

We make no commitment to particular specification notations or reasoning system but rather formulate modular reasoning semantically in a generic way that idealizes what is found in logics and tools. Using an operationally sound compositional semantics allows us to provide a foundation that will serve as a point of reference and as a basis for assessment and further development of specification languages and verification tools.

This paper makes the following contributions.

- We give a semantic characterization of supertype abstraction, which idealizes what is found in logics and verification tools. In contrast to related work, our definition does not rely on derived notions such as substituting one object for another [19, 27, 29], nor is it tied to a proof system [31, 40, 42, 43, 46].

* Supported in part by NSF grant CCF-0429567.

† Supported in part by NSF grants CCR-0208984 and CCF-0429894.

- We formalize behavioral subtyping in terms of refinement of observable behavior in a realistic programing model. Refinement does not need to hold between all syntactically related types but only when the subtype is a (non-abstract) class.
- In contrast to the standard view [29], we define refinement intrinsically, by quantifying over satisfying implementations. Separately, we characterize refinement in terms of relations between pre- and postconditions. Our characterization adapts previous work [13, 33, 41] that improves on the overly restrictive standard condition of postcondition implications [2, 3, 15, 29, 30]. (This also isolates the way in which characterization of completeness depends on the specification language, see Sect. 7.) An outcome of our focus on reasoning about correctness of programs rather than an abstract model is that we find abstraction functions are not an integral part of behavioral subtyping (compare, e.g., [19, 29]).
- We prove soundness and semantic completeness of behavioral subtyping for supertype abstraction. It was by seeking completeness that we were led to the surprising findings noted in the preceding items.
- We justify the standard condition on invariants [29], by precisely describing the necessary soundness conditions and how they can be enforced by recently proposed techniques for sound modular reasoning about invariants [24, 37, 32].
- We formalize specification inheritance [50] and show that it ensures behavioral subtyping. Specification inheritance is part of the semantic definition of JML and it embodies the proof obligations whereby some logics achieve behavioral subtyping [31, 40, 42, 43, 46].

The rest of this paper is organized as follows. The next section gives a synopsis of the main ideas. We then discuss related work. Following that are technical details about the language (Sect. 3) and specifications (Sect. 4). This is followed by a formalization of supertype abstraction and behavioral subtyping (Sect. 5), with the main theorems connecting them. We then extend these results to handle invariants (Sect. 6). In Sect. 7 we characterize behavioral subtyping by a checkable condition on specifications and prove that specification inheritance ensures behavioral subtyping. Sect. 8 concludes.

2. Synopsis

In an OO language it is not obvious how to do modular reasoning, because dynamic dispatch selects different methods depending on the exact run time type of an object. What specification should one use to reason about a call, such as $E.m()$, given that the static type of E, say T, is only an upper bound on its dynamic type? While we consider formal specifications and reasoning in this paper, the problem also applies to informal reasoning based on informal specifications. The first expert OO programmers used T's specification of m to reason about such calls. This kind of reasoning, *supertype abstraction* [22], is modular in that it does not depend on E's dynamic type, and hence does not have to be changed when subtypes of T are changed in compatible ways or are added to a program [27]. Supertype abstraction supports maintenance and evolutionary programming styles.

However, supertype abstraction is only valid if methods that override T's method m satisfy T's specification for m, since that specification is the one used in reasoning about such calls. Making overrides obey the specification of overridden methods, *specification inheritance* [14, 50], ensures that objects of subtypes of T do not cause surprising behavior when treated as if they are objects of type T; that is, it ensures *behavioral subtyping* [2, 3]. An alternative is to check the given specifications and treat violations as a design error [15]. Either way, a characterization is needed for soundness and to avoid unnecessary restriction.

2.1 Supertype abstraction

An influential discussion of the benefits of these ideas is Liskov's invited talk at OOPSLA 1987 [27]. Liskov stated an easily-remembered test for behavioral[1] subtyping (p. 25): "If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T." This is often called the "Liskov Substitutability Principle" (LSP) and is a strong form of supertype abstraction. The LSP is actually too strong, because it uses the notion of "unchanged" behavior; the point of introducing subtype objects is often to change behavior in a way that is allowed by the supertype's specification. A more flexible intuition defines observations that are not allowed by this specification as "surprising," and says that behavioral "subtyping prevents surprising behavior" [19, Chapter 1].

As a formulation of supertype abstraction, the LSP is not easy to apply to imperative OO languages. It is not clear what it means to substitute one object for another: imperative programs are not referentially transparent, object identity matters, and the state of "an object" often depends on other objects in the heap. (Object identity is also a problem with the algebraic work on applicative languages from which the LSP is drawn [11, 19].) One of the contributions of our paper is to precisely formalize supertype abstraction in a way that captures modular reasoning about imperative OO programs.

Our definition of supertype abstraction says that properties of a command can be proved by reasoning about its method calls as if they were statically dispatched to an arbitrary implementation that satisfies the specification associated with the static type of the receiver. The definition is in terms of a denotational semantics. For a command S, its meaning, $\mathcal{D}[\![S]\!]$, is interpreted in a *method environment*, μ, that gives a meaning to each method at each type. Thus $\mathcal{D}[\![S]\!]\mu$ is a function from initial states to final states. We are interested in proving that S satisfies some pre/post specification, *spec*. Modular reasoning proves that $\mathcal{D}[\![S]\!]\mu$ satisfies *spec*, using only the specifications associated with the static types of receivers of method calls. We call the latter a *specification table*. To avoid formalizing "reasoning" as such, our definition considers semantic consequences that hold for any μ that satisfies the specification table. Whereas the actual program semantics, $\mathcal{D}[\![-]\!]$, uses dynamic dispatch, we define a static dispatch semantics, $\mathcal{S}[\![-]\!]$, to formalize reasoning in terms of static types. Supertype abstraction is formalized roughly as follows: if it is true that $\mathcal{S}[\![S]\!]\mu$ satisfies *spec*, then the actual semantics $\mathcal{D}[\![S]\!]\mu$ satisfies *spec* (provided that μ satisfies the specification table). (For details see Def. 11.[2])

In a program logic, supertype abstraction is embodied by the proof rule for method invocation, which allows to derive $\{P\}\ E.m()\ \{Q\}$ only from a specification $(pre_m^T, post_m^T)$ associated with the static type, T, of E. Similarly, an automated verifier typically uses weakest precondition semantics and achieves modularity by replacing a call $E.m()$ by the sequence "**assert** pre_m^T; **assume** $post_m^T$" (with various optimizations, e.g., [23]). Both techniques aim to produce sound conclusions about the actual semantics. We model both by the static-dispatch semantics $\mathcal{S}[\![E.m()]\!]\mu$. What makes both techniques sound is behavioral subtyping, imposed by proof obligations on implementations of m in subtypes of T (typically via some form of specification inheritance). The proof obligations are modeled by our specification table; behavioral subtyping, i.e., refinement of specifications, is a property of the specification table.

The main result of the paper (Thms. 12, 24) says that supertype abstraction for commands is equivalent to behavioral subtyping. The proof that behavioral subtyping implies supertype abstraction goes by induction over the syntax of the language and is the paper's hardest technical result (Lemma 16).

2.2 Behavioral subtyping

Several authors have offered definitions of behavioral subtyping, but the most influential definition has been Liskov and Wing's [29]. We paraphrase part of their "constraint rule" (from their Figure 4, page 1823).[3] For type T to be a behavioral subtype of U:

[1] The quote refers to what we call "behavioral subtyping" simply as "subtyping."

[2] We resist the temptation to abstract the specification table as a single method environment that uses nondeterminacy to represent a specification by the "least refined implementation"; though elegant and useful (e.g, [39, 43]), this technique requires justification with respect to the actual program semantics.

[3] Their rule allows for method renamings and an abstraction function, our semantics of OO languages does not allow the former and allows us to disregard the latter. We also ignore the part of the rule about history constraints, since we study a sequential language.

```
interface Tracker {
  public model goal, curr: int;
  public invariant 0<self.goal ∧ self.goal≤self.curr;
  // ...
  meth lose(kg: int)
    requires self.goal ≤ self.curr − kg;
    ensures exc = null ∧ self.curr = old(self.curr − kg);
}

class WeightLoss extends Object implements Tracker {
  protected g, c: int;
  protected invariant 0<self.g ∧ self.g≤self.c;
  protected represents goal := self.g;
  protected represents curr := self.c;
  // ...
  meth lose(kg: int)
    requires true;
    ensures (old(self.goal ≤ self.curr − kg)
               ⇒ (exc = null ∧ self.curr = old(self.curr−kg)))
          ∧ (old(self.goal > self.curr − kg)
               ⇒ (exc≠null));
  { if (self.g≤self.c−kg) then self.setCurr(self.c−kg)
    else throw new IAE()
  }
  meth setCurr(kg: int)
    requires self.goal≤kg;
    ensures exc = null ∧ self.curr = kg;
  { self.curr := kg }
}
```

Figure 1. The interface Tracker and the class WeightLoss.

- The subtype's invariant must imply the supertype's. That is, whenever T's invariant holds for a subtype object[4], then U's invariant holds:

$$inv^T(\mathsf{self}) \;\Rightarrow\; inv^U(\mathsf{self}) \quad \text{for all } \mathsf{self}: T. \tag{1}$$

- "Subtype methods preserve the supertype method's behavior." That is, if T's method m overrides U's method m, then the usual static typing conditions [12] hold, and for all subtype objects $\mathsf{self}: T$, U's precondition for m implies T's precondition:

$$pre_m^U(\mathsf{self}) \;\Rightarrow\; pre_m^T(\mathsf{self}) \tag{2}$$

and T's postcondition implies U's:

$$post_m^T(\mathsf{self}) \;\Rightarrow\; post_m^U(\mathsf{self}). \tag{3}$$

The above definition is intended to be part of a "descriptive and informal" presentation (p. 1813), which concentrates on ideas and has only "informal justifications" (p. 1813). However, even at a conceptual level, it has several problems.

Liskov and Wing's postcondition rule (3) is stronger (i.e., less flexible) than necessary for the soundness of supertype abstraction [14]. To see this, consider Fig. 1. The interface Tracker declares two model fields, goal and curr that are only used in specifications [25]. They stand for the goal of a diet and the current weight. The values of these model fields are given by the represents clauses in class WeightLoss. The clause "**represents** goal := self.g" declares part of the object invariant for WeightLoss, which says that goal = self.g. This is a predicate on the protected field and the inherited public model field. This is typical of JML and other specification languages, where the connection between a data representation and an abstraction are expressed by a hidden invariant. Frame axioms (modifies clauses) could be encoded in the postcondition, but for brevity we ignore them in our examples.

The lose method of the class WeightLoss illustrates the problem with the postcondition rule (3). That rule requires that lose's postcondition implies the postcondition in Tracker. However, this implication does not hold for this example, as can be seen by considering the case where goal is strictly larger than the difference curr − kg, since exc≠**null** contradicts exc = **null**. (The specification variable exc refers to exception results; when exc is null in a post-state, the method returned normally.) However, in this example, supertype abstraction works fine, because the lose method of the class WeightLoss obeys Tracker's specification for lose whenever Tracker's precondition holds. Clients that reason about calls to lose using Tracker's specification will not be surprised, WeightLoss can be a behavioral subtype of Tracker. One way of expressing the most flexible sound rule for U to be a behavioral subtype of T is to require that for all subtype objects $\mathsf{self}: U$, the precondition rule (2) holds and [13, 33, 41]:

$$old(pre_m^T(\mathsf{self})) \wedge post_m^U(\mathsf{self}) \;\Rightarrow\; post_m^T(\mathsf{self}). \tag{4}$$

where $old(P)$ refers to the value of predicate P in the pre-state of a call. The idea is that the behavior of the subtype's method does not need to be constrained when the supertype's precondition does not hold in the pre-state.

These conditions are just approximations of refinement and it is refinement that explains the equivalence between behavioral subtyping and supertype abstraction (Theorem 12). Our formulation of behavioral subtyping (Def 8) is in terms of the intrinsic refinement order on specifications. In Sect. 7 we confirm that (4) characterizes refinement in our setting and can thus be used to check behavioral subtyping. Disentangling the two is important since the characterization is sensitive to the form of specifications as discussed in Sect. 7.

2.3 Invariants and behavioral subtyping

While Liskov and Wing's postcondition rule is sound but too restrictive, their invariant rule is unsound if applied to general OO programs, unless further restrictions on programs are applied. Object invariants are predicates that can be assumed in method pre-states and must be established in method post-states [17]. Liskov and Wing's discussion implies that "the invariant" is what the programmer declares and reasons about. This is broken in unrestricted OO languages, because of sharing and reentrance [31, 32, 34, 38]. We illustrate these problems below. We also review recent methodologies for sound reasoning about invariants; in brief, the methodologies derive from the declared invariants an effective invariant for which Liskov and Wing's invariant rule does work.

To explain the problems, we begin by considering how invariants interact with pre- and postconditions. Invariants could be thought of as conjoined to the pre- and postcondition of each method's specification, because invariants must hold at the beginning and end of each method's execution; this is the usual proof obligation. This conjunction semantics would cause a conflict between the invariant rule (1) and the precondition rule (2), as the invariant may be strengthened in a subtype (covariantly), which will strengthen a precondition it is conjoined with, which in turn violates the rule that preconditions may only be weakened in a subtype (contravariantly). The standard solution to this problem [17] ensures that the implementation can assume the invariant as a precondition, by restricting the invariant to depend only on part of the state that is within an encapsulation boundary, say private to the class, so that it cannot be falsified by clients between calls to methods of the class. For example, consider the invariant in Fig. 1; if field c in Fig. 1 were public, instead of protected, then clients could assign -1 to it, falsifying the invariant. Indeed, one would like per-instance encapsulation, so that an invariant for object o can only be invalidated by a method execution whose receiver is o. But privacy in Java-like languages is at the level of classes, not instances, and in this example field c is visible in subclasses. The general rule is that if a mutable field is visible in some context, say a client or subclass, then that code must be responsible for maintaining invariants that depend on the field (and therefore such invariants must be visible).

Encapsulation is partly achieved using scope and visibility, but that is not enough, due to the problem of sharing mutable objects. between an object's representation and client code. This is the problem of representation exposure [28, 32, 38]. To see the problem, imagine that c in Fig. 1, instead of containing an integer, referred to a mutable IntCell object:

protected c: IntCell;

Imagine the IntCell objects contain a (model) field val that can be changed in a program and that the invariant is changed to refer to it:

[4] Liskov and Wing state the invariant rule for object "values," not for objects.

```
class WghtDiff extends WeightLoss {
  protected d: int;
  protected invariant 0<self.g ∧ self.g≤self.c
                      ∧ self.d=self.c−self.g;
  // ...
  meth setCurr(kg: int)
    requires self.goal≤kg;
    ensures exc = null ∧ self.curr = kg;
  { super.setCurr(kg);
    // invariant should hold, but doesn't
    self.d := self.c−self.g
  }
}
```

Figure 2. The class WghtDiff.

protected invariant 0<self.g ∧ self.g≤self.c.val;

The problem is that the IntCell object c can be shared between this version of the WeightLoss class and clients. This allows clients to mutate c.val, potentially violating the invariant. Liskov and Wing's rule is thus only sound if objects cannot contain mutable subobjects, or if some other restrictions on ownership are applied [24, 37, 32].

