Behavioral Subtyping is Equivalent to Modular Reasoning for Object-Oriented Programs

Gary T. Leavens[⋆1] and David A. Naumann[⋆⋆2]

[1] Iowa State University, Ames, IA 50011 USA, leavens@cs.iastate.edu
[2] Stevens Institute of Technology, Hoboken, NJ 07030 USA, naumann@cs.stevens.edu

Abstract. Behavioral subtyping enables modular reasoning about the functional behavior of object-oriented programs. It validates supertype abstraction, that is, modular reasoning about dynamically dispatched method calls, such as $E.m()$, using specifications associated with their receiver's static type, such as the static type of E. For languages with references and mutable objects neither behavioral subtyping nor supertype abstraction has been rigorously formalized as such. Moreover, the standard informal notion of behavioral subtyping has inadequacies. This paper gives a new formalization of behavioral subtyping and supertype abstraction, and a new proof of their equivalence. Our new formalization handles a realistic subset of sequential Java, with classes and interfaces, recursive types, and dynamically-allocated mutable objects.

Keywords: logical foundations of programming paradigms; program specification, modular verification, supertype abstraction, 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, allowing variations in data structures and algorithms to be handled uniformly [10]. They are problematic because a dynamically-dispatched method call, say $E.m()$, seems to require a case analysis to deal with all possible dynamic types of E. (Here the value of E is the *receiver* and m is a method name.) Modular reasoning uses m's specification from E's static type to reason about such calls. This technique is called *supertype abstraction* [8]. While modular type safety conditions for dynamically-dispatched methods are well-known, a straightforward translation into conditions on overriding method specifications is more restrictive than necessary. Hence, for modular reasoning one needs a behavioral notion of subtyping.

Remarkably, there is no previous, rigorous account of behavioral subtyping and its connection with modular reasoning for conventional OO programming languages, although there has been much study (e.g, [1, 4, 8, 11], see [6] for a survey). Some of the current understanding of behavioral subtyping is embodied in program logics [13, 18–20] but is difficult to disentangle from other complications. Some of is also embodied in

⋆ Supported in part by NSF grant CCF-0429567.
⋆⋆ Supported in part by NSF grants CCR-0208984 and CCF-0429894. Corresponding author.

languages and tools such as Eiffel [12] and JML [5], but these have unsoundnesses and incompletenesses, some by engineering design and some for lack of adequate theory.

Our contribution is to provide a rigorous analysis on which one can base more specialized assessments and justifications of specific tools and logics. We show how to formalize a general theory that pertains directly to reasoning about code in languages of practical interest. Fortunately, the theory is orthogonal to the daunting intricacies of current methodologies reasoning about invariants, heap encapsulation, and locality of effects. We consider a fairly big and unrestricted programming language, because some OO language features, such as mutation and type tests, 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 fragments of Java) that embody supertype abstraction in some form (e.g.,[19, 20]). But such results assess the reasoning power of a proof system and expressiveness of an assertion language, rather than assessing and explicating the connection between behavioral subtyping and supertype abstraction.

Our main result is that behavioral subtyping is not only sufficient but necessary for supertype abstraction (Corollary 1). The key insight and our key technical contribution is a purely semantic formulation of supertype abstraction. This new formulation uses two denotational semantics. The actual program semantics, $\mathscr{D}[\![-]\!]$, uses dynamic dispatch for method calls, whereas in the semantics used to formalize supertype abstraction, $\mathscr{S}[\![-]\!]$, method calls are statically dispatched. For a command S, its meaning, $\mathscr{D}[\![S]\!]$, is interpreted in a *method environment*, μ, that gives a meaning to each method at each type. Thus $\mathscr{D}[\![S]\!](\mu)$ is a function from initial states to final states, as is $\mathscr{S}[\![S]\!](\mu)$.

Our new definition of supertype abstraction captures reasoning about commands S using, for each method call, such as $E.m()$, the specifications associated with the static types of the receiver E. Modular reasoning about a command S means proving that $\mathscr{D}[\![S]\!](\mu)$ satisfies some pre/post specification, $spec$, using only the specifications associated with the static types of receivers of method calls. Specifications for all methods are collected in a *specification table*, which gives a specification for each method of each type. To avoid formalizing "reasoning" as such, our definition considers semantic consequences that hold for any μ that satisfies the specification table. Supertype abstraction is thus roughly as follows: If it is true that $\mathscr{S}[\![S]\!](\mu)$ satisfies $spec$, then the actual semantics $\mathscr{D}[\![S]\!](\mu)$ satisfies $spec$, provided μ satisfies the specification table.

Contributions. We make no commitment to particular specification notations or reasoning system but rather give a new, semantic formulation of supertype abstraction (modular reasoning) that idealizes verification in standard program logics and tools. In contrast to previous work, our definition does not rely on derived notions such as substituting one object for another [8, 10, 11], nor is it tied to a proof system [13, 18–21].

We give a new formalization of behavioral subtyping in terms of refinement in a realistic programing model (including classes and interfaces, inheritance, dynamically allocated mutable objects, reference equality, type tests, exceptions, and recursive types). Surprisingly, 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 [11], we use the intrinsic notion of refinement between specifications, defined by quantifying over satisfying implementations. Characterization of refinement in terms of relations between pre- and postconditions is important, but orthogonal. Known results [3, 16] can be used to characterize refinement of specifications of appropriate form [7], improving on the overly restrictive condition found in much work on behavioral subtyping [1, 11, 12]. We also find that abstraction functions are not an integral part of behavioral subtyping (compare, e.g., [4, 11]).

Finally, we prove that behavioral subtyping is sound and semantically complete for supertype abstraction in a realistic language (Corollary 1).

Outline. Technical details about the language (Sect. 2) and specifications (Sect. 3) are followed by a formalization of supertype abstraction, behavioral subtyping, and the main results (Sect. 4). Sect. 6 concludes. See the technical report [7] for more details and related work. *To aid the reviewers, some proofs are included in an Appendix.*

2 A programming language and its semantics

Our technical development uses an idealized OO language that models a large, sequential fragment of class-based languages like Java and C#. The semantics is adapted from [2] but with the addition of interface types and exceptions (but the latter are omitted from this version of the paper). Two improvements streamline the formal development. First, although a distinction is made between expressions E and commands S, both may have effects. Reasoning systems often restrict expressions to be pure; but we treat exceptions in full generality which entails heap effects in expressions. Second, the complication of threading state through the semantics of expressions is mitigated by the uniform use of a general form of *state transformer*, with separate variable declarations for the initial and final state spaces; e.g., the value of an expression is returned in a distinguished variable, res.

The metatheory is standard set theory. Application is written with juxtaposition and associates to the left as in $f\ a\ b$ for a curried f. It binds more tightly than other operators including comma in pairing. Finite mappings are used for typing contexts and variable stores, written like $[x\!:\!C, y\!:\!\mathbf{int}]$ or with brackets omitted. The extension of a finite mapping g to map b to z, where $b \notin \mathit{dom}\, g$, is written $[g,\ b\!:\!z]$. To override the mapping for an element $c \in \mathit{dom}\, g$ we write $[g \mid c\!:\!z]$.