Reentrance is another source of unsoundness for invariants. The class WghtDiff shown in Fig. 2 illustrates the problem. Consider a call such as wd.lose(5), where wd is an object of dynamic type WghtDiff. In this case, the inherited lose method of WeightLoss is called, which reenters WghtDiff by calling its setCurr method. WghtDiff's setCurr method makes a super call up to WeightLoss's setCurr method, which (in the nonexceptional case) sets g and c, potentially breaking WghtDiff's invariant in the last conjunct. This violates WghtDiff's invariant in a visible state, namely the post-state of this super call, as indicated in the code. According to the assumption that all invariants must hold in all visible states, the last assignment in WghtDiff's setCurr method is not necessary, even though operationally it is certainly needed to re-establish the invariant broken by the super call. This illustrates a problem with the simplest view of invariants. But the most difficult problem to solve is reentrant callbacks. The self-call to setCurr in the superclass method is dynamically dispatched to the override in WghtDiff; in this example nothing goes wrong but in general such a call can find the invariant temporarily false because there is already a method invocation (here, lose) in progress.

We expected to need visibility, ownership, etc. to formalize a sound notion of invariants —it was a pleasant surprise that in semantic terms these all amount to ways to ensure that invariants can be assumed as preconditions by method implementations, so they do not complicate our formalization (Def. 20). The effective invariant is governed by (1) and the means by which it is made sound to assume invariants as preconditions is independent from the justification of supertype abstraction (Thm. 24).

Two techniques for ensuring that invariants can be assumed as preconditions in OO languages have been investigated recently. The relevant invariant semantics is based on an ownership type system, which defines the invariants that are "relevant" to a given method [32]; in this technique a method must establish all relevant invariants that might have been broken before making a method call. The Boogie [7, 24, 37] technique uses a dynamic notion of ownership and strong invariants that hold in all states. Declared invariants are allowed to be temporarily broken. To express whether the declared invariant is in effect, Boogie uses a specification-only field, invar, that ranges over the object's supertypes. An object's invariant takes the form

$$\forall T \cdot \mathsf{self.invar} \leq T \Rightarrow inv^T(\mathsf{self}) \qquad (5)$$

where inv^T is the invariant explicitly declared for type T. During temporary violations of the declared invariant for T, this field is set to a supertype of T. Thus the methodology ensures that (5) holds in every reachable state. What matters is that these techniques provide invariants, derived from the declared ones, that can soundly be assumed as a precondition in verifying a method implementation.

2.4 Enforcing behavioral subtyping by specification inheritance

As can be seen from our examples, writing the specifications for a behavioral subtype often involves a certain amount of repetition. For example, the postcondition of the lose method in class WeightLoss (Fig. 1) has two conjuncts, and the first of these repeats both the precondition and the postcondition of the overridden method in the supertype Tracker (Fig. 1). A similar repetition happens for invariants, as can be seen in the invariant of class WghtDiff (Fig. 2), whose first two conjuncts repeat the invariant of its supertype WeightLoss.

Such repetition can be avoided using specification inheritance [14, 50]. In JML, the programmer declares two specifications, one for the supertype, and one for the subtype. The effective specification is given by specification inheritance as described below. For invariants, the extension of the subtype's specification produces the conjunction of its declared invariant with the conjunction of the invariants from its supertypes. This would allow the invariant of WghtDiff to be stated as follows:

protected invariant self.d=self.c − self.g;

which avoids repeating the first two conjuncts shown in Fig. 2. For methods, an overriding method declaration inherits the specifications of each declaration of that method that it overrides. Its effective specification is the join (least upper bound) of the explicit specification with the specifications of that method in all supertypes. In JML the join of a method specification with the specifications of overridden methods is indicated by the keyword **also**. Using this convention one could write the specification of the lose method in class WeightLoss as follows:

```
meth lose(kg: int)
  also
    requires self.goal > self.curr − kg;
    ensures exc≠null;
```

As we explain in Sect. 7, the meaning of the above specification under specification inheritance is essentially the same as the specification given in Fig. 2. That is, the meaning has as its precondition the disjunction of the explicit precondition and those inherited (which in this example is **true**), and as its postcondition a conjunction of implications, which says that if a precondition was satisfied, then the corresponding postcondition must hold. Theorem 30 says specification inheritance forces behavioral subtyping. A number of tools and logics enforce behavioral subtyping through equivalent means, though it is not always explicit.

2.5 Related work

Liskov and Wing's paper [29] has already been discussed above. Their paper is famous because they clearly present the main ideas and several interesting examples. Liskov and Wing formulate something like supertype abstraction, their "Subtype Requirement" (p. 1812), but it is sketched in terms of provability and does not directly address modular reasoning about code and method contracts. They present informal arguments why behavioral subtyping ensures their subtype requirement. The reasoning they permit involves only invariants and history constraints, apparently because the model of computation allows concurrency. By contrast, we use a sequential language that allows Hoare-style reasoning using invariants as well as pre- and postconditions.

Leavens and Dhara [21] survey a lot of older work on behavioral subtyping, including the pioneering work of America [2, 3] and Meyer [30]. Much of it is similar to Liskov and Wing's and has similar limitations.

Several logics have been given for sequential fragments of Java which incorporate supertype abstraction [31, 42, 43, 46]. These logics mostly achieve behavioral subtyping by requiring that each overriding method implementation in a type satisfies the corresponding specification in each of its supertypes. This has the same effect as our definition of specification inheritance, which explicitly constructs the "effective specifications". Some prove soundness and even completeness of a proof system with behavioral subtyping, which justifies supertype abstraction in their setting. Of these, only Müller's [31] considers interfaces, and even this misses our insight that interfaces can be exempted from the requirements of behavioral subtyping that must apply to (non-abstract) classes. Müller concentrates on a modular treatment of frame axioms (modifies clauses) and invariants using ownership.

Parkinson's work [42] is based on separation logic [39] and encapsulates invariants using a nonstandard form of opaque predicate [10] which can be seen as higher order quantification [9]. Parkinson says "behavioral subtyping" for the standard implications (2) and (3) and "specifi-

cation compatibility" for a proof-theoretic formulation of the intrinsic refinement ordering [42, Def. 3.5.1]. Pierik [43] gives a more conventional proof system, in particular a proof outline logic with a first-order assertion language using finite sequences for heap expressions. As we do in Sect. 7, Pierik explicitly connects specification refinement with adaptation rules. Supertype abstraction and behavioral subtyping are present but intertwined with many other details. Pierik and Parkinson both prove soundness and Pierik proves completeness. But the relation between a logic and its models does not address our objective of explicating the connection between behavioral subtyping and supertype abstraction.

Pierik and de Boer [44] investigate various notions of completeness of logics in the presence of behavioral subtyping.

One reason it is important to have disentangled behavioral subtyping from specification inheritance is that it illuminates an important design decision for specification languages. Many of the above cited works force behavioral subtyping by using specification inheritance. This decision prevents redundancy in specifications and avoids the need for subtypes to see all specifications in supertypes, some of which may be invisible (such as private invariants). The downside of this decision is that the join used in specification inheritance can lead to unsatisfiable specifications. Worse yet it can mask design mistakes.

Findler and Felleisen take the opposite decision [15], i.e., checking for behavioral subtyping rather than imposing it by fiat with specification inheritance. Their work is a foundational study of runtime assertion checking for pre/post specifications without invariants. Their contract checker can detect violations of behavioral subtyping as embodied in Liskov and Wing's pre/post rules, (2) and (3). They note that violations of the precondition rule (2) do not occur in a specification language that constructs effective preconditions by using disjunction, as in specification inheritance. So runtime checkers for such languages cannot detect violations of the precondition rule.

3. Language

The technical development uses an idealized object-oriented language that models a large fragment of the class-based languages like Java and C#. It includes interfaces, mutually recursive classes, exceptions as first-class objects, type test and casts, inheritance and dynamic binding. It does not model: concurrency, nested classes, and reflection including dynamic class loading, to avoid complications. Constructors are also omitted, to keep the language small. For issues like evaluation order and the semantics of null casts, where reasonable languages may differ on semantic details, we follow Java. The semantics is adapted from [6] which in turn draws on [18] for formalization of syntax including the class table.

Readers familiar with conventional languages should have little difficulty with the language syntax, but some effort is needed to follow the details of the semantic definitions so we begin with design rationale.

For reasoning about specifications, a "big step" or state transformer semantics is appropriate, since specifications describe behavior in terms of initial and final states of commands. Although state transformer semantics can be defined from a small-step semantics, it is more convenient to give a direct definition that is compositional, to support proof by induction on the structure of commands. Nonetheless, our model uses a standard concrete model of state and the semantic definitions for expressions and commands are operational and unsurprising.

A key aspect of modular reasoning is that reasoning about a command is based on the specifications of methods invoked by the command — independent from any particular implementations of those methods. To model this aspect of modular reasoning without over-specializing our theory by bringing proof theory into the picture, we use a denotational semantics: the meaning of a command or expression is defined with respect to a given *method environment* which provides the denotations of all methods.

Four technical features of the semantics streamline the formal development but may be unexpected. First, although a distinction is made between expressions and commands, both may have effects. Reasoning systems often restrict expressions to be pure; but we include exceptions in full generality which entails heap effects in expressions. Second, threading state through the semantics of expressions adds considerable complication which is mitigated by the uniform use of a general form of *state transformer*, with separate variable declarations for the initial and final state spaces. In the case of a command, the initial and final state space are the same except for the addition of an exception variable (exc) in the final state space. In the case of an expression, the initial state space is as expected, but the final state space consists only of two variables, the exception variable and a variable to hold the normal result (res). In the case of a method, the initial state consists of parameters and the receiver variable self; the final state has exception and normal result variables as in the case of expressions. A state transformer of a given type is a mapping from initial states to final states or $\bot$, the later modeling divergence.

The third feature is the encoding of exceptions. An expression may diverge, yield a normal result, or throw an exception. The semantics uses a disjoint sum of just two kinds of outcome: either $\bot$ or a state. But that state includes a special variable exc, of type Thrwbl, to encode a disjoint sum: the value of exc is either null, which signifies normal termination, or a reference to the exception object (which may contain references to other objects in the heap). Variable exc is not allowed to occur in the program text but is used in specifications.

Fourth, the syntax is in something like "A-normal form" [49], i.e., subexpressions in various constructs are restricted to be variables. To avoid loss of expressiveness, let-expressions are added; desugaring transformations are not difficult to define, e.g., a general equality test $E_1 = E_2$ can be desugared ($-^o$) by the rule

$$(E_1 = E_2)^o = \textbf{let } x \textbf{ be } E_1^o \textbf{ in let } y \textbf{ be } E_2^o \textbf{ in } x = y$$

which preserves order of evaluation and propagation of exceptions.

Notation summary. The metatheory is standard set theory and we often use partial functions (finite maps) treated as sets of pairs. Unqualified, the term *function* means total function. Much of the formalism is expressed in terms of dependently typed functions, the notation for which is deferred until later.

Application is written with juxtaposition and associates to the left as in $f\ a\ b$ for a curried f. Finite mappings are used for typing contexts and variable stores, e.g., $[\mathsf{self} : C, x : \textbf{int}]$ or with brackets omitted. The extension of a finite mapping g to map b to z, where $b \notin dom\ g$, is written $[g,\ b : z]$. To override the mapping for an element $c \in dom\ g$ we write $[g \mid c : z]$. To streamline the semantic definitions, our metalanguage uses $\bot$-strict let-expressions: the value of "let $x = \alpha$ in β" is $\bot$ in case $\alpha = \bot$, otherwise it is as usual. A raised dot separates a variable binding from its scope, as in $\forall x : \textbf{int} \cdot a[x] > 0$.

Syntax. The grammar is based on some given sets of names, using the following nomenclature for typical elements:

$C \in ClassName$	names of declared classes
$I \in InterfaceName$	names of declared interfaces
x, y, f	variable names (for param., fields, and locals)
m	method names

There are two distinguished variable names, self for the receiver object (*this* in Java) and res. The final value of res gives the return value of a method, as if every method body has the form "S; **return** res;". Using res fits with JML and lets us omit return statements. There is one distinguished interface name, Thrwbl, and three distinguished class names Object, NullDeref and ClassCast; the latter implement Thrwbl.

Class and interface declarations have the following forms:

$$\begin{array}{l}\textbf{class } C \textbf{ extends } C \textbf{ implements } \overline{I}\ \{\ \overline{vis\ f : T}\quad \overline{mdec}\ \} \\ \textbf{interface } I \textbf{ extends } \overline{I}\ \{\ \overline{vis\ f : T}\quad \overline{msig}\ \}\end{array}$$

Here and throughout we use over-lines to indicate sequences, possibly empty. Instance fields are included in interfaces since they are needed in specifications. In a specification language they would be designated as "ghost" or "model" fields and our results apply to programs that are properly annotated for ghost fields, but the distinction is not needed for our results.

The remaining syntactic categories are as follows. (Bold keywords and punctuation marks including "{" and "}" are terminal symbols.)

T	::=	$C \mid I \mid$ **bool** $\mid$ **int** $\mid$ **unit**	data type
vis	::=	**private** $\mid$ **protected** $\mid$ **public**	visibility modifier
msig	::=	$m(\overline{x}:\overline{T}):T$	method signature
mdec	::=	**meth** *msig* { S }	method declaration
S	::=	$x := E \mid x.f := x$	assign to var., to field
	$\mid$	**var** $x:T$ **in** S	local variable block
	$\mid$	$S;\ S \mid$ **if** E **then** S **else** S	sequence, conditional
	$\mid$	**throw** x	throw exception
	$\mid$	**try** S **catch**$(x:T)\ S$	try-catch
	$\mid$	**try** S **finally** S	try-finally
E	::=	$x \mid$ **it** $\mid$ **null** $\mid$ **true** $\mid 0\ldots$	variable, constants
	$\mid$	$x.f \mid x = x$	field access, equality test
	$\mid$	x **is** $T \mid (T)\ x$	type test, cast
	$\mid$	**new** $C()$	object construction
	$\mid$	$x.m(\overline{x})$	method call
	$\mid$	**let** x **be** E **in** E	sequenced local binding

For brevity the term *ref type* is used to mean any non-primitive data type, i.e., a class or interface name, and we define

$$\mathit{RefType} = \mathit{ClassName} \cup \mathit{InterfaceName}$$

A complete program is a collection of class and interface declarations. Formally, we consider a *class table* CT which is a mapping sending each class name C to its declaration $CT(C)$ and each interface name I to its declaration $CT(I)$. We eschew the term "statement" but use identifier S for commands since C is mnemonic for class names.

The typing rules are syntax directed. The rules for commands and expressions use judgments in which the variable context is explicit, giving names and types of local variables and parameters that are in scope (namely, the method parameters, any locals in scope, and the special variables self, res). The rules also depend on the complete class table, as is needed to deal with recursive class declarations —this is not made explicit in the judgments.

Various auxiliary notations are used that depend on the class table, again without explicit indication. Suppose $CT(C)$ is

$$\textbf{class}\ C\ \textbf{extends}\ D\ \textbf{implements}\ \overline{I}\ \{\overline{\mathit{vis}\ f:T;}\ \overline{\mathit{mdec}}\}$$

Then we define *super* $C = D$ and *superinterfaces* $C = \overline{I}$. Let mdec be in the list $\overline{\mathit{mdec}}$ of method declarations, so mdec has the form

$$\textbf{meth}\ m(\overline{x}:\overline{T}):T_1\ \{\ S\ \}$$

We record the type and parameter names by defining $\mathit{mtype}(C,m) = \overline{x}:\overline{T}\to T_1$. If m is inherited in C from D (i.e., is defined in D but not declared in C) then $\mathit{mtype}(C,m)$ is defined to be $\mathit{mtype}(D,m)$. Thus $\mathit{mtype}(C,m)$ is defined iff m is declared or inherited in C. Similarly for interfaces: if I extends $\overline{I}$ then *superinterfaces* $I = \overline{I}$ and $\mathit{mtype}(I,m)$ is defined the same as for classes.