Syntax. The grammar is based on some given sets of names, using the following nomenclature: $C \in$ *ClassName* for names of declared classes (and the distinguished class Object); $I \in$ *InterfaceName* for names of declared interfaces; x, y, f for variable names (for parameters, fields, and locals); and m for method names. Two distinguished variable names may occur in code: self for the receiver object 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;". Class and interface declarations have the following forms:

$$\textbf{class}\ \ C\ \ \textbf{extends}\ \ C\ \ \textbf{implements}\ \ \overline{I}\ \{\ \overline{f\!:\!T}\quad \overline{mdec}\ \} \tag{1}$$

$$\textbf{interface}\ \ I\ \ \textbf{extends}\ \ \overline{I}\ \{\ \overline{f\!:\!T}\quad \overline{msig}\ \} \tag{2}$$

T	$::=$	$C \mid I \mid$ **int**	data type (class C, interface I, or primitive)
$msig$	$::=$	$m(\overline{x}\colon \overline{T})\colon T$	method signature
$mdec$	$::=$	$msig\ \{\ S\ \}$	method declaration
S	$::=$	$x := E \mid x.f := x \mid S;\ S$	assign to variable, to field, sequence
	$\mid$	**var** $x\colon T$ **in** $S \mid$ **ifnz** x **then** S **else** S	local variable block, conditional
E	$::=$	$x \mid$ **null** $\mid 0 \mid 1 \mid \ldots \mid x.f \mid x = x$	variable, literals, field access, equality test
	$\mid$	x **is** $T \mid (T)\ x \mid$ **new** $C()$	type test, type cast, object construction
	$\mid$	$x.m(\overline{x}) \mid$ **let** x **be** E **in** E	method call, sequenced local binding

Fig. 1. Grammar. Punctuation marks including "{" and "}" are terminal symbols.

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 defined in Fig. 1. The syntax is in something like "A-normal form", i.e., subexpressions in various constructs are restricted to be variables. To avoid loss of expressiveness, let-expressions are added.

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 *class table*, i.e., a mapping CT that sends each class name C to its declaration $CT(C)$ and each interface name I to its declaration $CT(I)$.

The typing rules are syntax-directed so the semantics can be defined by recursion on typing derivations. The typing context Γ in judgment $\Gamma \vdash S$ gives names and types of parameters and local variables in scope, including the special variables $\mathsf{self}, \mathsf{res})$. Although it is not explicit in the judgments, typing depends on the whole class table, owing to recursive class declarations. Several auxiliary functions are used to access parts of the class table. The only one used in this version of the paper is *mtype*. If class C has method declaration $m(\overline{x}\colon \overline{T})\colon U\ \{\ S\ \}$ then $\mathit{mtype}(C, m) = \overline{x}\colon \overline{T} \to U$ (similarly for declaration in an interface). And if m is inherited in B from superclass C then $\mathit{mtype}(B, m) = \overline{x}\colon \overline{T} \to U$. The subtype relation $\leq$ is defined inductively by (a) $C \leq D$ if $\mathit{super}\, C = D$ (b) $T \leq I$ if $I \in \mathit{superinterfaces}\, T$, (c) $I \leq \mathsf{Object}$ for all I, and (d) $\leq$ is reflexive and transitive.

A class table CT is *well formed* provided it satisfies standard constraints such as acyclicity of $\leq$, and every method declaration $m(\overline{x}\colon \overline{T})\colon T\ \{S\}$ in $CT(C)$ is typable in the sense that $\Gamma \vdash S$ where $\Gamma = [\mathsf{self}\colon C, \mathsf{res}\colon T, \overline{x}\colon \overline{T}]$. Rules that define $\Gamma \vdash S$ are straightforward. Here is the rule for assignment and two of the rules for expressions.

$$\frac{\Gamma \vdash E\colon T \qquad T \leq \Gamma\, x \qquad x \neq \mathsf{self}}{\Gamma \vdash x := E} \qquad \frac{\Gamma \vdash E\colon T \qquad \Gamma, x\colon T \vdash E1\colon U}{\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ E\ \textbf{in}\ E1\colon U} \tag{3}$$

$$\frac{\Gamma \vdash x\colon T \qquad \mathit{mtype}(T, m) = \overline{z}\colon \overline{T} \to U \qquad \Gamma \vdash \overline{y}\colon \overline{V} \qquad \overline{V} \leq \overline{T}}{\Gamma \vdash x.m(\overline{y})\colon U} \tag{4}$$

Semantic domains. We assume a given set *Ref* of *references*. A *ref context* is a finite partial function ρ that maps references to class names (not interface names). The idea

is that if $o \in \mathit{dom}\,\rho$ then o is allocated and points to an object of type $\rho\, o$. Define $\mathit{RefCtx} = \mathit{Ref} \rightharpoonup \mathit{ClassName}$, where $\rightharpoonup$ denotes finite partial functions.

The domains encode some invariants: well typed values and absence of dangling references. For data type T the domain of values is defined by cases on T:

$$\begin{array}{ll} \mathit{Val}(\mathbf{int}, \rho) = \mathbb{Z} \\ \mathit{Val}(C, \rho) & = \{\mathit{null}\} \cup \{o \mid o \in \mathit{dom}\,\rho \wedge \rho\, o \leq C\} \\ \mathit{Val}(I, \rho) & = \{o \mid \exists C \cdot C \leq I \wedge o \in \mathit{Val}(C, \rho)\} \end{array}$$

For dependent function spaces and pairs, our notation is like that of the PVS theorem prover. *Stores* are dependent functions from variables to type-correct, allocated values:

$$\mathit{Store}(\Gamma, \rho) \;=\; (x : \mathit{dom}\,\Gamma) \to \mathit{Val}(\Gamma\, x, \rho)$$

What this means is that for any $r \in \mathit{Store}(\Gamma, \rho)$, r is a function with domain $\mathit{dom}\,\Gamma$ and $r\, x$ is an element of $\mathit{Val}(\Gamma\, x, \rho)$ for each $x \in \mathit{dom}\,\Gamma$. Next we build up to program states. Here *fields* C is the typing context that declares the fields of C.

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

A *heap* h is map sending each allocated reference o to a store, $h\, o$, of the object's current field values. The most important domain is state transformers:

$$\mathit{STrans}(\Gamma, \Gamma') = (s : \mathit{State}(\Gamma)) \to \{\bot\} \cup \{s' \mid s' \in \mathit{State}(\Gamma') \wedge \mathit{extState}(s, s')\}$$

Relation $\mathit{extState}$ says that allocated objects persist: $\mathit{extState}((\rho, h, r), (\rho', h', r'))$ iff $\rho \subseteq \rho'$. Elements of $\mathit{STrans}(\Gamma, \Gamma')$ are functions that map a state in $\mathit{State}(\Gamma)$ to either $\bot$ or a state in $\mathit{State}(\Gamma')$, with a possibly extended heap. The domain of state transformers subsumes meanings for methods, expressions and commands:

$$\begin{array}{ll} \mathit{SemExpr}(\Gamma, T) & = \mathit{STrans}(\Gamma, [\mathsf{res} : T]) \\ \mathit{SemCmd}(\Gamma) & = \mathit{STrans}(\Gamma, \Gamma) \\ \mathit{SemMeth}(T, m) & = \mathit{STrans}([\mathsf{self} : T, \overline{x} : \overline{T}], [\mathsf{res} : U]) \\ & \quad \text{where } \mathit{mtype}(m, T) = \overline{x} : \overline{T} \to U \end{array}$$

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

$$\mathit{MethEnv} \;=\; (C : \mathit{ClassName}) \times (m : \mathit{Meths}\; C) \to \mathit{SemMeth}(C, m)$$

A method environment μ is defined on pairs (C, m) where C is a class with method m; and $\mu(C, m)$ is a state transformer suitable as the meaning of a method of type $\mathit{mtype}(C, m)$. 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.