For declared fields of the class displayed above we define *dfields* $C = \overline{\mathit{vis}\ f:T}$ and similarly *dfields* I are the fields declared by interface I. To include inherited fields for such a class C we define

$$\mathit{fields}\ C = \mathit{fields}\ D \cup \mathit{dfields}\ C \cup (\cup I \in \mathit{superinterfaces}\ C \cdot \mathit{fields}\ I)$$

In a well formed class table each of these unions will be disjoint. For interfaces define *fields* I = *dfields* $I \cup$ *fields*(*superinterfaces* I). We define $\mathit{Meths}\ T = \{m \mid \mathit{mtype}(T,m)\ \text{is defined}\}$, noting that $\mathit{mtype}(T,m)$ is defined just when T is a ref type and m is declared or inherited in T.

The subtype relation $\leq$ is defined inductively by

- $C \leq D$ if *super* $C = D$
- $T \leq I$ if $I \in$ *superinterfaces* T (in which case T is a ref type)
- $I \leq \mathsf{Object}$ if $I \in \mathit{dom}(CT)$ is an interface
- $\leq$ is reflexive and transitive

Note that for primitive types, $T \leq U$ holds just if T is U.

A class table CT is *well formed* provided it satisfies standard constraints such as acyclicity of $\leq$. Every type in a field declaration, parameter, etc, is one of the primitives or in $\mathit{RefType}$. Every method declaration $m(\overline{x}:\overline{T}):T\ \{S\}$ in $CT(C)$ is typable in the sense that $\Gamma \vdash S$ where $\Gamma = [\mathsf{self}:C, \mathsf{res}:T, \overline{x}:\overline{T}]$. Method overrides must not change the signature. Rules that define $\Gamma \vdash S$ are in Appendix A. We give here just two.

$$\frac{\Gamma \vdash E:T \qquad [\Gamma,\ x:T] \vdash E1:U}{\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ E\ \textbf{in}\ E1:U}$$

$$\frac{\Gamma \vdash x:T \qquad T \in \mathit{RefType} \qquad \mathit{mtype}(T,m) = \overline{z}:\overline{T}\to U \qquad \Gamma \vdash \overline{y}:\overline{V} \qquad \overline{V} \leq \overline{T}}{\Gamma \vdash x.m(\overline{y}):U}$$

3.1 Semantic domains

Details of the semantics for expressions and commands are not very important, although they are needed in some proofs. But familiarity with the semantic domains —especially state transformers and method environments— is essential since specifications are treated semantically.

We assume a given set Ref of *references* —abstract addresses. A *ref context* is a finite partial function ρ that maps references to class names. The idea is that if $o \in \mathit{dom}\ \rho$ then o is allocated and moreover o points to an object of type $\rho\ o$. We define the set $\mathit{RefCtx} = \mathit{Ref} \rightharpoonup \mathit{ClassName}$, where $\rightharpoonup$ denotes partial functions. Note that for ρ and ρ' in *RefCtx*, we can write $\rho \subseteq \rho'$ to express that the domain of ρ' includes at least the objects in ρ and for objects allocated in ρ the types are the same in ρ'.

Semantic domains are the sets of possible meanings for various kinds of phrases such as data types, expressions, and method bodies. The definitions capture important invariants about the semantics, e.g., that the value of a field or variable of a given static type is a value of that type. Moreover, if T is a ref type then its values include references to objects in subclasses. Finally, there are no dangling references. Most of the semantic domains are defined in terms of a given ref context. For example, $\mathit{Val}(T,\rho)$ is the set of values of type T in a state where ρ is the ref context. In case T is a primitive type, the definition of $\mathit{Val}(T,\rho)$ is independent from ρ. But if T is a class type, then $\mathit{Val}(T,\rho)$ is some set containing null and some references —all the allocated objects of some type U such that $U \leq T$.

Here is a guide to the domains. Those marked with $*$ are just special cases of the others as explained below.

domain	description	metavariables	
$\mathit{Val}(T,\rho)$	value of type T		o, v
$\mathit{Store}(\Gamma,\rho)$	stores for Γ		r
$*\ \mathit{Obrecord}(C,\rho)$	fields of C-objects		
Heap(ρ)	heaps		h
$\mathit{State}(\Gamma)$	states for Γ		s, t
$\mathit{STrans}(\Gamma_1,\Gamma_2)$	state transformers		σ, τ
$*\ \mathit{SemExpr}(\Gamma,\ T)$	semantic expression		
$*\ \mathit{SemCommand}(\Gamma)$	semantic command		
$*\ \mathit{SemMeth}(C,m)$	semantics of method m in C		
$\mathit{MethEnv}$	method environment		μ
$\mathit{XMethEnv}$	extended meth. env. (incl. interfaces)		$\dot{\mu}$

For data types T the definition is by cases on T:

$$\begin{array}{lcl} \mathit{Val}(\textbf{bool},\rho) & = & \{\mathit{true}, \mathit{false}\} \\ \mathit{Val}(\textbf{int},\rho) & = & \{\ldots,-2,-1,0,1,2,\ldots\} \\ \mathit{Val}(\textbf{unit},\rho) & = & \{\mathit{it}\} \\ \mathit{Val}(C,\rho) & = & \{\mathit{null}\} \cup \{o \mid o \in \mathit{dom}\ \rho \wedge \rho\ o \leq C\} \\ \mathit{Val}(I,\rho) & = & \{o \mid \exists C \cdot C \leq I \wedge o \in \mathit{Val}(C,\rho)\} \end{array}$$

The next definitions involve dependent function spaces or dependent pairs, for which we use the following.

Notation for dependent types in metatheory

Suppose X is a set and for every $x \in X$ a set Z_x is given. Then the dependent product $(y:X) \times Z_y$ is the set of pairs (x,z) such that $x \in X$ and $z \in Z_x$. The dependent function space $(y:X) \to Z_y$ is the set of functions f from X to $\cup_{x\in X} Z_x$ such that $f\ x \in Z_x$

for all $x \in X$. Note that y is a bound variable in these notations.[5]

The first use is in the definition

$$Store(\Gamma, \rho) = (x : dom\,\Gamma) \to Val(\Gamma\, x, \rho)$$

What this means is that $Store(\Gamma, \rho)$ is a set of functions; and for any r in $Store(\Gamma, \rho)$ (the name r is mnemonic for *store*), the domain of r is $dom\,\Gamma$ and $r\,x$ is an element of $Val(\Gamma\, x, \rho)$ for each $x \in dom\,\Gamma$.

Next we build up to program states.

$$\begin{array}{lcl} Obrecord(C, \rho) & = & Store(\mathit{fields}\ C, \rho) \\ Heap(\rho) & = & (o : dom\,\rho) \to Obrecord(\rho\, o, \rho) \\ State(\Gamma) & = & (\rho : \mathit{RefCtx}) \times Heap(\rho) \times Store(\Gamma, \rho) \end{array}$$

Recall that *obcontext* C is the variable context obtained by removing visibility markers from *fields* C. So a heap h is map sending each allocated reference o to a record, $h\ o$, of the object's current field values.

The most important domain is state transformers:

$$\begin{array}{lcl} STrans(\Gamma, \Gamma') & = & (s : State(\Gamma)) \\ & & \to \{\bot\} \cup \{s' | s' \in State(\Gamma') \wedge extState(s, s')\} \end{array}$$

Relation $extState$ is used to say that one state's ref context extends the other's:[6] $extState((\rho, h, r), (\rho', h', r')) \iff \rho \subseteq \rho'$.
Elements of $STrans(\Gamma_1, \Gamma_2)$ are functions that map a state in $State(\Gamma_1)$ to either $\bot$ or a state in $State(\Gamma_2)$. The domain of state transformers subsumes meanings for methods, expressions and commands.

$$\begin{array}{lcl} SemExpr(\Gamma,\ T) & = & STrans(\Gamma, [\mathsf{res} : T, \mathsf{exc} : \mathsf{Thrwbl}]) \\ SemCommand(\Gamma) & = & STrans(\Gamma, [\Gamma,\ \mathsf{exc} : \mathsf{Thrwbl}]) \\ SemMeth(T, m) & = & STrans(\ [\mathsf{self} : T, \overline{x} : \overline{T}], \\ & & \qquad\qquad [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thrwbl}]) \\ & & \text{where } mtype(m,\ T) = \overline{x} : \overline{T} \to U \end{array}$$

A (normal) *method environment* is defined to be a table of meanings for all methods in all classes:

$$MethEnv = (C : ClassName) \times (m : Meths\ C) \to SemMeth(C, m)$$

The idea is that a method environment μ is defined for pairs (C, m) where C is a class with method m and moreover $\mu(C, m)$ is a state transformer suitable to be the meaning of a method of type $mtype(C, m)$. In case m is inherited in C from B, $\mu(C, m)$ will be the restriction of $\mu(B, m)$ to receiver objects of type C.

The semantics of a command depends on its method environment and a complete program CT denotes a method environment, described later.

In the formulation of modular reasoning based on static types it turns out to be convenient to use an extended kind of method environment which associates method meanings to interfaces as well as to classes (even though the receiver of an invocation is always an object of some class, cf. the definition of $Val(I, \rho)$).

An *extended method environment* is defined to be a table that also includes meanings for methods of interfaces:

$$XMethEnv = (T : RefType) \times (m : Meths\ T) \to SemMeth(T, m)$$

The metavariable μ ranges over normal method environments and $\dot{\mu}$ is used to range over extended method environments —the dot is intended to be mnemonic for the dotted i in interface.

3.2 Semantics of expressions, commands, and class table

The semantic definitions for expressions and commands are straightforward but mostly relegated to Appendix B. We explain just what is needed for the proofs. For an expression $\Gamma \vdash E : T$, the semantics $[\![\Gamma \vdash E : T]\!]$ gets applied to a method environment μ to yield a state transformer $[\![\Gamma \vdash E : T]\!]\mu$ that in turn is applied to a state, e.g., $[\![\Gamma \vdash E : T]\!]\mu(\rho, h, r)$. Similarly for commands. The definition is syntax-directed, e.g., here is the semantics of an assignment:

$$\begin{array}{l} [\![\Gamma \vdash x := E]\!]\mu(\rho, h, r)\ = \\ \quad \mathsf{let}\ (\rho_1, h_1, r_1) = [\![\Gamma \vdash E : T]\!]\mu(\rho, h, r)\ \mathsf{in} \\ \quad \mathsf{if}\ r_1\ \mathsf{exc} = null\ \mathsf{then}\ (\rho_1, h_1, [r \mid x : r_1\ \mathsf{res}, \mathsf{exc} : null]) \\ \quad \mathsf{else}\ (\rho_1, h_1, [r,\ \mathsf{exc} : r_1\ \mathsf{exc}]) \end{array}$$

If E yields $\bot$ then so does the assignment. If E throws no exception then its value is assigned to x and the final state is extended with exc set null. Otherwise, the final state is extended with exc mapped to the exception.

This definition and the ones in the appendix use the notation $[\![-]\!]$ for the semantics function, but this abbreviates two definitions. They are defined in the same way, except in the case of method call. The *dynamic dispatch semantics*, for which we decorate the semantics brackets as $\mathcal{D}[\![-]\!]$, is the operationally accurate one. It dispatches to a method meaning in the method environment based on the dynamic type of the receiver. Let $T = \Gamma\, x$ and $\overline{z} : \overline{T} \to U = mtype(m,\ T)$ as in the typing rule. Define

$$\begin{array}{l} \mathcal{D}[\![\Gamma \vdash x.m(\overline{y}) : U]\!]\mu(\rho, h, r)\ = \\ \quad \mathsf{if}\ r\,x = null\ \mathsf{then}\ except(\rho, h,\ U, \mathsf{NullDeref}) \\ \quad \mathsf{else\ let}\ r_1 = [\mathsf{self} : r\,x, \overline{z} : r\,\overline{y}]\ \mathsf{in}\ \mu(\rho(r\,x), m)(\rho, h, r_1) \end{array}$$

Because the dynamic type of the receiver is a class (specifically, $\rho\,(r\,x)$), this semantics is based on a normal method environment. The helping function $except$ builds a state with exc set to a new NullDeref object (see Appendix B).

The *static dispatch* or nominal [22] semantics of method call applies a method meaning determined by the static type T of the receiver. Since interfaces are included among the static types, the static dispatch semantics is defined in terms of an extended method environment $\dot{\mu}$.

$$\begin{array}{l} \mathcal{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!]\dot{\mu}(\rho, h, r)\ = \\ \quad \mathsf{if}\ r\,x = null\ \mathsf{then}\ except(\rho, h,\ U, \mathsf{NullDeref}) \\ \quad \mathsf{else\ let}\ r_1 = [\mathsf{self} : r\,x, \overline{z} : r\,\overline{y}]\ \mathsf{in}\ \dot{\mu}(T, m)(\rho, h, r_1) \end{array}$$

One of the proofs deals with the case of let-expressions in detail. Here is the semantics, where $[\![-]\!]$ is either $\mathcal{S}[\![-]\!]$ or $\mathcal{D}[\![-]\!]$ throughout.

$$\begin{array}{l} [\![\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ E\ \textbf{in}\ E1 : U]\!]\mu(\rho, h, r)\ = \\ \quad \mathsf{let}\ (\rho_0, h_0, r_0) = [\![\Gamma \vdash E : T]\!]\mu(\rho, h, r)\ \mathsf{in} \\ \quad \mathsf{if}\ r_0\ \mathsf{exc} \neq null\ \mathsf{then}\ (\rho_0, h_0, [\mathsf{res} : \mathit{default}\ U, \mathsf{exc} : r_0\ \mathsf{exc}]) \\ \quad \mathsf{else\ let}\ r_1 = [r,\ x : r_0\ \mathsf{res}]\ \mathsf{in}\ [\![\Gamma \vdash E1 : U]\!]\mu(\rho_0, h_0, r_1) \end{array}$$

The semantics is defined with respect to an arbitrary allocator. An *allocator* is just a choice function for unused locations, i.e., a function $fresh$ that maps a pair (ρ, h), with $h \in Heap(\rho)$, to a location such that $fresh(\rho, h) \notin dom\,\rho$.[7]

Semantics of class table. A well formed class table denotes a method environment, $\widehat{\mu}$, the least upper bound of an ascending chain of method environments—the *approximation chain*—each of which is given using the command semantics for method bodies and the preceding approximation. The ith element in the chain approximates $\widehat{\mu}$ in a way such that, in operational terms, it gives the correct semantics for executions with method call stack bounded in length by i. Details are in Appendix B.

4. Specifications and refinement

This section formalizes method specifications and satisfaction by state transformers. On this basis we define specification tables and satisfaction for them as well as the induced refinement relation between specifications. Refinement is used in the following section to define behavioral subtyping.

From now on it is assumed that CT is some well formed class table.