A command $\Gamma \vdash S$ denotes a function from method environments to $\mathit{SemCmd}(\Gamma)$ and $\Gamma \vdash E : T$ denotes a function from method environments to $\mathit{SemExpr}(\Gamma, T)$. A complete program CT denotes a method environment obtained as a least fixed point in the straightforward way: $\mathit{State}(\Gamma)$ is ordered discretely, $\mathit{STrans}(\Gamma, \Gamma')$ is ordered

pointwise relative to the flat, $\perp$-lifted order on $State(\Gamma')$, and method environments are ordered pointwise.

In our new formulation of supertype abstraction (i.e., modular reasoning based on static types) we use an extended 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 $Val(I, \rho)$. *Extended method environments* are defined by

$$\mathit{XMethEnv} \;=\; (T : \mathit{RefType}) \times (m : \mathit{Meths}\ T) \to \mathit{SemMeth}(T, m) \quad .$$

Semantics of expressions, commands, and class table. The semantic definitions for expressions and commands are straightforward and mostly omitted. Our meta-language includes notation for the lift monad: $\mathsf{lets}\ x = \alpha\ \mathsf{in}\ \beta$ means $\mathsf{if}\ \alpha = \perp\ \mathsf{then}\ \perp\ \mathsf{else}\ \mathsf{let}\ x = \alpha\ \mathsf{in}\ \beta$ (and let has its ordinary mathematical meaning). We use "lets" because the proof of the main result relies on careful analysis of the semantic definitions.

The semantics of expressions and commands is defined by recursion on typing derivations. For example, here is the semantics of assignment (see the rule in (3)):

$$\begin{array}{l}[\![\Gamma \vdash x := E]\!](\mu)(\rho, h, r) \;= \\ \quad \mathsf{lets}\ (\rho_1, h_1, r_1) = [\![\Gamma \vdash E : T]\!](\mu)(\rho, h, r)\ \mathsf{in}\ \ (\rho_1, h_1, [r \mid x : r_1\,\mathsf{res}])\end{array} \tag{5}$$

If E yields $\perp$ then so does the assignment (owing to lets). Otherwise its value, $r_1\,\mathsf{res}$, is assigned to x.

This definition uses 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 $\mathscr{D}[\![-]\!]$, is the operationally accurate one. It dispatches to a method meaning based on the dynamic type of the receiver. Let $T = \Gamma\, x$ and $\overline{z} : \overline{T} \to U = \mathit{mtype}(m, T)$ as in (4), to define

$$\begin{array}{l}\mathscr{D}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\mu)(\rho, h, r) \;= \\ \quad \mathsf{if}\ r\,x = \mathit{null}\ \mathsf{then}\ \perp\ \mathsf{else}\ \mathsf{let}\ r_1 = [\mathsf{self} : r\,x, \overline{z} : r\,\overline{y}]\ \mathsf{in}\ \mu(\rho(r\,x), m)(\rho, h, r_1)\end{array} \tag{6}$$

The receiver object is $r\,x$, thus $\rho(r\,x)$ is the dynamic type of the object; so to look up in method environment μ the meaning of the dynamically dispatched method we write $\mu(\rho(r\,x), m)$. It is applied to state (ρ, h, r_1) containing the arguments. Since the argument expressions $\overline{y}$ are variables we can write $r\,\overline{y}$ for their values. Because the dynamic type of the receiver is a class (specifically, $\rho\,(r\,x)$), this semantics is based on a normal method environment.

The *static dispatch* semantics of method call $x.m(\overline{y})$ applies a method meaning determined by the static type, $\Gamma\, x$, of the receiver. Since interfaces are included among the static types, the static dispatch semantics is defined in terms of an extended method environment $\dot{\mu}$:

$$\begin{array}{l}\mathscr{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\dot{\mu})(\rho, h, r) \;= \\ \quad \mathsf{if}\ r\,x = \mathit{null}\ \mathsf{then}\ \perp\ \mathsf{else}\ \mathsf{let}\ r_1 = [\mathsf{self} : r\,x, \overline{z} : r\,\overline{y}]\ \mathsf{in}\ \dot{\mu}(\Gamma x, m)(\rho, h, r_1)\end{array} \tag{7}$$

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. In operational terms, the ith element in the chain gives the correct semantics for executions with method call stack bounded in length by i. The details are not relevant in this paper.

3 Specifications and refinement

This section formalizes method specifications and satisfaction by state transformers (in the sense of total correctness). On this basis we define specification tables and satisfaction for them as well as the induced refinement relation between specifications.

In practice, most specifications are written using *two-state* postconditions over program state, with special syntax like "old(x)" to refer to the initial state. A specification of this kind can be desugared into one where the pre- and post-conditions are one-state predicates, using auxiliary variables universally quantified over the Hoare triple. Care must be taken in reasoning with such specifications to avoid soundness pitfalls. For our purposes it is convenient to distinguish auxiliaries from ordinary program variables by considering indexed families of pre/post predicates on program state. We also use the notion of *state transformer type*, written $\Gamma \rightsquigarrow \Gamma'$, for specifications of state transformers in $STrans(\Gamma, \Gamma')$.

Definition 1. *A* simple specification of type $\Gamma \rightsquigarrow \Gamma'$ *is a pair* $(pre, post)$ *such that* pre *is a subset of* $State(\Gamma)$ *and* $post$ *is a subset of* $State(\Gamma')$

A general specification of type $\Gamma \rightsquigarrow \Gamma'$ *is a triple* $(J, pre, post)$ *consisting of a set* J *and indexed families* $pre \in J \to \mathbb{P}(State(\Gamma))$ *and* $post \in J \to \mathbb{P}(State(\Gamma'))$.

A method specification of type (T, m) *is one of type* $[\mathsf{self}\colon T, \overline{x}\colon \overline{T}] \rightsquigarrow [\mathsf{res}\colon U]$ *where* $mtype(m, T) = \overline{x}\colon \overline{T} \to U$. *And a* Γ-specification *is one of type* $\Gamma \rightsquigarrow \Gamma$.

Unqualified, "specification" means general specification unless the context makes it obvious that simple specifications are considered.

Example 1. Given a two-state postcondition $Q \subseteq State(\Gamma) \times State(\Gamma')$ and a precondition $P \subseteq State(\Gamma)$, one obtains a general specification $(J, pre, post)$ by taking J to be $State(\Gamma)$ and defining pre_s and $post_s$, for $s \in State(\Gamma)$, by $pre_s = \{t \mid t = s \wedge s \in P\}$ and $post_s = \{t \mid (s, t) \in Q\}$.

Definition 2. *(*$\models$*) Suppose* $(pre, post)$ *has type* $\Gamma \rightsquigarrow \Gamma'$ *and* $\sigma \in STrans(\Gamma, \Gamma')$. *Then* σ satisfies $(pre, post)$, *written* $\sigma \models (pre, post)$, *iff* $\forall s \cdot s \in pre \Rightarrow \sigma\, s \in post$. *For general specification* $(J, pre, post)$, $\sigma \models (J, pre, post)$ *iff* $\sigma \models (pre_i, post_i)$ *for all* $i \in J$.