Specification languages have long used special syntax in postconditions to refer to initial state, such as "old(x)" in JML, so that postconditions are two-state predicates (but see Sect. 7 in regards to auxiliary variables).

Specifications for methods and commands are obviously needed, but expression specifications are also needed, to prove one of the main results. So it is convenient to use the notation of a *state transformer type* $\Gamma \rightsquigarrow \Gamma'$ for specifications of state transformers in $STrans(\Gamma, \Gamma')$.

[5] More standard notations are $\Sigma\, y : X.Z_y$ for $(y : X) \times Z_y$ and $\Pi\, y : X.Z_y$ for $(y : X) \to Z_y$. Ours are similar to the PVS prover's.

[6] Since ρ and ρ' are partial functions which we treat as sets of pairs, $\rho \subseteq \rho'$ says that ρ' has at least the domain of ρ and they agree on their common domain.

[7] As a simple example, Ref can be taken to be the naturals and $fresh(\rho, h)$ can be the least n not in $dom\,\rho$. A realistic allocator depends on program state which is why we include h here.

Definition 1 (state transformer specification) A *specification of type* $\Gamma \rightsquigarrow \Gamma'$ is a pair $(pre, post)$ such that

- pre is a subset of $State(\Gamma)$
- $post$ is a subset of $State(\Gamma) \times State(\Gamma')$

For ref type T, a *method specification of type* (T, m) is a specification of type $[\mathsf{self}: T, \overline{x}: \overline{T}] \rightsquigarrow [\mathsf{res}: U, \mathsf{exc}: \mathsf{Thrwbl}]$ where $mtype(m, T) = \overline{x}: \overline{T} \rightarrow U$. A Γ-*specification* is one of type $\Gamma \rightsquigarrow [\Gamma, \ \mathsf{exc}: \mathsf{Thrwbl}]$.

Specifications are interpreted in the sense of total correctness, which is expressed very simply since $\perp$ is not in $State(\Gamma')$.

Definition 2 (satisfaction by state transformer) Let $(pre, post)$ be a specification of type $\Gamma \rightsquigarrow \Gamma'$ and σ be in $STrans(\Gamma, \Gamma')$. Then σ *satisfies* $(pre, post)$, written $\sigma \models (pre, post)$, iff

$$\forall t \cdot t \in pre \ \Rightarrow \ (t, \sigma\, t) \in post$$

A specification table provides specifications for all methods (we add an invariant for each class in Sect. 6). It models what might be called the "effective specification", which is typically obtained from declared specifications by means of context-dependent interpretation of modifies clauses [26, 31], specification inheritance (see Sect. 7), etc. Roughly speaking, a method environment satisfies a specification table if each method satisfies its specification; formally there are two notions corresponding to the two kinds of method environment.

Definition 3 (specification table) A *specification table*, ST, contains a method specification $ST(T, m)$ of type $mtype(T, m)$ for each ref type T and each $m \in Meths\ T$.

For perspicuity we sometimes write $(pre_m^T, post_m^T)$ for $ST(T, m)$.

Definition 4 (satisfaction by method environment) Let ST be a specification table. An extended method environment $\dot{\mu}$ *satisfies* ST, written $\dot{\mu} \ \models \ ST$, iff $\dot{\mu}(T, m) \ \models \ ST(T, m)$ for all ref types T and $m \in Meths\ T$.

A normal method environment μ *satisfies* ST, written $\mu \models ST$, iff $\mu(C, m) \models ST(C, m)$ for all classes C and $m \in Meths\ C$.

Note this does not require $\mu(C, m)$ to satisfy specifications of the interfaces implemented by C, nor of its superclass. The idea is that $ST(C, m)$ is the actual behavioral condition specified for C, m. In Sect. 7 we define specification inheritance as a means to derive $ST(C, m)$ from declared specifications of C's interfaces and superclasses.

Ordering specifications. The behavioral subtyping property is expressed in terms of a refinement ordering on specifications, saying that if T is a subtype of U then $ST(T, m)$ is a stronger specification than $ST(U, m)$ in the sense that any method satisfying $ST(T, m)$ also satisfies $ST(U, m)$. This intrinsic ordering on specifications is determined by the nature of command denotations and the definition of satisfaction. Some care needs to be taken with the details, because if T is a class, a method in class T is defined to act on receiver objects of type T whereas a specification of type (U, m) imposes a requirement on state transformers for target type U. Owing to the semantics of dynamic dispatch, however, it is sound for a method in class T to assume a strengthened precondition saying that the receiver object has type T. (This is explicit in the proof obligation for method bodies in proof systems, e.g. [31, 45].)

For a method m of class U with $mtype(U, m) = \overline{x}: \overline{T} \rightarrow V$, the relevant state transformers are in $SemMeth(U, m)$, i.e., of type $[\mathsf{self}: U, \overline{x}: \overline{T}] \rightsquigarrow [\mathsf{res}: V, \mathsf{exc}: \mathsf{Thrwbl}]$. For T, a method meaning will have type $[\mathsf{self}: T, \overline{x}: \overline{T}] \rightsquigarrow [\mathsf{res}: V, \mathsf{exc}: \mathsf{Thrwbl}]$ —only the type of self varies.[8] This leads us to define a nonstandard notion of satisfaction. To that end it is convenient to define an operation that converts a predicate to a two-state predicate. For p a set of states, define $old(p)$ by

$$(s, t) \in old(p) \iff s \in p$$

[8] It is well known how to vary parameter and result types; our results can be easily adapted to such variation but we use invariant method overriding as in Java to avoid unenlightening notational complications.

Definition 5 (satisfaction under a type, $\lfloor, \models^T$) Let $(pre, post)$ be a specification of type $\Gamma \rightsquigarrow \Gamma'$ and let $T \leq \Gamma\,\mathsf{self}$. Define $(pre, post)\lfloor T$ to be the specification $(pre', post')$, of type $[\Gamma \mid \mathsf{self}: T] \rightsquigarrow \Gamma'$, where pre' is defined by $(\rho, h, r) \in pre' \iff r\,\mathsf{self} \leq T \wedge (\rho, h, r) \in pre$ and $post' = old(pre') \cap post$.

An element $\sigma \in STrans([\Gamma \mid \mathsf{self}: T], \Gamma')$ *satisfies* $(pre, post)$ *under* T, written $\sigma \models^T (pre, post)$, iff[9] $\sigma \models (pre, post)\lfloor T$

This differs from $\models$ only by strengthening the precondition to restrict to states where the receiver object is in $Vals(T, \rho)$. So $\models^T$ is $\models$ in case $T = \Gamma\,\mathsf{self}$.

Definition 6 (specification refinement, $\sqsupseteq^T$) Let $spec_0$ be a specification of type $\Gamma \rightsquigarrow \Gamma'$ and $spec_1$ be of type $[\Gamma \mid \mathsf{self}: T] \rightsquigarrow \Gamma'$ where $T \leq \Gamma\,\mathsf{self}$. Then $spec_1$ *refines* $spec_0$ *with respect to* T, written $spec_1 \sqsupseteq^T spec_0$, iff

$$\sigma \models spec_1 \ \Rightarrow \ \sigma \models^T spec_0 \ \text{ for all } \ \sigma \in STrans([\Gamma \mid \mathsf{self}: T], \Gamma')$$

Note that σ ranges over the smaller set of transformers and only weakly satisfies $spec_0$. In case $T = \Gamma\,\mathsf{self}$, however, $\sigma \models spec_0$ is the same as $\sigma \models^T spec_0$. We may omit the superscript on $\sqsupseteq$ just in this case.

Note that $\sqsupseteq$ is not antisymmetric: by definition of satisfaction any $(pre, \ old(pre) \cap post)$ satisfies the same specifications as $(pre, post)$.[10]

Lemma 7 (weak transitivity) Suppose $spec_0$ is a specification of type $\Gamma \rightsquigarrow \Gamma'$, $spec_1$ is of type $[\Gamma \mid \mathsf{self}: T] \rightsquigarrow \Gamma'$ where $T \leq \Gamma\,\mathsf{self}$, and $spec_2$ is of type $[\Gamma \mid \mathsf{self}: U] \rightsquigarrow \Gamma'$ with $U \leq T$. If $spec_2 \sqsupseteq^U spec_1$ and $spec_1 \sqsupseteq^T spec_0$ then $spec_2 \sqsupseteq^U spec_0$, provided $spec_1$ is satisfiable.

Proof To show $spec_2 \sqsupseteq^U spec_0$, assume $\sigma \in STrans([\Gamma \mid \mathsf{self}: U], \Gamma')$ and $\sigma \models spec_2$ with the aim to prove $\sigma \models^U spec_0$. From $spec_2 \sqsupseteq^U spec_1$ we get $\sigma \models^U spec_1$; but this does not yield $\sigma \models spec_1$ which isn't even defined. Define $\tau \in STrans([\Gamma \mid \mathsf{self}: T], \Gamma')$ by

$$\tau(\rho, h, r) \ = \ \mathsf{if}\ r\,\mathsf{self} \leq U\ \mathsf{then}\ \sigma(\rho, h, r)\ \mathsf{else}\ s$$

where s is an arbitrarily chosen state that satisfies $spec_1$ for (ρ, h, r). From $\sigma \models^U spec_1$ we get $\tau \models spec_1$. Then by $spec_1 \sqsupseteq^T spec_0$ we get that $\tau \models^T spec_0$. Now $\sigma \models^U spec_0$ follows from $\tau \models^T spec_0$ by definition of τ and $\models$. □

To see that the satisfiability condition is necessary, let $spec_1$ have $pre_1 = true$ and let $post_1$ say that self **is** U. No element of $STrans([\Gamma \mid \mathsf{self}: T], \Gamma')$ satisfies $spec_1$. Owing to unsatisfiability we have $spec_1 \sqsupseteq^T spec_0$ for any $spec_0$. Define $spec_2$ to have $pre_2 = true = post_2$. Then because $\sqsupseteq^U$ restricts the initial state we get $spec_2 \sqsupseteq^U spec_1$. But it is easy to choose $spec_0$ so that $spec_2 \not\sqsupseteq^U spec_0$.

5. Supertype abstraction and behavioral subtyping

This section states and proves the supertype abstraction theorem. Behavioral subtyping is defined in terms of the intrinsic refinement order on method specifications and then an alternative formulation is given. The rest of the section is devoted to stating and proving the main theorem.

The definition of behavioral subtyping is slightly weaker than one might expect, as needed for an equivalence with supertype abstraction.

Definition 8 (behavioral subtyping) A specification table ST has *behavioral subtyping* if and only if for all ref types U and classes C with $C \leq U$ and all $m \in Meths\ U$ we have $ST(C, m) \sqsupseteq^C ST(U, m)$.

If $\geq$ is any preorder relation on some set, an instance $a \geq b$ is equivalent to $\forall c \cdot b \geq c \ \Rightarrow \ a \geq c$. Supertype abstraction for methods is roughly a restatement of behavioral suptyping in this manner, though taking into account the change of type.

Definition 9 (supertype abstraction for method specifications) Specification table ST has *supertype abstraction for method specifications* iff the following holds for all ref types T, all $m \in Meths\ T$, and all

[9] The notation $\models^T$ doesn't fully capture the contextual assumptions but it should suffice to avoid ambiguity in our uses of it.

[10] If desired, antisymmetry can be achieved without loss of generality by normalizing specifications.

method specifications *spec* of type *mtype*(T, m): If $ST(T, m) \sqsupseteq spec$ then $ST(C, m) \sqsupseteq^C spec$ for every class C with $C \leq T$.

Lemma 10 A satisfiable specification table ST has behavioral subtyping iff it has supertype abstraction for method specifications.

Proof For the implication left to right, suppose ST has behavioral subtyping. Consider any pair(T, m) and method specification *spec* for (T, m) such that $ST(T, m) \sqsupseteq spec$ and any C with $C \leq T$. By behavioral subtyping we have $ST(C, m) \sqsupseteq^C ST(T, m)$. Since $ST(T, m)$ is satisfiable, we can apply weak transitivity (Lemma 7) to get $ST(C, m) \sqsupseteq^C spec$.

For the implication right to left, suppose ST has supertype abstraction for method specifications. Consider $C \leq U$ where U is any ref type. Instantiate supertype abstraction with $T := U$ and $spec := ST(U, m)$. Since $ST(U, m) \sqsupseteq^T ST(U, m)$, supertype abstraction yields $ST(C, m) \sqsupseteq^C ST(U, m)$. □

Supertype abstraction for commands. The idea is that a modular reasoning system is a sound means to prove that some command $\Gamma \vdash S$ satisfies some specification *spec*. For the proof to be modular means that reasoning about method calls in S is based only on the specifications of those methods. Thus our semantic formulation says that S satisfies *spec* when S is interpreted by the static dispatch semantics. Of course the static dispatch semantics of a command has many properties that are inconsistent with its standard semantics, so reasoning on the basis of static dispatch semantics with a particular method environment would be unsound. To capture that reasoning about method calls is based only on their specifications, our formulation quantifies over all environments that satisfy ST. Reasoning is expressed in terms of program semantics, so this is a property of a class table together with a specification table.

Definition 11 Let ST be a specification table for class table CT. *Supertype abstraction is valid for* ST, CT iff for all $\Gamma \vdash S$ and all Γ-specifications *spec*, (6) implies (7), where

$$\forall \dot{\mu} \in XMethEnv \cdot \dot{\mu} \models ST \Rightarrow \mathcal{S}[\![\Gamma \vdash S]\!]\dot{\mu} \models spec \qquad (6)$$

$$\forall \mu \in MethEnv \cdot \mu \models ST \Rightarrow \mathcal{D}[\![\Gamma \vdash S]\!]\mu \models spec \qquad (7)$$

The idea is that modular reasoning establishes (6) but it is then a consequence that S satisfies *spec* in the sense of the standard semantics, i.e., $\mathcal{D}[\![\Gamma \vdash S]\!]\widehat{\mu} \models spec$ where $\widehat{\mu}$ is the semantics of the class table —provided that $\widehat{\mu}$ does satisfy ST (i.e., there is a proof obligation that each method implementation satisfies its specification). In fact, owing to modularity of reasoning about satisfaction as described by (6), the stronger conclusion (7) can be drawn.

In light of Lemma 10, the main result says that behavioral subtyping validates such reasoning, and indeed it is necessary.

Theorem 12 (supertype abstraction) For any satisfiable ST the following are equivalent.

(a) ST has supertype abstraction for method specifications.
(b) supertype abstraction is valid for ST, CT.

We sketch the argument here and return to it after laying some groundwork. The idea for (b) ⇒ (a) is to instantiate S with suitable method call and unfold the semantics. The idea for (a) ⇒ (b) is to prove (b) by structural induction on S assuming that ST has supertype abstraction. A key lemma to prove (b) is an analogous result for expressions, also proved by induction. In these proofs there are three kinds of cases:

- S is a method call —then the semantics is used to reduce implication (6) ⇒ (7) to the implication given by the supertype abstraction property.
- S is some other primitive expression or command —then the semantics is used to prove (7) directly from (6).
- S is a compound command or expression. In that case we decompose it to some state transformers used in its semantic definition and appeal to the induction hypothesis for these state transformers and certain specifications obtained using weakest preconditions.

It is important that the quantifiers are arranged as they are in Def. 11, so that the induction hypothesis is of the form "for all *spec* and S, (6) ⇒ (7)", because for a given S of the third kind we need multiple instantiations of *spec* and S.

Putting the Theorem together with lemma 10 we obtain the following.