A *specification table* ST contains a method specification $ST(T, m)$ having type $mtype(T, m)$ for each ref type T and each $m \in Meths\ T$. It models what might be called the "effective specification", which is typically obtained from declared specifications by means of specification inheritance, context-dependent interpretation of modifies clauses, and invariant disciplines.

Definition 3. *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* $m \in Meths\ C$.

Satisfaction by a normal method environment 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 entire proof obligation imposed on an implementation of m in class C —so $\mu(C,m)$ will satisfy specifications of m in super-classes and super-interfaces provided ST has behavioral subtyping.

Our definition of behavioral subtyping is expressed in terms of a refinement ordering on specifications. Refinement says 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.

For a method m of class U with $mtype(U,m) = \overline{x}\colon\overline{T}\to V$, the relevant state transformers are in $SemMeth(U,m)$, i.e., of type $[\mathsf{self}\colon U, \overline{x}\colon\overline{T}] \rightsquigarrow [\mathsf{res}\colon V]$. For T, a method meaning will have type $[\mathsf{self}\colon T, \overline{x}\colon\overline{T}] \rightsquigarrow [\mathsf{res}\colon V]$ —only the type of self varies.

Definition 4. *($\Vdash^T$) Let $(pre, post)$ be a specification of type $\Gamma \rightsquigarrow \Gamma'$ and let $T \le \Gamma\,\mathsf{self}$. Define $(pre, post)\restriction T$ to be the specification $(pre', post)$, of type $[\Gamma \mid \mathsf{self}\colon T] \rightsquigarrow \Gamma'$, where pre' is defined by $(\rho,h,r) \in pre' \iff r\,\mathsf{self} \le T \wedge (\rho,h,r) \in pre$. For a general specification, $(J, pre, post)\restriction T$ is $(J, pre', post)$ where $pre'_i = pre_i\restriction T$ for all $i \in J$.*

An element $\sigma \in STrans([\Gamma \mid \mathsf{self}\colon T], \Gamma')$ satisfies *$(pre, post)$* under *T, written $\sigma \Vdash^T (pre, post)$, iff $\sigma \models (pre, post)\restriction T$. For a general specification, the restriction is applied at each index.*

Definition 5. *($\sqsupseteq^T$) Let $spec_0$ be a specification of type $\Gamma \rightsquigarrow \Gamma'$ and $spec_1$ be of type $[\Gamma \mid \mathsf{self}\colon T] \rightsquigarrow \Gamma'$ where $T \le \Gamma\,\mathsf{self}$. Then $spec_1$* refines *$spec_0$* with respect to *T, written $spec_1 \sqsupseteq^T spec_0$, iff $\sigma \models spec_1 \Rightarrow \sigma \Vdash^T spec_0$ for all $\sigma \in STrans([\Gamma \mid \mathsf{self}\colon T], \Gamma')$.*

This applies in particular to general specifications. 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 $\sigma \Vdash^T spec_0$. We may omit the superscript on $\sqsupseteq$ just in this case.

Lemma 1 (weak transitivity). *Suppose $spec_0$ is a specification of type $\Gamma \rightsquigarrow \Gamma'$, $spec_1$ is of type $[\Gamma \mid \mathsf{self}\colon T] \rightsquigarrow \Gamma'$ where $T \le \Gamma\,\mathsf{self}$, and $spec_2$ is of type $[\Gamma \mid \mathsf{self}\colon U] \rightsquigarrow \Gamma'$ with $U \le T$. 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.*

4 Supertype abstraction and behavioral subtyping

This section defines behavioral subtyping in terms of the intrinsic refinement order on method specifications and then an alternative formulation (Def. 7) is given. Then supertype abstraction for commands is defined. Finally, the two are proved equivalent.

Definition 6 (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 $ST(C, m) \sqsupseteq^C ST(U, m)$.*

Note that the quantification is over subclasses of U, ignoring interface subtypes of U.

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

Definition 7 (supertype abstraction for method specifications). *Specification table ST has* supertype abstraction for method specifications *iff the following holds for all ref types T and classes C with $C \leq T$, and all $m \in Meths\ T$, and every spec of type mtype(T, m): If $ST(T, m) \sqsupseteq$ spec then $ST(C, m) \sqsupseteq^C$ spec.*

Lemma 2. *A satisfiable specification table ST has behavioral subtyping iff it has supertype abstraction for method specifications.*

Supertype abstraction for commands. Reasoning systems use logic and axiomatic semantics to prove that commands satisfy specifications. A proof that command S satisfies *spec* is considered *modular* if reasoning about method calls in S is based only on the specifications of those methods. Thus our semantic formulation refers to satisfaction of *spec* by the static dispatch semantics of S. But 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.

Definition 8. *Let ST be a specification table.* Supertype abstraction is valid for ST *iff for all $\Gamma \vdash S$ and all general Γ-specifications spec, (8) implies (9), where*

$$\forall \dot{\mu} \in \mathit{XMethEnv} \cdot \dot{\mu} \models ST \Rightarrow \mathscr{S}[\![\Gamma \vdash S]\!](\dot{\mu}) \models \mathit{spec} \tag{8}$$

$$\forall \mu \in \mathit{MethEnv} \cdot \mu \models ST \Rightarrow \mathscr{D}[\![\Gamma \vdash S]\!](\mu) \models \mathit{spec} \tag{9}$$

The idea is that a modular reasoner establishes (8) but it follows that S satisfies *spec* in the sense of the standard semantics, i.e., $\mathscr{D}[\![\Gamma \vdash S]\!](\widehat{\mu}) \models \mathit{spec}$ where $\widehat{\mu}$ is the semantics of the class table —provided that $\widehat{\mu}$ does satisfy ST (i.e., the usual proof obligation that each method implementation satisfies its specification).

Theorem 1. *For any satisfiable ST the following are equivalent.*
(a) ST has supertype abstraction for method specifications (Def. 7).
(b) supertype abstraction is valid for ST (Def. 8).

To prove (b) $\Rightarrow$ (a) we instantiate S in Def. 8 with suitable method call, as follows.

Lemma 3. *If ST is satisfiable then it has supertype abstraction for method specifications if (10) implies (11) for all T and all $m \in Meths\ T$ with $mtype(m, T) = \overline{x}\colon \overline{T} \to U$, and all spec of type (T, m), where*

$$\forall \dot{\mu} \in \mathit{XMethEnv} \cdot \dot{\mu} \models ST \Rightarrow \mathscr{S}[\![\mathsf{self}\colon T, \overline{y}\colon \overline{T} \vdash \mathsf{self}.m(\overline{y})\colon U]\!](\dot{\mu}) \models \mathit{spec} \tag{10}$$

$$\forall \mu \in \mathit{MethEnv} \cdot \mu \models ST \Rightarrow \mathscr{D}[\![\mathsf{self}\colon T, \overline{y}\colon \overline{T} \vdash \mathsf{self}.m(\overline{y})\colon U]\!](\mu) \models \mathit{spec} \tag{11}$$

The straightforward proof goes by unfolding the semantics. The formulation exploits that expression and method meanings are the same domain, so a method specification can be used as an expression specification. (We could avoid expression specifications by using a command $z := \mathsf{self}.m(\overline{y})$.)

The (a) $\Rightarrow$ (b) direction of Theorem 1 amounts to the following.

Lemma 4 (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 $\Gamma \rightsquigarrow \Gamma$, we have that $\forall \dot{\mu} \in XMethEnv \cdot \dot{\mu} \models ST \Rightarrow \mathscr{S}[\![\Gamma \vdash S]\!](\dot{\mu}) \models spec$ implies $\forall \mu \in MethEnv \cdot \mu \models ST \Rightarrow \mathscr{D}[\![\Gamma \vdash S]\!](\mu) \models spec$.*

The proof is by structural induction on S, using the following result for expressions and a technique discussed below.

Lemma 5 (supertype abstraction for expressions). *Suppose ST has supertype abstraction for methods and is satisfiable.*[3] *Then for all Γ, E, T, spec such that $\Gamma \vdash E : T$ and spec is of type $\Gamma \rightsquigarrow [\mathsf{res}: T]$ we have that (12) implies (13) where*

$$\forall \dot{\mu} \in XMethEnv \cdot \dot{\mu} \models ST \Rightarrow \mathscr{S}[\![\Gamma \vdash E : T]\!](\dot{\mu}) \models spec \tag{12}$$

$$\forall \mu \in MethEnv \cdot \mu \models ST \Rightarrow \mathscr{D}[\![\Gamma \vdash E : T]\!](\mu) \models spec \tag{13}$$

Again, the proof is by structural induction on E.

In the proofs of these two lemmas there are three kinds of cases. For method call, the semantics is used to reduce implication (8) $\Rightarrow$ (9) to the implication (12) $\Rightarrow$ (13). For other primitive expressions and commands, the semantics is used to prove (13) directly from (12) (and *mutatis mutandis* for commands), which is easy because the semantics are identical and do not involve the method environment.

The third kind of proof case is compound commands and expressions. For these we appeal to the induction hypothesis for the constituent expressions and commands. It is important that the quantifiers are arranged as they are in Def. 8, so that the induction hypothesis is of the form "for all S and *spec*, (8) $\Rightarrow$ (9)", because for a given S' of the third kind we need multiple instantiations of S and *spec*. Moreover, to obtain suitable specifications to instantiate the induction hypothesis, we decompose the semantic definitions using a little meta-language for state transformers, as follows.

Consider any well formed expression $\Gamma \vdash E : T$ other than a method call. (And *mutatis mutandis* for commands.) Suppose that either $[\![-]\!]$ is the dynamic dispatch semantics and μ a normal method environment or $[\![-]\!]$ is the static dispatch semantics and μ an extended method environment. Then the semantics $[\![\Gamma \vdash E : T]\!](\mu)$ —and the semantics of every other expression and command— can be written as a function of the state transformers denoted by its constituent parts, together with some primitive state transformers of various types, using the following three operations.

Kleisli composition For σ_0 of type $\Gamma_0 \rightsquigarrow \Gamma_1$ and σ_1 of type $\Gamma_1 \rightsquigarrow \Gamma_2$, define $\sigma_0; \sigma_1$ of type $\Gamma_0 \rightsquigarrow \Gamma_2$ by $(\sigma_0; \sigma_1)\, s = (\mathsf{if}\ \sigma_0\, s = \bot\ \mathsf{then}\ \bot\ \mathsf{else}\ \sigma_1(\sigma_0\, s))$.

[3] The satisfiability hypothesis is necessary. Suppose that $ST(I, m)$ is unsatisfiable for some interface type I, but satisfiable at all other types. This falsifies the antecedent of (12) but not of (13).

Alternatives For σ_0, σ_1 of type $\Gamma \rightsquigarrow \Gamma'$ and $P \subseteq \mathit{State}(\Gamma)$, define IF P THEN σ_0 ELSE σ_1 of type $\Gamma \rightsquigarrow \Gamma'$ by $(\text{IF } P \text{ THEN } \sigma_0 \text{ ELSE } \sigma_1)\, s = (\mathsf{if}\ s \in P\ \mathsf{then}\ \sigma_0\, s\ \mathsf{else}\ \sigma_1\, s)$.

Store-pairing For Γ and Γ' with disjoint domains and σ of type $\Gamma \rightsquigarrow \Gamma'$, define the *store-pairing*[4] $\langle \sigma \mid id \rangle$ of type $\Gamma \rightsquigarrow [\Gamma'|\Gamma]$ by

$$\langle \sigma \mid id \rangle(\rho, h, r) = \\ \quad \mathsf{if}\ \sigma(\rho, h, r) = \bot\ \mathsf{then}\ \bot\ \mathsf{else}\ (\rho', h', [r|r']) \quad \text{where } (\rho', h', r') = \sigma(\rho, h, r)$$

For constructs other than method call, $\mathscr{D}[\![-]\!]$ and $\mathscr{S}[\![-]\!]$ are defined as identical functions of the semantics of their constituent expressions/commands. For example, $\mathscr{D}[\![\Gamma \vdash x := E]\!](\mu)$ is a function of $\mathscr{D}[\![\Gamma \vdash E : T]\!](\mu)$ and $\mathscr{S}[\![\Gamma \vdash x := E]\!](\dot{\mu})$ is exactly the same function of $\mathscr{S}[\![\Gamma \vdash E : T]\!](\dot{\mu})$. To see this, recall the semantics (5). We can write $[\![\Gamma \vdash x := E]\!](\mu)$ as the following sequence of state transformers:

$$\Gamma \xrightarrow{\langle\langle([\![\Gamma \vdash E : T]\!](\mu)\,;\, \mathit{rename}) \mid id\rangle} [\Gamma, \mathsf{res}' : T] \xrightarrow{\mathit{assg}} [\Gamma] \tag{14}$$

Here $\mathit{rename} : [\mathsf{res} : T] \rightsquigarrow [\mathsf{res}' : T]$ just renames res to give a context disjoint from Γ so we can use store-pairing. And assg is the state transformer that updates x with the value of res' and drops res' from the store.

Suppose σ is a state transformer of type $\Gamma \rightsquigarrow \Gamma'$ and post is a subset of $\mathit{State}(\Gamma')$. The *weakest precondition of σ with respect to post*, written $\mathit{wp}\,\sigma\,\mathit{post}$, is the subset of $\mathit{State}(\Gamma)$ defined by $\mathit{wp}\,\sigma\,\mathit{post} = \{t \mid \sigma\, t \in \mathit{post}\}$.

Lemma 6 (sequential decomposition). *Suppose σ_i has type $\Gamma_i \rightsquigarrow \Gamma_{i+1}$ for $i = 0, 1$. Let $(\mathit{pre}, \mathit{post})$ be a simple specification of type $\Gamma_0 \rightsquigarrow \Gamma_2$. Define $\mathit{spec}_0 = (\mathit{pre}, (\mathit{wp}\,\sigma_1\,\mathit{post}))$ and $\mathit{spec}_1 = ((\mathit{wp}\,\sigma_1\,\mathit{post}), \mathit{post})$. Then $(\sigma_0; \sigma_1) \models (\mathit{pre}, \mathit{post})$ iff $\sigma_0 \models \mathit{spec}_0$ and $\sigma_1 \models \mathit{spec}_1$. Moreover, for any σ_0', σ_1', if $\sigma_0' \models \mathit{spec}_0$ and $\sigma_1' \models \mathit{spec}_1$ then $\sigma_0'; \sigma_1' \models (\mathit{pre}, \mathit{post})$.*

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. The similar result for conditional decomposition is omitted. (See Appendix.) For store-pairing, the decomposition result needs to use a general specification, because a predicate over $[\Gamma'|\Gamma]$ need not be rectangular, i.e., need not factor using a conjunction of separate conditions on Γ and on Γ'. This is already true with two integer variables and no heap. It is why categories of predicate transformers have very lax products [15].