Corollary 13 (semantic soundness and completeness) Suppose ST has behavioral subtyping. Suppose $\mathcal{S}[\![\Gamma \vdash S]\!]\dot{\mu}$ can be proved to satisfy some Γ-specification *spec*, assuming only that $\dot{\mu}$ satisfies ST. Then the actual semantics $\mathcal{D}[\![\Gamma \vdash S]\!]\widehat{\mu}$ satisfies *spec*, provided that the semantics, $\widehat{\mu}$, of the class table satisfies ST. Moreover, if such reasoning is sound then ST has behavioral subtyping.

Proving the Theorem. The most difficult part of the proof is to prove (7) by structural induction on S (Lemmas 16 and 17). We need to argue in a way that can be illustrated by considering the case that S is a sequence $S_0; S_1$. Dropping the semantic brackets etc., the idea is that to show $S_0; S_1 \models spec$ we find specifications $spec_0$ and $spec_1$ such that $S_0 \models spec_0$, $S_1 \models spec_1$. If $S_0; S_1 \models spec$ in the static dispatch semantics then owing to our careful choice of $spec_0$ and $spec_1$ we get $S_0 \models spec_0$ and $S_1 \models spec_1$ in the static dispatch semantics. Now the induction hypothesis can be invoked to yield that these hold in the dynamic dispatch semantics, whence the semantic equation for sequence can be used to get $S_0; S_1 \models spec$. Suitable $spec_0$ and $spec_1$ can be obtained by using the weakest precondition of S_1 with respect to the given *post*. So the technical details involve defining the weakest precondition for a state transformer and proving decomposition results for sequential and conditional composition of state transformers.

The semantics for expressions and commands are based on explicitly defined state transformers (described by the mathematical notations for updates, dropping variables, etc.) and those given by the method environment. These are composed sequentially and conditionally. (The definitions use the $\perp$-strict **let** construct which combines sequence with testing for $\perp$.)

Suppose σ is a state transformer of type $\Gamma \rightsquigarrow \Gamma'$ and *post* is a subset of $State(\Gamma) \times State(\Gamma')$. Define the *weakest precondition of* σ *with respect to post*, written $wp\ \sigma\ post$, to be a subset of $State(\Gamma)$ as follows:

$$t \in wp\ \sigma\ post \iff (t, \sigma\ t) \in post$$

Clearly $\sigma \models ((wp\ \sigma\ post), post)$. Now we can give the decomposition lemma for sequenced transformers.

Lemma 14 (sequential decomposition) Suppose σ is a state transformer of type $\Gamma_0 \rightsquigarrow \Gamma_2$. Suppose moreover that $\sigma = \sigma_0; \sigma_1$ where each σ_i has type $\Gamma_i \rightsquigarrow \Gamma_{i+1}$ and semicolon is function composition. Suppose $spec = (pre, post)$ and define $spec_0 = (pre, (wp\ \sigma_1\ post))$ and $spec_1 = ((wp\ \sigma_1\ post), post)$. Then

$$\sigma \models spec \text{ iff } \sigma_0 \models spec_0 \text{ and } \sigma_1 \models spec_1$$

and moreover for any σ_0', σ_1', if $\sigma_0' \models spec_0$ and $\sigma_1' \models spec_1$ then $\sigma_0'; \sigma_1' \models spec$.

These are well known facts about weakest preconditions; the lemma merely spells them out in a particular way because their use later is a little intricate. Similarly for the following.

Lemma 15 (conditional decomposition) Suppose σ is a state transformer of type $\Gamma \rightsquigarrow \Gamma'$ such that for all states s we have

$$\sigma\ s = (\mathsf{if}\ s \in P\ \mathsf{then}\ \sigma_0\ s\ \mathsf{else}\ \sigma_1\ s)$$

where $P \subseteq State(\Gamma)$ and each σ_i has type $\Gamma \rightsquigarrow \Gamma'$. Suppose *spec* has type $\Gamma \rightsquigarrow \Gamma'$ and let $spec = (pre, post)$ to define $spec_0 = (P \cap pre, post)$ and $spec_1 = (pre - P, post)$. Then

$$\sigma \models spec \text{ iff } \sigma_0 \models spec_0 \text{ and } \sigma_1 \models spec_1$$

and moreover $(\mathsf{if}\ s \in P\ \mathsf{then}\ \sigma_0'\ s\ \mathsf{else}\ \sigma_1'\ s) \models spec$ for any σ_0', σ_1' that satisfy $spec_0, spec_1$.

Lemma 16 (supertype abstraction for expressions) Suppose ST has supertype abstraction and is satisfiable. Then for all Γ, E, T, *spec* such that $\Gamma \vdash E : T$ and *spec* is of type $\Gamma \rightsquigarrow [\mathsf{res} : T, \mathsf{exc} : \mathsf{Thrwbl}]$ we have that (8) implies (9) where

$$\forall \dot{\mu} \in XMethEnv \cdot \dot{\mu} \models ST \Rightarrow \mathcal{S}[\![\Gamma \vdash E : T]\!]\dot{\mu} \models spec \qquad (8)$$

$$\forall \mu \in \mathit{MethEnv} \cdot \mu \models ST \;\Rightarrow\; \mathcal{D}[\![\Gamma \vdash E : T]\!]\mu \models \mathit{spec} \qquad (9)$$

Proof By structural induction on E. For the case $\Gamma \vdash x : T$, the result follows from a much stronger property:

$$\mathcal{S}[\![\Gamma \vdash x : T]\!]\dot{\mu} = \mathcal{D}[\![\Gamma \vdash x : T]\!]\mu \quad \text{for all } \mu \text{ and all } \dot{\mu}$$

This holds because the two semantics are identical, there are no subexpressions, and the semantics is independent of the method environment. The argument is the same for **null** and other constants, as well as for $x.f$, $x = y$, E **is** T, $(T)\,x$, and **new** $C()$.

The remaining cases are $x.m(\overline{x})$, which involves the method environment and supertype abstraction, and **let** x **be** E **in** $E1$, for which the induction hypothesis is used.

For case $\Gamma \vdash$ **let** x **be** E **in** $E1 : U$, assume

$$\begin{array}{l}\forall \dot{\mu} \in \mathit{XMethEnv} \cdot \dot{\mu} \models ST \\ \Rightarrow \mathcal{S}[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!]\dot{\mu} \models \mathit{spec}\end{array} \qquad (10)$$

for some *spec* of type $\Gamma \rightsquigarrow [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thrwbl}]$. Let μ be any normal method environment such that $\mu \models ST$. We must show

$$\mathcal{D}[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!]\mu \models \mathit{spec}$$

By assumption (10) and satisfiability of ST there is some $\dot{\mu}$ such that

$$\mathcal{S}[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!]\dot{\mu} \models \mathit{spec} \qquad (11)$$

Let us write out the definition of $\mathcal{S}[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!]\dot{\mu}$ in a way that makes explicit the manipulation of $\bot$. (The definitions in Table 3 use the monadic **let** to suppress $\bot$.)

$$\begin{array}{l}\mathcal{S}[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!]\dot{\mu}(\rho, h, r) = \\ \quad \mathsf{let}\ X = \mathcal{S}[\![\Gamma \vdash E : T]\!]\dot{\mu}(\rho, h, r)\ \mathsf{in} \\ \quad \mathsf{if}\ X = \bot\ \mathsf{then}\ \bot \\ \quad \mathsf{else\ let}\ (\rho_0, h_0, r_0) = X\ \mathsf{in} \\ \quad\quad \mathsf{if}\ r_0\,\mathsf{exc} \neq \mathit{null} \\ \quad\quad \mathsf{then}\ (\rho_0, h_0, [\mathsf{res} : \mathit{default}\ U, \mathsf{exc} : r_0\,\mathsf{exc}]) \\ \quad\quad \mathsf{else\ let}\ r_1 = [r,\ x : r_0\,\mathsf{res}]\ \mathsf{in}\ \mathcal{S}[\![\Gamma \vdash E1 : U]\!]\dot{\mu}(\rho_0, h_0, r_1)\end{array}$$

The dynamic-dispatch semantics is identical except for replacing the three occurrences of $\mathcal{S}[\![-]\!]$ with $\mathcal{D}[\![-]\!]$ and using a normal method environment.

The point of writing out the semantics is to make clear that it is just the alternative/sequential composition of certain state transformers:[11]

- $\mathcal{S}[\![\Gamma \vdash E : T]\!]\dot{\mu}$ of type $\Gamma \rightsquigarrow [\mathsf{res} : T, \mathsf{exc} : \mathsf{Thrwbl}]$
- $\mathcal{S}[\![[\Gamma,\ x : T] \vdash E1 : U]\!]\dot{\mu}$ of type $[\Gamma,\ x : T] \rightsquigarrow [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thrwbl}]$
- others, such as one we will call f, that sends (ρ_0, h_0, r_0) to $(\rho_0, h_0, [\mathsf{res} : \mathit{default}\ U, \mathsf{exc} : r_0\,\mathsf{exc} : \mathsf{Thrwbl}])$.

By the decomposition lemmas there are specifications spec_E, spec_{E1}, spec_f, ... such that (11) holds iff each of the component transformers satisfies its specification.

Since assumption (10) holds for all $\dot{\mu}$, it follows that $\mathcal{S}[\![\Gamma \vdash E : T]\!]\dot{\mu} \models \mathit{spec}_E$ for all $\dot{\mu}$ and $\mathcal{S}[\![\Gamma \vdash E1 : U]\!]\dot{\mu} \models \mathit{spec}_{E1}$ for all $\dot{\mu}$. (Some readers will want to check that the arrangement of quantifiers in the Lemma does justify this step and that plausible simplifications do not work.) As a consequence, we may appeal to the induction hypothesis for Γ, E, T, spec_E and for $[\Gamma,\ x : T]$, $E1$, U, spec_{E1}. This yields that $\mathcal{D}[\![\Gamma \vdash E : T]\!]\mu \models \mathit{spec}_E$ and $\mathcal{D}[\![[\Gamma,\ x : T] \vdash E1 : U]\!]\mu \models \mathit{spec}_{E1}$ for our arbitrarily chosen μ. The other component transformers like f are same in both the static and dynamic dispatch semantics. Having established that the component transformers of $\mathcal{D}[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!]\mu$ all satisfy the component specifications, we obtain $\mathcal{D}[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!]\mu \models \mathit{spec}$ which was to be proved.

Finally, consider the case of $\Gamma \vdash x.m(\overline{y}) : U$. Recall the static dispatch semantics for methods:

$$\begin{array}{l}\mathcal{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!]\dot{\mu}(\rho, h, r) \;= \\ \quad \mathsf{if}\ r\,x = \mathit{null}\ \mathsf{then}\ \mathit{except}(\rho, h, U, \mathsf{NullDeref}) \\ \quad \mathsf{else\ let}\ r_1 = [\mathsf{self} : r\,x, \overline{z} : r\,\overline{y}]\ \mathsf{in}\ \dot{\mu}(T, m)(\rho, h, r_1)\end{array}$$

Suppose $\mathit{mtype}(m, T) = \overline{z} : \overline{T} \rightarrow U$ as in the typing rule for method call. Suppose *spec* has type $\Gamma \rightsquigarrow [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thrwbl}]$ and choose some $\dot{\mu}$ such that $\mathcal{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!]\dot{\mu} \models \mathit{spec}$. (Such $\dot{\mu}$ exists owing to satisfiability of ST and the assumption (10).) By decomposition we obtain spec' of type $\overline{z} : \overline{T} \rightarrow U$ such that $\dot{\mu}(T, m) \models \mathit{spec}'$. Moreover, noting that if $\dot{\mu} \models ST$ then so does $[\dot{\mu} \mid (T, m) : \sigma]$ for any σ with $\sigma \models ST(T, m)$, it follows from assumption (10) that $ST(T, m) \sqsupseteq^T \mathit{spec}'$.

Now suppose μ is any normal method environment that satisfies ST and recall the dynamic dispatch semantics which differs in using the dynamic type $\rho(r\,x)$ of the receiver, rather than its static type T, to look up the method in the environment.

$$\begin{array}{l}\mathcal{D}[\![\Gamma \vdash x.m(\overline{y}) : U]\!]\mu(\rho, h, r) \;= \\ \quad \mathsf{if}\ r\,x = \mathit{null}\ \mathsf{then}\ \mathit{except}(\rho, h, U, \mathsf{NullDeref}) \\ \quad \mathsf{else\ let}\ r_1 = [\mathsf{self} : r\,x, \overline{z} : r\,\overline{y}]\ \mathsf{in}\ \mu(\rho(r\,x), m)(\rho, h, r_1)\end{array}$$

By supertype abstraction for methods (Def. 9), $ST(T, m) \sqsupseteq^T \mathit{spec}'$ implies that $C \leq T \Rightarrow ST(C, m) \sqsupseteq^C \mathit{spec}'$ for all C. Since $\mu \models ST$ we have for each $C \leq T$ that $\mu(C, m) \models ST(C, m)$ and thus $\mu(C, m) \Vdash^C \mathit{spec}'$. To complete the proof of $\mathcal{D}[\![\Gamma \vdash x.m(\overline{y}) : U]\!]\mu$ it is not enough to use decomposition backwards; we also unfold the definition of $\models$ and since $\mu(C, m)$ is used just in case $\rho(r\,x) \leq C$. □

The satisfiability hypothesis is necessary. Suppose that $ST(C, m)$ is unsatisfiable for some C. Then (8) implies (9) because both have false antecedents. This does not let us drop the satisfiability hypothesis because it can happen that the only unsatisfiable part is some interface specification $ST(I, m)$, falsifying the antecedent only of (8).

The following result amounts to the (a) ⇒ (b) part of Theorem 12.

Lemma 17 (supertype abstraction for commands) Suppose ST has supertype abstraction and is satisfiable. Then for all Γ, S, *spec* such that $\Gamma \vdash S$ and *spec* is of type $\Gamma \rightsquigarrow [\Gamma,\ \mathsf{exc} : \mathsf{Thrwbl}]$ we have that

$$\forall \dot{\mu} \in \mathit{XMethEnv} \cdot \dot{\mu} \models ST \;\Rightarrow\; \mathcal{S}[\![\Gamma \vdash S]\!]\dot{\mu} \models \mathit{spec}$$

implies

$$\forall \mu \in \mathit{MethEnv} \cdot \mu \models ST \;\Rightarrow\; \mathcal{D}[\![\Gamma \vdash S]\!]\mu \models \mathit{spec}$$

Proof By structural induction on S.

In the cases that S is $x.f := y$ and **throw** x, the semantics using $\mathcal{S}[\![-]\!]$ and $\mathcal{D}[\![-]\!]$ are identical so the proof is immediate.

In the cases of conditional, sequence, try-catch and try-finally, the argument is by induction following the pattern for let-expression in the proof of Lemma 16. That is also the pattern for the remaining command form, $x := E$, except that instead of the induction hypothesis there is an appeal to Lemma 16 for E. □

It remains to prove the (b) ⇒ (a) part of Theorem 12; for this, (b) can be specialized to the case of method calls. For lack of space the proof is in an appendix.