For $(\rho, h, r') \in \mathit{State}(\Gamma')$ and $r \in \mathit{Store}(\Gamma)$ with Γ disjoint from Γ' we write $(\rho, h, r') + r$ for the evident state in $\mathit{State}([\Gamma|\Gamma'])$.

Lemma 7 (store-pairing decomposition). *Suppose Γ and Γ' have disjoint domains and σ has type $\Gamma \rightsquigarrow \Gamma'$. Let (P, Q) be a simple specification of type $\Gamma \rightsquigarrow [\Gamma'|\Gamma]$. Define general specification $(J, \mathit{pre}, \mathit{post})$ by taking $J = \mathit{Store}(\Gamma)$ and, for $r \in J$, defining pre_r and post_r by $\mathit{pre}_r = \{(\rho, h, q) \in \mathit{State}(\Gamma) \mid q = r \wedge (\rho, h, r) \in P\}$ and $\mathit{post}_r = \{(\rho, h, r') \in \mathit{State}(\Gamma') \mid (\rho, h, r') + r \in Q\}$. Then $\langle \sigma \mid id \rangle \models (P, Q)$ iff $\sigma \models (J, \mathit{pre}, \mathit{post})$. Moreover, $\langle \sigma' \mid id \rangle \models (P, Q)$ for any σ' that satisfies $(J, \mathit{pre}, \mathit{post})$.*

[4] Store-pairing seems *ad hoc*. It is tempting to extend the little calculus of state transformers to include products, with pairing $\Gamma \rightsquigarrow \Gamma' \times \Gamma''$, etc. Then we might exploit the categorical theory of predicate transformers [15]. But to prove the theorem what we need is to decompose specifications; succumbing to the temptation would require us to introduce specifications of type $\Gamma \rightsquigarrow \Gamma' \times \Gamma''$ which would obscure the connection with JML and other logics for Java.

Finally, Theorem 1 can be combined with Lemma 2 to obtain the following.

Corollary 1 (semantic soundness and completeness). *Suppose ST has behavioral subtyping. Suppose $\mathscr{S}[\![\Gamma \vdash S]\!](\dot{\mu})$ can be proved to satisfy some Γ-specification spec, assuming about $\dot{\mu}$ only that it satisfies ST. And suppose that the semantics, $\widehat{\mu}$, of the class table satisfies ST. Then the actual semantics $\mathscr{D}[\![\Gamma \vdash S]\!](\widehat{\mu})$ satisfies spec. Moreover, such reasoning is sound only if ST has behavioral subtyping.*

5 Related work

An influential discussion of the benefits of supertype abstraction is Liskov's invited talk at OOPSLA 1987 [10]. Liskov stated an easily-remembered test for behavioral[5] 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, while subtypes often must change behavior in ways allowed by the supertype's specification [8].

In a program logic, supertype abstraction is embodied by the proof rule for method invocation, which allows one to derive $\{pre_m^T\}\ E.m()\ \{post_m^T\}$ 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). Both techniques aim to produce sound conclusions about the actual semantics. We model both using the static-dispatch semantics $\mathscr{S}[\![E.m()]\!](\mu)$. 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 [4]). These proof obligations are modeled by our specification table.

Several authors have offered definitions of behavioral subtyping, but the most influential definition has been Liskov and Wing's [11]. Their "constraint rule" (from their Figure 4, page 1823) says, "Subtype methods preserve the supertype method's behavior." Paraphrasing (and ignoring invariants and history constraints), this means that for type T to be a behavioral subtype of U: whenever T's method m overrides U's method m, then the usual static typing conditions hold, and for all subtype objects $\mathsf{self}\colon T$,

$$pre_m^U(\mathsf{self}) \Rightarrow pre_m^T(\mathsf{self}) \quad \text{and} \quad post_m^T(\mathsf{self}) \Rightarrow post_m^U(\mathsf{self}). \tag{15}$$

This is intended to be part of a "descriptive and informal" presentation (p. 1813), which has only "informal justifications" (p. 1813). However, even at a conceptual level, it has two problems. First, the postcondition rule in (15) is stronger (i.e., less flexible) than necessary for the soundness of supertype abstraction [4]. It is an inexact approximation of specification refinement, a point closely related to the completeness of rules of Adaptation in Hoare logic [3, 16, 20]. Refinement explains the equivalence between behavioral subtyping and supertype abstraction, as our main result (Corollary 1) shows.

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

Second, Liskov and Wing's "constraint rule" also considers that each type specifies a per-instance invariant and says it must be strengthened in each subtype. But the encapsulation on which object invariants depend can be severely compromised in OO programs due to shared mutable objects etc. State of the art solutions use instead a single global invariant that combines alias control with conditions derived from whatever invariants are explicitly declared. This can be encoded in pre- and post-conditions and is thus modeled by our notion of a specification table.

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. Although behavioral subtyping is defined in terms of pre- and post-conditions, the Subtype Requirement pertains to reasoning only about invariants and history constraints. By contrast, we focus on reasoning about pre-post properties of commands. We also consider the semantics of a programming language, and include arbitrarily nested object structures, whereas they use a model with only atomic values and top-level references.

Much older work on behavioral subtyping, including the pioneering work of America [1] and Meyer [12], is is similar to Liskov and Wing's and has similar limitations.

Several logics have been given for sequential Java fragments that incorporate supertype abstraction [13, 19–21]. 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. 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 [13] considers interfaces (in the sense of Java), and even this misses our insight that interfaces can be exempted from the requirements of behavioral subtyping that must apply to (non-abstract) classes.

Parkinson's work [19] is based on separation logic. Parkinson says "behavioral subtyping" for the standard implications (15) and "specification compatibility" for a proof-theoretic formulation that is closer to the intrinsic refinement ordering [19, Def. 3.5.1]. Pierik [20] gives a more conventional proof system, in particular a proof outline logic with a first-order assertion language using finite sequences for heap expressions. 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.

6 Discussion and Conclusions

The main result, Corollary 1, says that specifications conform to the restriction known as behavioral subtyping if and only if it is sound to reason about method calls in terms of the static type of the receiver. One consequence is that behavioral subtyping is ultimately about the intrinsic ordering on specifications determined by the programming language and notion of satisfaction. For enforcement of behavioral subtyping in a verification logic or axiomatic semantics, what is needed is either a sound criterion for

refinement expressed in terms of pre- and post-conditions (preferably complete, unlike (15), see [3, 16]) or a means to obtain suitable specifications by constructions from arbitrary specifications declared by the programmer.

Our result pertains to method calls that use a method assumed to be correct. To verify total correctness of a method implementation, reasoning about any (mutually) recursive calls is similar to what we formalized but requires the addition of well founded measures. It is pleasant that our characterization of behavioral subtyping does not involve such reasoning, even though our specifications are for total correctness.

Several tools, and the JML specification language, allow declaration of arbitrary specifications but enforce behavioral subtyping by fiat. The effective specification of a method m at type T, modeled by our $ST(T, m)$, is defined as the least upper bound of specifications of m declared all types $U \leq T$. This is known as *specification inheritance* [4, 7].

Liskov and Wing's formulation of behavioral subtyping considers object invariants as well as method specifications. Object invariants play a crucial role in verification in practice, but it has proved quite difficult to develop sound and modular principles for reasoning about invariants. Liskov and Wing's condition works for an invariant that depends only on values of the object's own fields [14]. Useful invariants for Java programs depend on fields of other objects and thus require means to achieve encapsulation in the presence of shared mutable state. State of the art methodologies enforce a single global invariant, formed as a conjunction of object-specific invariants derived from the invariant declarations in classes and using notions like ownership for heap encapsulation [9, 17, 14]. In these works, invariants are thus folded into method specifications.