Lemma 18 If ST is satisfiable then it has supertype abstraction for method specifications provided that (12) implies (13) for all T and all $m \in \mathit{Meths}\ T$ with $\mathit{mtype}(m, T) = \overline{x} : \overline{T} \rightarrow U$, where

$$\begin{array}{l}\forall \dot{\mu} \in \mathit{XMethEnv} \cdot \dot{\mu} \models ST \Rightarrow \\ \mathcal{S}[\![[\mathsf{self} : T, \overline{y} : \overline{T}, z : U] \vdash z := \mathsf{self}.m(\overline{y})]\!]\dot{\mu} \models \mathit{spec}\end{array} \qquad (12)$$

$$\begin{array}{l}\forall \mu \in \mathit{MethEnv} \cdot \mu \models ST \Rightarrow \\ \mathcal{D}[\![[\mathsf{self} : T, \overline{y} : \overline{T}, z : U] \vdash z := \mathsf{self}.m(\overline{y})]\!]\mu \models \mathit{spec}\end{array} \qquad (13)$$

6. Object invariants

This section formalizes a semantic account of invariants in which we can express what is achieved by invariant methodologies (Sect. 2.3) and prove soundness of supertype abstraction with invariants.

An object invariant is a predicate that holds of a particular object together with some others on which it depends. So an invariant suitable for objects of type T can be taken to be a subset of $\mathit{State}([\mathsf{self} : T])$. In this section we extend the notion of specification table as follows.

Definition 19 A specification table, ST, consists of method specifications $ST(T, m)$ exactly as in Definition 3, together with a predicate $ST(T) \subseteq \mathit{State}([\mathsf{self} : T])$, also written inv^T, for each ref type T.

Invariants serve as pre- and post-conditions for methods. For a method specification of the form $ST(T, m) = (\mathit{pre}^T, \mathit{post}^T)$ where

[11] We refrain from spelling out the details, which are a bit intricate owing to the way r is threaded through.

$mtype(m, T) = \overline{x} : \overline{T} \to U$, we will be concerned with specifications of the form

$$(pre_m^T \cap inv^T,\ post_m^T \cap inv_m^T) \tag{14}$$

Taken literally this is nonsense —the intersections are empty— because inv^T is a subset of $State([\mathsf{self} : T])$ whereas pre^T is a subset of $State([\mathsf{self} : T, \overline{x} : \overline{T}])$ and $post^T$ is a subset of $State([\mathsf{self} : T, \overline{x} : \overline{T}]) \times State([\mathsf{res} : U, \mathsf{exc} : \mathsf{Thrwbl}])$. But there is a natural way to make sense of the intersections. For the precondition, we can define $\widehat{inv}^T \subseteq State([\mathsf{self} : T, \overline{x} : \overline{T}])$ by $(\rho, h, r) \in \widehat{inv}^T$ iff $(\rho, h, r - \overline{x}) \in inv^T$. (Here $r - \overline{x}$ means to remove it from the domain of r.) Now the intersection of $\widehat{inv}^T$ with pre^T makes sense. For the postcondition we interpret $post^T \cap \widetilde{inv}^T$ to mean the set of $(s, (\rho, h, r))$ such that $(s, (\rho, h, r))$ is in $post^T$ and $(\rho, h, r - \overline{x})$ is in inv^T.

What is achieved by invariant methodologies is to arrange that in fact inv^T holds before any method call, so that the reasoning is sound. In this section our formalization adapts Def. 4 as follows.

Definition 20 (satisfaction by method environment, $\models^{\mathsf{inv}}$) Let ST be a specification table. An extended method environment $\dot{\mu}$ satisfies ST, written $\dot{\mu} \models^{\mathsf{inv}} ST$, iff for all ref types T and $m \in Meths\ T$ we have

$$\dot{\mu}(T, m) \models (pre^T \cap \widehat{inv}^T,\ post^T \cap \widetilde{inv}^T)$$

where $(pre^T, post^T) = ST(T, m)$ and $inv^T = ST(T)$.

A normal method environment μ satisfies ST, written $\mu \models^{\mathsf{inv}} ST$ iff for all classes C and $m \in Meths\ C$ we have

$$\mu(C, m) \models (pre^C \cap \widehat{inv}^C,\ post^C \cap \widetilde{inv}^C)$$

Our formalization in the preceding section treated a specification $ST(T, m)$ both as the proof obligation for an implementation and as an assumption to be made for an invocation. Now we are considering the situation where the proof obligation is (14).

What is achieved by a sound methodology has a simple semantic formulation: reasoning about a command S is carried out using the static-dispatch semantics as before but now the proof obligation for method implementations is in terms of $\mu \models^{\mathsf{inv}} ST$ from Def. 20. That is, we adapt Def. 11 as follows.

Definition 21 Let ST be a specification table for class table CT. *supertype abstraction is valid for* ST, CT iff for all $\Gamma \vdash S$ and all Γ-specifications *spec*, (15) implies (16), where

$$\forall \dot{\mu} \in XMethEnv \cdot \dot{\mu} \models^{\mathsf{inv}} ST \Rightarrow \mathcal{S}[\![\Gamma \vdash S]\!]\dot{\mu} \models spec \tag{15}$$

$$\forall \mu \in MethEnv \cdot \mu \models^{\mathsf{inv}} ST \Rightarrow \mathcal{D}[\![\Gamma \vdash S]\!]\mu \models spec \tag{16}$$

The point is that (15) will be false unless inv^T holds as a precondition of every invocation on an object of dynamic type T. Of course it is not an explicit precondition, since inv^T is a conjunction of several invariants which may be implicit in the methodology and/or not visible at the call site. The soundness result for a methodology must nonetheless ensure that the effective invariant, which we refer to as inv^T, does hold before every call.

The authors were surprised to find how little impact the addition of invariants has on the theory of supertype abstraction and behavioral subtyping.[12] Definition 9 of supertype abstraction does not need to change. Recall that ST has supertype abstraction for method specifications just if for all ref types T, all $m \in Meths\ T$, and all method specifications *spec* of type $mtype(T, m)$: if $ST(T, m) \sqsupseteq^T spec$ then $ST(C, m) \sqsupseteq^C spec$ for every class C with $C \leq T$.

Definition 22 (behavioral subtyping, with invariants) A specification table ST has *behavioral subtyping* if and only if for all ref types U and classes C with $C \leq U$ and all $m \in Meths\ U$ we have $ST(C, m) \sqsupseteq^C ST(U, m)$ and moreover $ST(C) \subseteq ST(U)$.

The revised version of Lemma 10 is no longer an equivalence.

[12] We suspect that the history constraints may similarly have little impact.

Lemma 23 (behavioral subtyping, with invariants) Any satisfiable specification table with behavioral subtyping has supertype abstraction for method specifications.

This is an immediate consequence of Lemma 10: the relevant definitions have not changed except for the added invariant condition in Def. 22. Unlike in Lemma 10, the converse does not hold. Supertype abstraction as we have defined it is only concerned with reasoning about method calls. Behavioral subtyping also imposes a condition on invariants. It would be straightforward to add "invariant reasoning" to the notion of supertype abstraction and thereby get an equivalence like Lemma 10, but it would shed no light. Thus the analog of Corollary 13, which we refrain from stating, is only an implication.

Theorem 24 (supertype abstraction with invariants) For any satisfiable ST the following are equivalent.

(a) ST has supertype abstraction for method specifications.
(b) Supertype abstraction is valid for ST, CT.

Remarkably, the proof looks just the same as the proof of Theorem 12. The reason is that the proof involves conditions dependent on $\mu \models ST$ and $\dot{\mu} \models ST$ and various arguments about refinement and specifications. But it does not involve the definition of $\mu \models ST$ or $\dot{\mu} \models ST$ per se. The argument goes through using $\models^{\mathsf{inv}}$ instead of $\models$.

What has changed is the interpretation of the result. Hypothesis (15) describes modular reasoning that S satisfies *spec* under the assumption that methods satisfy their specifications, in the revised sense of satisfaction that adjoins the invariant. In practice, this means the reasoning system has restricted the invariants and restricted S in such a way that an object's invariant indeed holds as a precondition for every invocation. The conclusion (16) says that indeed S is correct in the context of the real program semantics, provided that the implementations do satisfy their specifications with invariant adjoined.

Although supertype abstraction is based on $\models^{\mathsf{inv}}$, behavioral subtyping (Def 22) is still defined using the relation $\sqsupseteq$ applied to the pre/post specifications in the specification table.

7. Checking and ensuring behavioral subtyping

In light of the importance of behavioral subtyping, it is obviously useful to check whether given specifications satisfy the refinement condition. A characterization of $\sqsupseteq$ is needed since the definition of quantifies over all state transformers. This is the first topic of this section. The second topic is an alternative to checking: to impose behavioral subtyping by fiat.

Characterizing refinement. The most common formulation of behavioral subtyping uses the implications (2) and (3) that correspond to the rule of consequence in Hoare logic which derives a weaker specification from a stronger one. Even in Hoare logic for simple procedures these are incomplete. Hoare proposed an "adaptation rule" which is not complete and some subsequent proposals were found to have subtle unsoundness in connection with auxiliary variables in specifications (corrected in [4]). By now, sound and complete rules are known and the connection with specification refinement has been made clear [13, 33, 41, 43].

To characterize when $(pre', post') \sqsupseteq (pre, post)$ holds, it is not difficult to show necessity of $pre \subseteq pre'$; it is the postconditions that are tricky. In our setting, one characterization is $old(pre) \cap post' \subseteq post$, cf. (4). There is an equivalent condition [14] that is similar to the join of specifications investigated later.

$$pre \subseteq pre' \ \wedge\ (\neg old(pre') \cup post') \subseteq (\neg old(pre) \cup post)$$

where we write $\neg p$ for the complement of p with respect to the set of all states of its type. Taking the type of self into account we have the following (the proof can be adapted from arguments in [13]).

Lemma 25 (characterization of refinement) Let $(pre, post)$ be a specification of type $\Gamma \rightsquigarrow \Gamma'$ and $(pre', post')$ be of type $[\Gamma \mid \mathsf{self} : T] \rightsquigarrow \Gamma'$ where $T \leq \Gamma\,\mathsf{self}$. Then $(pre', post') \sqsupseteq^T (pre, post)$ if and only if

$$pre \cap p \subseteq pre' \ \wedge\ old(pre) \cap post' \subseteq post$$

where $p = State([\Gamma \mid \mathsf{self} : T])$.

Completeness of such a condition depends on the program semantics, since the definition of $\sqsupseteq^T$ quantifies over all program meanings. Moreover the presence of general auxiliary variables scoped over specifications can require a more complicated condition [33, 41, 43]. For the results in this paper, we found that using general auxiliary variables led to quite complicated semantic definitions so for perspicuity we chose the simpler form of specification with two-state postconditions. By disentangling the characterization of refinement from the use of refinement to define and reason about behavioral subtyping, we make it straightforward to adapt our results to other forms of specification.

Specification inheritance. Specification inheritance was pioneered by Wills [50, 51] and is found in JML where it is made explicit by the **also** keyword. It is defined in terms of the join of specifications. Regardless of whether the class table is finite, there are only finitely many U that are supertypes of a given T so binary join suffices. The key property of specification inheritance is that it forces behavioral subtyping [14], as we show below.

Definition 26 (join of specifications) Let $(pre, post)$ and $(pre', post')$ be specifications of type $\Gamma \rightsquigarrow \Gamma'$. Define $(pre, post) \sqcup (pre', post')$ to be (p, q) of the same type, where

$$p = pre \cup pre'$$
$$q = (\neg old(pre) \cup post) \cap (\neg old(pre') \cup post')$$

Lemma 27 (least upper bound) Consider specifications of some type $\Gamma \rightsquigarrow \Gamma'$. The specification $(pre, post) \sqcup (pre', post')$ is a least upper bound of $(pre, post)$ and $(pre', post')$ among specifications of type $\Gamma \rightsquigarrow \Gamma'$.

The proof is in Appendix C.

Join only involves specifications at a single type, but for specification inheritance we need to consider its effect in terms of different types. To that end we need the following result which has a tedious but elementary proof in which the definitions are unfolded all the way to satisfaction.

Lemma 28 (monotonicity) Suppose $spec_0$ and $spec_1$ are specifications of some type $\Gamma \rightsquigarrow \Gamma'$ and suppose $T \leq \Gamma\,\mathsf{self}$. Suppose $spec_0'$ and $spec_1'$ are specifications of type $[\Gamma \mid \mathsf{self} : T]$ and moreover $spec_i' \sqsupseteq^T spec_i$ for $i = 0$ and $i = 1$. Then $(spec_0' \sqcup spec_1') \sqsupseteq^T (spec_0 \sqcup spec_1)$.

Definition 29 (inheriting specifications) Let ST be a specification table as in Def. 19. Define specification table $\widehat{ST}$ as follows.

For each T let $\widehat{ST}(T)$ be the intersection of $ST(U)$ over all U with $T \leq U$.

For each T and m in $Meths\ T$, let $\widehat{ST}(T, m)$ be the join of over all U with $T \leq U$ of $ST(U, m)\restriction T$.

Recall from Def. 5 that $(pre, post)\restriction T$ denotes the restriction of the specification to states where self is $\leq T$.

Theorem 30 If $\widehat{ST}$ is the extension of ST by specification inheritance then $\widehat{ST}$ has behavioral subtyping.

Proof If $T \leq V$ then $\widehat{ST}(T) \subseteq \widehat{ST}(V)$ because if $V \leq U$ then $T \leq U$ so $\widehat{ST}(T)$ is an intersection of more sets.

For methods, suppose $T \leq V$. Let X be the set of U such that $U \geq V$ and observe that X is a subset, possibly a proper subset, of the set of U with $U \geq T$. Now $\widehat{ST}(V, m)$ is the join of all $ST(U, m)\restriction V$ with $U \in X$. Let JT be the join of $ST(U, m)\restriction T$ with $U \in X$. By the least upper bound property (Lemma 27), $\widehat{ST}(T, m) \sqsupseteq JT$ since $\widehat{ST}(T, m)$ joins over a set containing X. By definitions we have for every $U \geq V$ that $ST(U, m)\restriction T \sqsupseteq^T ST(U, m)\restriction V$, so by Lemma 28 we have that $JT \sqsupseteq^T \widehat{ST}(V, m)$. Provided that JT is satisfiable, this can be put together with $\widehat{ST}(T, m) \sqsupseteq JT$ to get $\widehat{ST}(T, m) \sqsupseteq^T \widehat{ST}(V, m)$ (by Lemma 7). If JT is unsatisfiable, then since $\widehat{ST}(T, m)$ is unsatisfiable and $\widehat{ST}(T, m) \sqsupseteq^T \widehat{ST}(V, m)$ is immediate by definition of $\sqsupseteq^T$. □

8. Conclusions

We have formalized behavioral subtyping in terms of the intrinsic refinement ordering on specifications, where $spec \sqsupseteq spec'$ means that any program satisfying $spec$ also satisfies $spec'$. For T to be a behavioral subtype of U involves conditions like $spec_m^T \sqsupseteq spec_m^U$, where $spec_m^U$ is the specification for method m in class U. We have also formalized supertype abstraction: static reasoning about a program fragment using only specifications of methods it calls. The main result, Corollary 13, says that behavioral subtyping not only validates supertype abstraction but is equivalent to it, which we view as semantic completeness (see Thms. 12 and 24).

The semantic model has been encoded in the PVS theorem prover and type soundness proved, building on previous work [35]. Machine checking of the results of this paper is planned in the near future.

It is impractical to check $\sqsupseteq$ directly since its definition quantifies over all implementations. The contravariant/covariant relations (2) and (3) are a sound approximation but not complete. A corollary of Lemma 25 is that behavioral subtyping is equivalent to a simple logical condition relating pre- and post-conditions in the relevant specifications. However, to apply the condition to check, say $spec_m^T \sqsupseteq spec_m^U$, requires reasoning about the entire specification whereas in practice the specification should be defined in terms of state in supertypes that is often not visible in the subtype, and vice versa. This is especially true for postconditions that express modifies specifications. This is one motivation for constructing the "effective specifications" by specification inheritance from arbitrary invariants and method specifications explicitly declared by the programmer. Theorem 30 confirms that the result has behavioral subtyping.

On the other hand, joining specifications in this way can lead to unimplementable specifications. It can be argued that, at least in situations where both super- and sub-type specifications are visible, failure of refinement should be detected as a design error [15] rather than masked by specification inheritance. Further investigation would be worthwhile to reconcile the issues pertaining to information hiding and reuse with the need to detect design flaws, leading to better tools and specification notations.

Another direction for future work is to extend our treatment of behavioral subtyping to other kinds of specifications and programming languages, especially concurrency and temporal logic. Such an extension would involve answering questions such as: what is the right notion of specification inheritance for temporal logic? Another direction would be to extend our treatment to explicitly deal with frame axioms (modifies) clauses and other features of rich specifications (such as universally quantified auxiliaries).

An active area of research is to find easily used disciplines for making local definitions of relevant pieces of specifications —especially invariants— which can given an interpretation that validates modular reasoning. Our results serve to specify the semantic properties to be achieved by any sound discipline. Tools may be unsound or incomplete by design. Our results will help ensure that it is not for lack of theory or understanding.

Acknowledgments

Thanks to Paulo Borba and Augusto Sampaio for hosting us in Recife during our initial work on this paper and for stimulating discussions. Thanks to Shengchao Qin, Joseph Kiniry, Andreas Podelski, and Erik Poll for comments on earlier drafts.

References

[1] S. Alagic and S. Kouznetsova. Behavioral compatibility of self-typed theories. In B. Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *LNCS*, pages 585–608, Berlin, June 2002. Springer-Verlag.

[2] P. America. Inheritance and subtyping in a parallel object-oriented language. In J. Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, June 1987. Springer-Verlag. Lecture Notes in Computer Science, Volume 276.

[3] 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, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *LNCS*, pages 60–90. Springer-Verlag, New York, 1991.

[4] P. America and F. de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84(2):129–164, 1990.

[5] A. Banerjee and D. A. Naumann. Representation independence, confinement and access control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 166–177, 2002.

[6] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, Nov. 2005. Extended version of [5].

[7] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.

[8] 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, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop (CASSIS 2004), Revised Selected Papers*, volume 3362 of *LNCS*, pages 49–69, 2005.

[9] B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines and higher-order separation logic. In *European Symposium on Programming (ESOP)*, volume 3444 of *LNCS*, pages 233–247, 2005.

[10] G. Bierman and M. Parkinson. Separation logic and abstraction. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 247–258, 2005.

[11] K. B. Bruce and P. Wegner. An algebraic model of subtypes in object-oriented languages (draft). *ACM SIGPLAN Notices*, 21(10), Oct. 1986.

[12] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988.

[13] Y. Chen and B. H. C. Cheng. A semantic foundation for specification matching. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 91–109. Cambridge University Press, New York, NY, 2000.

[14] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, Mar. 1996.

[15] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *OOPSLA '01 Conference Proceedings, Object-Oriented Programming, Systems, Languages, and Applications, October 14-18, 2001, Tampa Bay, Florida, USA*, pages 1–15, Oct. 2001.

[16] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conf. on Program. Lang. Design and Implementation (PLDI)*, pages 234–245, 2002.

[17] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[18] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. Syst.*, 23(3):396–459, May 2001.

[19] G. T. Leavens. Verifying object-oriented programs that use subtypes. Technical Report 439, Massachusetts Institute of Technology, Laboratory for Computer Science, Feb. 1989. The author's Ph.D. thesis.

[20] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev29, Iowa State University, Department of Computer Science, Jan. 2006. To appear in *ACM SIGSOFT Software Engineering Notes*.

[21] G. T. Leavens and K. K. Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[22] G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, Nov. 1995.

[23] K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[24] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming*, pages 491–516, 2004.

[25] K. R. M. Leino and P. Müller. A verification methodology for model fields. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[26] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Technical Report see SRC160, 2000.

[27] B. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988. Revised version of the keynote address given at OOPSLA '87.

[28] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[29] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6), 1994.

[30] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, second edition, 1997.

[31] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[32] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. Technical Report 424, ETH Zurich, Mar. 2005.

[33] D. A. Naumann. Calculating sharp adaptation rules. *Inf. Process. Lett.*, 77:201–208, 2001.

[34] D. A. Naumann. Assertion-based encapsulation, object invariants and simulations. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Post-proceedings, Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *LNCS*, pages 251–273, 2005.

[35] D. A. Naumann. Verifying a secure information flow analyzer. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics TPHOLS*, volume 3603 of *LNCS*, pages 211–226, 2005.

[36] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *IEEE Symp. on Logic in Computer Science (LICS)*, pages 313–323, 2004.

[37] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theoretical Comput. Sci.*, 2006. Extended version of [36], to appear.

[38] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *LNCS*, pages 158–185. Springer-Verlag, July 1998.

[39] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 268–280, 2004.

[40] D. v. Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P. A. Lindsay, editors, *Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *LNCS*, pages 89–105. Springer, 2002. `http://isabelle.in.tum.de/Bali/papers/NanoJava.html`.

[41] E.-R. Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Comput. Sci.*, 24:337–347, 1983.

[42] M. J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, Nov. 2005. Dissertation.

[43] C. Pierik. Validation techniques for object-oriented proof outlines. Dissertation, Universiteit Utrecht, 2006.

[44] C. Pierik and F. de Boer. On behavioral subtyping and completeness. In *ECOOP Workshop on Formal Techniques for Java-like Programs*. 2005. To appear.

[45] C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Comput. Sci.*, 2005. to appear.

[46] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

[47] E. Poll. A coalgebraic semantics of subtyping. In H. Reichel, editor, *Coalgebraic Methods in Computer Science (CMCS)*, number 33 in Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam, 2000.

[48] B. Reus. Modular semantics and logics of classes. In M. Baaz and J. A. Makowsky, editors, *Computer Science Logic (CSL)*, volume 2803 of *LNCS*, pages 456–469, 2003.

[49] A. Sabry and M. Felleisen. Reasoning about programs in continuation passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

[50] A. Wills. Specification in Fresco. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, chapter 11, pages 127–135. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[51] J. M. Wing. A two-tiered approach to specifying programs. Technical Report TR-299, MIT Lab for Computer Science, 1983.

A. Syntax

A class table CT is *well formed* provided it satisfies the following.

1. The subtype ordering $\leq$ is acyclic and $T \leq \mathsf{Object}$ for all T.
2. Any ref type that appears as a field type, superclass, local variable type, cast etc. is declared in CT.
3. Field names are not shadowed, that is, if $vis\ f : T$ is in *fields* T and *super* $T = U$ then f is not in *dfields* U. (Note that the effect of the definition of *fields* is that fields are inherited.)
4. Method types —including parameter names— are invariant, that is, if *mtype*(U, m) is defined and $T \leq U$ then *mtype*$(T, m) =$ *mtype*(U, m).
5. For any C, every method declaration $m(\overline{x} : \overline{T}) : T\ \{S\}$ in $CT(C)$ is typable in the sense that $\Gamma \vdash S$ where $\Gamma = [\mathsf{self} : C, \mathsf{res} : T, \overline{x} : \overline{T}]$. Rules that define $\Gamma \vdash S$ appear just below.
6. For any C, any $I \in$ *superinterfaces* C, and any method signature $m(\overline{x} : \overline{T}) : T$ declared or inherited in I, there is a declared or inherited method in C with the same signature.
7. Variable exc does not occur anywhere. (It must be available for use in the semantics, and postconditions can refer to it.)

Typing rules for expressions

$$\Gamma \vdash 0 : \mathbf{int} \qquad \Gamma \vdash x : \Gamma x \qquad \Gamma \vdash \mathbf{new}\ C()$$

$$\frac{T \in \mathit{RefType}}{\Gamma \vdash \mathbf{null} : T} \qquad \frac{\Gamma \vdash x : T_1 \qquad \Gamma \vdash y : T_2}{\Gamma \vdash x = y : \mathbf{bool}}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : U \qquad (\mathit{vis}\ f : T) \in \mathit{dfields}\ V \\ U \leq V \qquad (\mathit{vis} = \mathbf{private} \Rightarrow \Gamma\,\mathsf{self} = V) \\ (\mathit{vis} = \mathbf{protected} \Rightarrow \Gamma\,\mathsf{self} \leq V)\end{array}}{\Gamma \vdash x.f : T}$$

$$\frac{\Gamma \vdash x : T \qquad U \leq T \qquad T \in \mathit{RefType}}{\Gamma \vdash (U)\ x : U}$$

$$\frac{\Gamma \vdash x : T \qquad U \leq T \qquad T \in \mathit{RefType}}{\Gamma \vdash x\ \mathbf{is}\ U : \mathbf{bool}}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : T \qquad T \in \mathit{RefType} \\ \mathit{mtype}(T, m) = \overline{z} : \overline{T} \to U \qquad \Gamma \vdash \overline{y} : \overline{V} \qquad \overline{V} \leq \overline{U}\end{array}}{\Gamma \vdash x.m(\overline{y}) : U}$$

$$\frac{\Gamma \vdash E : T \qquad [\Gamma,\ x : T] \vdash E1 : U}{\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ E\ \mathbf{in}\ E1 : U}$$

Typing rules for commands

$$\frac{\Gamma \vdash E : T \qquad T \leq \Gamma x \qquad x \neq \mathsf{self}}{\Gamma \vdash x := E}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : U \qquad (\mathit{vis}\ f : T) \in \mathit{dfields}\ V \qquad U \leq V \\ \Gamma \vdash y : T1 \qquad T1 \leq T \qquad (\mathit{vis} = \mathbf{private} \Rightarrow \Gamma\,\mathsf{self} = V) \\ (\mathit{vis} = \mathbf{protected} \Rightarrow \Gamma\,\mathsf{self} \leq V)\end{array}}{\Gamma \vdash x.f := y}$$

$$\frac{\Gamma \vdash x : \mathbf{bool} \qquad \Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2} \qquad \frac{[\Gamma,\ x : T] \vdash S}{\Gamma \vdash \mathbf{var}\ x : T\ \mathbf{in}\ S}$$

$$\frac{\Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash S_1;\ S_2} \qquad \frac{\Gamma \vdash x : T \qquad T \leq \mathsf{Thrwbl}}{\Gamma \vdash \mathbf{throw}\ x}$$

$$\frac{\Gamma \vdash S_1 \qquad [\Gamma,\ x : T] \vdash S_2 \qquad T \leq \mathsf{Thrwbl}}{\Gamma \vdash \mathbf{try}\ S_1\ \mathbf{catch}(x : T)\ S_2}$$

$$\frac{\Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash \mathbf{try}\ S_1\ \mathbf{finally}\ S_2}$$

B. Semantics of commands and expressions

The semantics of commands and expressions are defined in figures 3 and 4. The semantics of expressions is defined by recursion on the structure of E. In case E has subexpressions, the definition uses nomenclature from the corresponding typing rule.

To streamline the semantics of expressions, we define a helping function to create exceptional result states. Given ref context ρ, heap $h \in \mathit{Heap}(\rho)$, classname $C \leq \mathsf{Thrwbl}$, and any type T we define $\mathit{except}(\rho, h, T, C)$ to be an element of $\mathit{State}([\mathsf{res} : T, \mathsf{exc} : \mathsf{Thrwbl}])$ as follows.

$$\begin{array}{rcl}\mathit{except}(\rho, h, T, C) & = & \mathsf{let}\ o = \mathit{fresh}(\rho, h)\ \mathsf{in} \\ & & \mathsf{let}\ \rho_o = [\rho,\ o : C]\ \mathsf{in} \\ & & \mathsf{let}\ h_o = [h,\ o : \mathit{default Obrecord}\ C]\ \mathsf{in} \\ & & (\rho_0, h_0, [\mathsf{res} : \mathit{default}\ T, \mathsf{exc} : o])\end{array}$$

This is similar to the semantics of **new** $C()$, but the new object is assigned to exc rather than to res.

A similar helping function is used in the semantics of commands. Given (ρ, h, r) in $\mathit{State}(\Gamma)$ and classname $C \leq \mathsf{Thrwbl}$ we define $\mathit{except}(\rho, h, r, C)$ to be an element of $\mathit{State}([\Gamma,\ \mathsf{exc} : \mathsf{Thrwbl}])$ as follows.

$$\begin{array}{rcl}\mathit{except}(\rho, h, r, C) & = & \mathsf{let}\ o = \mathit{fresh}(\rho, h)\ \mathsf{in} \\ & & \mathsf{let}\ \rho_o = [\rho,\ o : C]\ \mathsf{in} \\ & & \mathsf{let}\ h_o = [h,\ o : \mathit{default Obrecord}\ C]\ \mathsf{in} \\ & & (\rho_0, h_0, [r,\ \mathsf{exc} : o])\end{array}$$

For any derivable typing $\Gamma \vdash E : T$, any method environment μ, and any state $(\rho, h, r) \in \mathit{State}(\Gamma)$, the definition of $[\![\Gamma \vdash E : T]\!]\mu(\rho, h, r)$ yields either $\perp$ or an element of $\mathit{State}([\mathsf{res} : T, \mathsf{exc} : \mathsf{Thrwbl}])$. That is, $[\![\Gamma \vdash E : T]\!]$ is an element of $\mathit{SemExpr}(\Gamma, T)$.

For cast and type test, the typing rule ensures that type T (and hence type U) is a class or interface; this justifies application of ρ in the semantics.

For method call, the receiver object is $r\,x$ so $\rho(r\,x)$ is the dynamic type of the object; thus to look up in method environment μ the meaning of the dynamically dispatched method we write $\mu(\rho(r\,x))m$. Since the argument expressions $\overline{y}$ are variables we can write $r\,\overline{y}$ for their values.

The semantics of commands is also defined by recursion on structure and using nomenclature from the typing rules. For any derivable typing $\Gamma \vdash S$, the definition of $[\![\Gamma \vdash S]\!]$ yields an element of $\mathit{SemCommand}(\Gamma)$. For semantics of local variables we use another bit of notation: To remove an element from the domain of a function we use the minus sign, e.g., if r is a store then $r - \mathsf{exc}$ is the same store but with exc removed from its domain.

Semantics of class table. The semantics of a complete program is a method environment defined as a limit. The first step is to give a meaning for a method declaration $mdec$ of the form **meth** $m(\overline{x}:\overline{T}):T\ \{\ S\ \}$ in some class C. We define $[\![mdec]\!]$ to be a function in

$$MethEnv \to STrans([\mathsf{self}:C, \overline{x}:\overline{T}], [\mathsf{res}:T, \mathsf{exc}:\mathsf{Thrwbl}])$$

as follows, using $\Gamma = [\mathsf{self}:C, \mathsf{res}:T, \overline{x}:\overline{T}]$ so that $\Gamma \vdash S$ (owing to condition 5 in the definition of well formed class table). For any method environment μ and state (ρ, h, r) in $State([\mathsf{self}:C, \overline{x}:\overline{T}])$, define

$$\begin{array}{rl} [\![mdec]\!]\mu(\rho,h,r) = & \mathsf{let}\ r_0 = [r,\ \mathsf{res}:\mathit{default}\ T]\ \mathsf{in} \\ & \mathsf{let}\ (\rho_1,h_1,r_1) = [\![\Gamma \vdash S]\!]\mu(\rho,h,r_0)\ \mathsf{in} \\ & (\rho_1,h_1,r_1 - (\mathsf{self},\overline{x})) \end{array}$$

where *default* is a fixed default value. The next step is to define an ascending chain $\mu \in \mathbb{N} \to MethEnv$ of method environments as follows.