While operationally justified, our semantics is not compositional at the level of classes. That is, the meaning of a class is not a function of the meanings of other classes, but rather the meaning of each method is a function of the meanings of all methods in all classes. This is fine for the sort of modular reasoning found in verifiers: A method implementation is verified with respect to specifications of all the methods it might call, and in terms of the interface/class names that appear in its field declarations, casts, and type tests. (Our theory allows infinitely many interfaces and classes in a class table so we can also define a universal class table by enumerating all possible class hierarchies.) To adapt our results to a model that is compositional at the level of classes, it might be possible to build on the untyped model of Reus [22].

Our programming language and denotational model were developed in the context of a project on foundations for the JML specification language including relational reasoning to deal with purity, modifies clauses, etc. The semantics has been encoded in the PVS theorem prover and type soundness proved. Machine checking of the results of this paper is underway at the time of writing. It would be interesting to develop similar results using other semantic models that cater for recursive types and relational reasoning.

References

1. P. America. Designing an object-oriented programming language with behavioural subtyping. In *Foundations of Object-Oriented Languages*, volume 489 of *LNCS*, pages 60–90. 1991.

2. A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM*, 52(6):894–960, 2005.
3. 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, 2000.
4. K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *18th International Conference on Software Engineering*, pages 258–267, 1996.
5. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT S.E. Notes*, 31(3):1–38, Mar. 2006.
6. 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, 2000.
7. G. T. Leavens and D. A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report 06-36, Computer Science, Iowa State University, Dec. 2006.
8. G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, 1995.
9. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, pages 491–516, 2004.
10. B. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988.
11. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.
12. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, second edition, 1997.
13. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. 2002.
14. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Programming*, 62(3):253–286, 2006.
15. D. A. Naumann. A categorical model for higher order imperative programming. *Mathematical Structures in Computer Science*, 8(4):351–399, 1998.
16. D. A. Naumann. Calculating sharp adaptation rules. *Inf. Process. Lett.*, 77:201–208, 2001.
17. D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *LICS*, pages 313–323, 2004.
18. D. v. Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In *FME'02*, volume 2391 of *LNCS*, pages 89–105, 2002.
19. M. J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, 2005.
20. C. Pierik. Validation techniques for object-oriented proof outlines. Dissertation, Universiteit Utrecht, 2006.
21. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In *Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 162–176, 1999.
22. B. Reus. Modular semantics and logics of classes. In *Computer Science Logic (CSL)*, volume 2803 of *LNCS*, pages 456–469, 2003.

A Proofs for review

This section gives the most interesting proofs. Sect. B gives a few more which seem less likely to interest the reviewers. The technical report [7] gives complete typing rules and semantic definitions.

The decomposition lemmas are straightforward so we omit their proofs. Here is the statement of the one omitted from the body of the paper.

Lemma 8 (conditional decomposition). *Suppose σ_i has type $\Gamma \rightsquigarrow \Gamma'$, for $i = 0, 1$, and $P \subseteq State(\Gamma)$. Let $(pre, post)$ be a simple specification of type $\Gamma \rightsquigarrow \Gamma'$. Define $spec_0 = (P \cap pre, post)$ and $spec_1 = (pre - P, post)$. Then*

$$\text{IF } P \text{ THEN } \sigma_0 \text{ ELSE } \sigma_1 \models (pre, post) \;\; \textit{iff} \;\; \sigma_0 \models spec_0 \textit{ and } \sigma_1 \models spec_1$$

Moreover, IF P THEN σ_0' ELSE $\sigma_1' \models (pre, post)$ *for any σ_0', σ_1' that satisfy $spec_0, spec_1$.*

Lemma 5(supertype abstraction for expressions). Suppose ST has supertype abstraction for methods and is satisfiable. Then for all Γ, E, T, *spec* such that $\Gamma \vdash E : T$ and *spec* is of type $\Gamma \rightsquigarrow [\mathsf{res} : T]$ we have that (16) implies (17) where

$$\forall \dot{\mu} \in \mathit{XMethEnv} \cdot \dot{\mu} \models ST \Rightarrow \mathscr{S}[\![\Gamma \vdash E : T]\!](\dot{\mu}) \models spec \tag{16}$$

$$\forall \mu \in \mathit{MethEnv} \cdot \mu \models ST \Rightarrow \mathscr{D}[\![\Gamma \vdash E : T]\!](\mu) \models spec \tag{17}$$

Proof By structural induction on E, and cases on E.

For the case $\Gamma \vdash x : T$, the result follows from the much stronger property that for all μ and all $\dot{\mu}$, $\mathscr{S}[\![\Gamma \vdash x : T]\!](\dot{\mu}) = \mathscr{D}[\![\Gamma \vdash x : T]\!](\mu)$. The two semantics are identical in this case: there are no sub-expressions and the semantics of expression x is independent of the method environment. The argument is the same for **null** and other literals, as well as for $x.f$, $x = y$, E **is** T, $(T)\ x$, and **new** $C()$.

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

$$\forall \dot{\mu} \in \mathit{XMethEnv} \cdot \dot{\mu} \models ST \Rightarrow \mathscr{S}[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!](\dot{\mu}) \models spec \tag{18}$$

for *spec* of type $\Gamma \rightsquigarrow [\mathsf{res} : U]$. Without loss of generality we can assume *spec* is a simple specification. (For a general one, apply the argument to all $(pre_i, post_i)$.) Let μ be any normal method environment such that $\mu \models ST$. We must show $\mathscr{D}[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!](\mu) \models spec$. By assumption (18) and satisfiability of ST there is some $\dot{\mu}$ such that

$$\mathscr{S}[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!](\dot{\mu}) \models spec \tag{19}$$

Here is the semantics of rule (3), written using $[\![-]\!]$ and μ' to stand for either $\mathscr{D}[\![-]\!]$ and a normal method environment or $\mathscr{S}[\![-]\!]$ and an extended method environment:

$$\begin{array}{l}
[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!](\mu')(\rho, h, r) \;= \\
\quad \mathsf{lets}\ (\rho_0, h_0, r_0) = [\![\Gamma \vdash E : T]\!](\mu')(\rho, h, r)\ \mathsf{in} \\
\quad \mathsf{let}\ r_1 = [r, x : r_0\, \mathsf{res}]\ \mathsf{in}\ [\![\Gamma, x : T \vdash E1 : U]\!](\mu')(\rho_0, h_0, r_1)
\end{array}$$

It is straightforward to show that $[\![\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ E\ \textbf{in}\ E1 : U]\!](\mu')$ can be written as

$$\Gamma \xrightarrow{\langle([\![\Gamma \vdash E : T]\!](\mu');\, rename) \mid id\rangle} [\Gamma, \mathsf{res}' : T] \xrightarrow{alt} [\mathsf{res} : U]$$

Here $rename$ just renames res, as in (14). And alt is

$$init\,;\, [\![\Gamma, x : T \vdash E1 : U]\!](\mu')$$

where $init : [\Gamma, \mathsf{res}' : T] \rightsquigarrow [\Gamma, x : T]$ sets x to the value of res'. Thus by the decomposition lemmas there are specifications $spec_E$ of type $\Gamma \rightsquigarrow [\mathsf{res} : T]$, $spec_{E1}$ of type $[\Gamma, x : T] \rightsquigarrow [\mathsf{res} : U]$, and $spec_{init}$, $spec_{rename}$ of the evident types, such that (19) holds iff each of the component transformers satisfies its specification.