$\mu_0(C,m) = \lambda s \cdot \bot$, for any m declared or inherited in C.
$\mu_{j+1}(C,m) = [\![mdec]\!]\mu_j$, if m is declared as $mdec$ in C.
$\mu_{j+1}(C,m) = restr((\mu_{j+1}(B,m)), C)$, if m inherited in C from B.

Here $restr$ restricts the function $\mu_{j+1}(B,m)$, which is defined on stores with $\mathsf{self}:B$, to stores with $\mathsf{self}:C$. This works because $C \le B$ implies $[\![C]\!] \subseteq [\![B]\!]$ which in turn induces an inclusion for stores.

We take pains to make such conversions explicit throughout the paper. It is necessary for machine-checking the results. More importantly a key aspect of behavioral subtyping is the need for a method declared in some class C to satisfy a specification in which self has some different type $T \ge C$.

Method environments are ordered by $\mu \le \mu'$ iff $\mu(C,m) \le \mu'(C,m)$ for all C, m. This refers to the usual ordering on state transformers: For σ and τ in $STrans(\Gamma,\Gamma')$, define $\sigma \le \tau$ iff for all s in $State(\Gamma)$ we have either $\sigma\, s = \tau\, s$ or $\sigma\, s = \bot$. The everywhere-$\bot$ function is the least element in the set of state transformers of a given type, and this induces a least method environment. These sets are closed under limits of ascending chains. For lack of space we refrain from proving that for any $\Gamma \vdash S$, the semantics $[\![\Gamma \vdash S]\!]$ is a monotonic function from method environments to state transformers. Similarly, the semantics of a method declaration is monotonic in the method environment. It follows that $i \le j \Rightarrow \mu_i \le \mu_j$ for the approximation chain. The semantics $\widehat{\mu}$ is defined to be the least upper bound of the approximation chain.[13]

Remark on closed world. The semantics is given for a class table which is a closed collection of class declarations. Although it is compositional at the level of commands, the semantics is not compositional at the level of classes: the semantics of a class table is not defined by composing a separate semantics for each class. Such a semantics, for a language with mutually recursive methods and mutually recursive class declarations, would be far more complex [48] and less operationally transparent. But one may wonder whether our semantics is suitable for formulating results concerning modular reasoning. In fact we lose little or nothing, since our results quantify over arbitrary class tables. Indeed, there may be infinitely many classes and infinitely many methods in a class. So in some sense there is a universal class table containing all programs.

C. Additional proofs

Proof of Lemma 18. **Proof** For any T, m and any *spec* of type $mtype(m,T)$ we need to show that $ST(T,m) \sqsupseteq^T spec$ implies $ST(C,m) \sqsupseteq^C spec$ for all $C \le T$. This follows by weak transitivity from $ST(C,m) \sqsupseteq^C ST(T,m)$ which we will prove.

The command $z := \mathsf{self}.m(\overline{y})$ is chosen because we can unfold the semantics of $z := \ldots$ as in the proof of Lemma 16, so that this command satisfies *spec* just if $\mathsf{self}.m(\overline{y})$ satisfies an associated expression specification of type $mtype(m,T)$. We refrain from giving the transformation on specifications and simply observe that for any σ that satisfies $ST(T,m)$ there is $\dot{\mu}$ with $\dot{\mu}(T,m) = \sigma$ and $\dot{\mu} \models ST$. Moreover it is only $\dot{\mu}(T,m)$ that has any bearing on whether $\mathcal{S}[\![[\mathsf{self}:T, \overline{y}:\overline{T}, z:U] \vdash z := \mathsf{self}.m(\overline{y})]\!]\dot{\mu}$ satisfies *spec*. So if we instantiate the antecedent (12) by $spec := ST(T,m)$ it amounts to $\forall \sigma \cdot \sigma \models ST(T,m) \Rightarrow \sigma \models ST(T,m)$ which is true. Thus we obtain the consequent (13) with $spec := ST(T,m)$, which boils down, for any $C \le T$, to $\forall \sigma \in SemMeth(C,m) \cdot \sigma \models ST(C,m) \Rightarrow \sigma \models^C ST(T,m)$ whence $ST(C,m) \sqsupseteq^C ST(T,m)$. □

Proof of Lemma 27. **Proof** To show that it is an upper bound, instantiate Lemma 25 with $pre' := pre \cup pre'$ and $post' := (\neg old(pre) \cup post) \cap (\neg old(pre') \cup post')$. Thus the upper bound property $(\neg old(pre) \cup post) \cap (\neg old(pre') \cup post') \sqsupseteq (pre, post)$ follows from $pre \subseteq pre \cup pre'$ —which is immediate— and

$$old(pre) \cap (\neg old(pre) \cup post) \cap (\neg old(pre') \cup post') \subseteq post$$

which also reduces to true.

To show that $\sqcup$ gives the least upper bound, suppose $(pre, post) \sqsupseteq (pre_0, post_0)$ and $(pre, post) \sqsupseteq (pre_1, post_1)$. Now $(pre, post) \sqsupseteq (pre_0, post_0) \sqcup (pre_1, post_1)$ iff $\sigma \models (pre, post)$ implies $\sigma \models (pre_0 \cup post_0,\ ((\neg old(pre_0) \cup post_0) \cap (\neg old(pre_1) \cup post_1))$ for all σ. To prove the consequent for arbitrary σ, consider any state t and suppose $t \in pre_0 \cup pre_1$. We must show that $(t, \sigma\, t)$ is in $(\neg old(pre) \cup post)$ (and a symmetric condition). In the case $t \in pre_0$ we have $(t, \sigma\, t) \in (\neg old(pre_0) \cup post_0)$ immediately. For the case $t \in pre_1$, note that from $(pre, post) \sqsupseteq (pre_1, post_1)$ we have $pre \supseteq pre_1$. So we can use the antecedent $\sigma \models (pre, post)$ to yield $(t, \sigma\, t) \in post$, which concludes the proof that $(t, \sigma\, t)$ is in $(\neg old(pre) \cup post)$. □

[13] A similar semantics is used in [6]; a characterization of least upper bounds and machine-checked proofs of the monotonicity properties etc. appears in [35].

$[\![\Gamma \vdash x : T]\!]\mu(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : r\,x, \mathsf{exc} : \mathit{null}]$

$[\![\Gamma \vdash \mathbf{true} : \mathbf{bool}]\!]\mu(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : \mathit{true}, \mathsf{exc} : \mathit{null}])$

$[\![\Gamma \vdash 0 : \mathbf{int}]\!]\mu(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : 0, \mathsf{exc} : \mathit{null}])$

$[\![\Gamma \vdash \mathbf{null} : T]\!]\mu(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : \mathit{null}, \mathsf{exc} : \mathit{null}])$

$[\![\Gamma \vdash x = y : \mathbf{bool}]\!]\mu(\rho, h, r) \;=\; \mathsf{let}\ v = (\mathsf{if}\ (r\,x = r\,y)\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ \mathit{false})\ \mathsf{in}\ (\rho, h, [\mathsf{res} : v, \mathsf{exc} : \mathit{null}])$

$[\![\Gamma \vdash \mathbf{new}\ C() : C]\!]\mu(\rho, h, r) \;=$

$\quad \mathsf{let}\ o = \mathit{fresh}(\rho, h)\ \mathsf{in}\ \mathsf{let}\ \rho_o = [\rho,\ o : C]\ \mathsf{in}\ \mathsf{let}\ h_o = [h,\ o : \mathit{defaultObrecord}\ C]\ \mathsf{in}\ (\rho_0, h_0, [\mathsf{res} : o, \mathsf{exc} : \mathit{null}])$

$[\![\Gamma \vdash x.f : T]\!]\mu(\rho, h, r) \;=\; \mathsf{if}\ r\,x \neq \mathit{null}\ \mathsf{then}\ (\rho, h, [\mathsf{res} : h(r\,x).f, \mathsf{exc} : \mathit{null}])\ \mathsf{else}\ \mathit{except}(\rho, h, T, \mathsf{NullDeref})$

$[\![\Gamma \vdash (U)\,x : U]\!]\mu(\rho, h, r) \;= \mathsf{if}\ r\,x = \mathit{null} \vee \rho(r\,x) \leq U\ \mathsf{then}\ (\rho, h, [\mathsf{res} : r\,x, \mathsf{exc} : \mathit{null}])\ \mathsf{else}\ \mathit{except}(\rho, h, U, \mathsf{ClassCast})$

$[\![\Gamma \vdash x\ \mathbf{is}\ U : \mathbf{bool}]\!]\mu(\rho, h, r) \;= \mathsf{let}\ v = \mathsf{if}\ r\,x \neq \mathit{null} \wedge \rho(r\,x) \leq U\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ \mathit{false}\ \mathsf{in}\ (\rho, h, [\mathsf{res} : v, \mathsf{exc} : \mathit{null}])$

$[\![\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ E\ \mathbf{in}\ E1 : U]\!]\mu(\rho, h, r) \;=$

$\quad \mathsf{let}\ (\rho_0, h_0, r_0) = [\![\Gamma \vdash E : T]\!]\mu(\rho, h, r)\ \mathsf{in}$

$\quad \mathsf{if}\ r_0\,\mathsf{exc} \neq \mathit{null}\ \mathsf{then}\ (\rho_0, h_0, [\mathsf{res} : \mathit{default}\ U, \mathsf{exc} : r_0\,\mathsf{exc}])\mathsf{else}\ \mathsf{let}\ r_1 = [r,\ x : r_0\,\mathsf{res}]\ \mathsf{in}\ [\![\Gamma \vdash E1 : U]\!]\mu(\rho_0, h_0, r_1)$

$[\![\Gamma \vdash x.m(\overline{y}) : U]\!]\mu(\rho, h, r) \;=$

$\quad \mathsf{if}\ r\,x = \mathit{null}\ \mathsf{then}\ \mathit{except}(\rho, h, U, \mathsf{NullDeref})$

$\quad \mathsf{else}\ \mathsf{let}\ \overline{z} : \overline{T} \rightarrow U = \mathit{mtype}(m, T)\ \mathsf{in}\ \mathsf{let}\ r_1 = [\mathsf{self} : r\,x, \overline{z} : r\,\overline{y}]\ \mathsf{in}\ \mu(\rho(r\,x))m(\rho, h, r_1)$

Figure 3. Semantics of expressions.

$[\![\Gamma \vdash x := E]\!]\mu(\rho, h, r) \;=$

$\quad \mathsf{let}\ (\rho_1, h_1, r_1) = [\![\Gamma \vdash E : T]\!]\mu(\rho, h, r)\ \mathsf{in}\ \mathsf{if}\ r_1\,\mathsf{exc} = \mathit{null}\ \mathsf{then}\ (\rho_1, h_1, [r \mid x : r_1\,\mathsf{res}, \mathsf{exc} : \mathit{null}])\ \mathsf{else}\ (\rho_1, h_1, [r,\ \mathsf{exc} : r_1\,\mathsf{exc}])$

$[\![\Gamma \vdash x.f := y]\!]\mu(\rho, h, r) \;=\; \mathsf{if}\ r\,x \neq \mathit{null}\ \mathsf{then}\ (\rho, [h \mid r\,x.f : r\,y], r)\ \mathsf{else}\ \mathit{except}(\rho, h, r, \mathsf{NullDeref})$

$[\![\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2]\!]\mu(\rho, h, r) \;=\; \mathsf{if}\ r\,x = \mathit{true}\ \mathsf{then}\ [\![\Gamma \vdash S_1]\!]\mu(\rho, h, r) \quad \mathsf{else}\ [\![\Gamma \vdash S_2]\!]\mu(\rho, h, r)$

$[\![\Gamma \vdash \mathbf{var}\ x : T\ \mathbf{in}\ S]\!] \;=\; \mathsf{let}\ (\rho_1, h_1, r_1) = [\![\Gamma, x : T \vdash S]\!]\mu(\rho, h, [r,\ x : \mathit{default}\ T])\ \mathsf{in}\ (\rho_1, h_1, r_1 - x)$

$[\![\Gamma \vdash S_1;\ S_2]\!]\mu(\rho, h, r) \;=\; \mathsf{let}\ (\rho_1, h_1, r_1) = [\![\Gamma \vdash S_1]\!]\mu(\rho, h, r)\ \mathsf{in}\ \mathsf{if}\ r_1\,\mathsf{exc} = \mathit{null}\ \mathsf{then}\ [\![\Gamma \vdash S_2]\!]\mu(\rho_1, h_1, r_1 - \mathsf{exc})\ \mathsf{else}\ (\rho_1, h_1, r_1)$

$[\![\Gamma \vdash \mathbf{throw}\ x]\!]\mu(\rho, h, r) \;=\; \mathsf{if}\ r\,x \neq \mathit{null}\ \mathsf{then}\ (\rho, h, [r,\ \mathsf{exc} : r\,x])\ \mathsf{else}\ \mathit{except}(\rho, h, r, \mathsf{NullDeref})$

$[\![\Gamma \vdash \mathbf{try}\ S_1\ \mathbf{catch}(x : T)\ S_2]\!]\mu(\rho, h, r) \;=$

$\quad \mathsf{let}\ (\rho_1, h_1, r_1) = [\![\Gamma \vdash S_1]\!]\mu(\rho, h, r)\ \mathsf{in}$

$\quad \mathsf{if}\ r_1\,\mathsf{exc} = \mathit{null} \vee \rho(r_1\,\mathsf{exc}) \not\leq T\ \mathsf{then}\ (\rho_1, h_1, r_1)$

$\quad \mathsf{else}\ \mathsf{let}\ r_3 = [r_1 \mid x : r_1\,\mathsf{res}] - \mathsf{exc}\ \mathsf{in}$

$\qquad \mathsf{let}\ (\rho_2, h_2, r_2) = [\![\Gamma, x : T \vdash S_2]\!]\mu(\rho_1, h_1, r_3)\ \mathsf{in}\ (\rho_2, h_2, r_2 - x)$

$[\![\Gamma \vdash \mathbf{try}\ S_1\ \mathbf{finally}\ S_2]\!]\mu(\rho, h, r) \;=$

$\quad \mathsf{let}\ (\rho_1, h_1, r_1) = [\![\Gamma \vdash S_1]\!]\mu(\rho, h, r)\ \mathsf{in}$

$\quad \mathsf{let}\ (\rho_2, h_2, r_2) = [\![\Gamma, x : T \vdash S_2]\!]\mu(\rho_1, h_1, r_1 - \mathsf{exc})\ \mathsf{in}$

$\quad \mathsf{if}\ r_2\,\mathsf{exc} = \mathit{null}\ \mathsf{then}\ (\rho_2, h_2, [r_2,\ \mathsf{exc} : r_1\,\mathsf{exc}])\ \mathsf{else}\ (\rho_2, h_2, r_2)$

Figure 4. Semantics of commands.