Since assumption (18) holds for all $\dot{\mu}$, it follows that $\mathscr{S}[\![\Gamma \vdash E : T]\!](\dot{\mu}) \models spec_E$ for all $\dot{\mu}$ and $\mathscr{S}[\![\Gamma \vdash E1 : U]\!](\dot{\mu}) \models spec_{E1}$ for all $\dot{\mu}$. As a consequence, we may appeal to the induction hypothesis for $E, spec_E$ and for $E1, spec_{E1}$. This yields that $\mathscr{D}[\![\Gamma \vdash E : T]\!](\mu) \models spec_E$ and $\mathscr{D}[\![\Gamma, x : T \vdash E1 : U]\!](\mu) \models spec_{E1}$ for our arbitrarily chosen μ. The other component transformers like $rename$ are the same in both the static and dynamic dispatch semantics. Having established that the component transformers of $\mathscr{D}[\![\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ E\ \textbf{in}\ E1 : U]\!](\mu)$ all satisfy the component specifications, we obtain $\mathscr{D}[\![\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ E\ \textbf{in}\ E1 : U]\!](\mu) \models spec$, which was to be proved.

Finally, consider the case of $\Gamma \vdash x.m(\overline{y}) : U$. Recall the static dispatch semantics (7) for methods. Suppose $mtype(m, T) = \overline{z} : \overline{T} \to U$ as in the typing rule for method call. Choose some $\dot{\mu}$ such that $\mathscr{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\dot{\mu}) \models spec$. (Such $\dot{\mu}$ exists owing to satisfiability of ST and the assumption (18).) By decomposition we obtain $spec'$ of type $[\mathsf{self} : T, \overline{z} : \overline{T}] \to U$ such that $\dot{\mu}(T, m) \models spec'$. 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 (18) that $ST(T, m) \sqsupseteq^T spec'$.

Now suppose μ is any normal method environment that satisfies ST and recall the dynamic dispatch semantics (6), 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. By supertype abstraction for methods (Def. 7), $ST(T, m) \sqsupseteq^T spec'$ implies that $C \leq T \Rightarrow ST(C, m) \sqsupseteq^C spec'$ for all C. Since $\mu \models ST$ we have for each $C \leq T$ that $\mu(C, m) \models ST(C, m)$ and thus $\mu(C, m) \models\!\!\models^C spec'$. To complete the proof of $\mathscr{D}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\mu)$ it is not enough to use decomposition backwards; we also unfold the definition of $\models$ and the semantics since $\mu(C, m)$ is used just in case $\rho(r\,x) \leq C$. □

Lemma 4 (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 $\Gamma \rightsquigarrow \Gamma$ we have that $\forall \dot{\mu} \in XMethEnv \cdot \dot{\mu} \models ST \Rightarrow \mathscr{S}[\![\Gamma \vdash S]\!](\dot{\mu}) \models spec$ implies $\forall \mu \in MethEnv \cdot \mu \models ST \Rightarrow \mathscr{D}[\![\Gamma \vdash S]\!](\mu) \models spec$.

Proof By structural induction on S. In the cases that S is $x.f := y$, the semantics using $\mathscr{S}[\![-]\!]$ and $\mathscr{D}[\![-]\!]$ is identical so the proof is immediate. In the cases of conditional and sequence, the argument is by induction following the pattern for let-expression in the proof of Lemma 5. 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 5 for E. □

B Additional proofs

Lemma 1.

Proof To show $spec_2 \sqsupseteq^U spec_0$, assume $\sigma \in STrans([\Gamma \mid \mathsf{self}\colon U], \Gamma')$ and $\sigma \models spec_2$ with the aim to prove $\sigma \models\!\!\models^U spec_0$. From $spec_2 \sqsupseteq^U spec_1$ we get $\sigma \models\!\!\models^U spec_1$; but this doesn't yield $\sigma \models spec_1$, which isn't even defined. Define $\tau \in STrans([\Gamma \mid \mathsf{self}\colon T], \Gamma')$ 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 $\sigma \models\!\!\models^U spec_1$ we get $\tau \models spec_1$. Then by $spec_1 \sqsupseteq^T spec_0$ we get that $\tau \models\!\!\models^T spec_0$. Now $\sigma \models\!\!\models^U spec_0$ follows from $\tau \models\!\!\models^T spec_0$ by definition of τ and $\models\!\!\models$. □

To see that the satisfiability condition is necessary, let $spec_1$ be the simple specification $(pre_1, post_1)$ where $pre_1 = \mathit{true}$ and $post_1$ says that self **is** U. No element of $STrans([\Gamma \mid \mathsf{self}\colon T], \Gamma')$ satisfies $spec_1$. Owing to unsatisfiability we have $spec_1 \sqsupseteq^T spec_0$ for any $spec_0$. Define $spec_2$ to have $pre_2 = \mathit{true} = post_2$. 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 $spec_2 \not\sqsupseteq^U spec_0$.

Lemma 2.

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 $ST(C, m) \sqsupseteq^C ST(T, m)$. Since $ST(T, m)$ is satisfiable, we can apply Lemma 1 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 $ST(C, m) \sqsupseteq^C ST(U, m)$. □

Lemma 7.

Proof Observe that, by definition, $\sigma \models (J, pre, post)$ is equivalent to $\forall r \in Store(\Gamma) \cdot \sigma \models (pre_r, post_r)$ which is equivalent, by definition of satisfaction, to

$$\forall r \in Store(\Gamma) \cdot \forall \rho, h, q \cdot (\rho, h, q) \in pre_r \Rightarrow \sigma(\rho, h, q) \in post_r$$

By definition of pre and $post$ this is equivalent to

$$\forall r \in Store(\Gamma) \cdot \forall \rho, h, q \cdot r = q \wedge (\rho, h, q) \in P \Rightarrow (\sigma(\rho, h, q) + r) \in Q$$

which is logically equivalent to $\forall \rho, h, r \cdot (\rho, h, r) \in P \Rightarrow (\sigma(\rho, h, r) + r) \in Q$ and since $\bot \notin Q$ this amounts to $\langle \sigma \mid id \rangle \models (P, Q)$. We leave to the reader the argument why $\langle \sigma' \mid id \rangle \models (P, Q)$ for any σ' that satisfies $(J, pre, post)$. □

Proof of Lemma 3.

Proof For all T, m, all *spec* of type *mtype*(m, T), and all $C \leq T$, we must show that $ST(T, m) \sqsupseteq^T$ *spec* implies $ST(C, m) \sqsupseteq^C$ *spec*. 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 5, 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 $\mathscr{S}[\![[\mathsf{self}\colon T, \overline{y}\colon \overline{T}, z\colon U] \vdash z := \mathsf{self}.m(\overline{y})]\!](\dot{\mu})$ satisfies *spec*. So if we instantiate the antecedent (10) 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 (11) with $spec := ST(T, m)$, which, for any $C \leq T$, boils down to $\forall \sigma \in SemMeth(C, m) \cdot \sigma \models ST(C, m) \Rightarrow \sigma \models\!\!\!\models^C ST(T, m)$, whence $ST(C, m) \sqsupseteq^C ST(T, m)$. $\square$