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

Gary T. Leavens and David A. Naumann

TR #06-36
December 22, 2006

Keywords: Behavioral subtyping, modular reasoning, object-oriented programming, recursive types, denotational semantics, functional specification, Java Modeling Language (JML).

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 — 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 is Equivalent to Modular Reasoning for Object-oriented Programs

Gary T. Leavens[*]
Dept. of Computer Science
Iowa State University
Ames, IA 50011 USA
leavens@cs.iastate.edu

David A. Naumann[†]
Dept. of Computer Science
Stevens Institute of Technology
Hoboken, NJ USA
naumann@cs.stevens.edu

December 22, 2006

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 and the standard informal notion has inadequacies. This paper formalizes behavioral subtyping and introduces a new formalization of modular reasoning, called supertype abstraction. A Java-like sequential language is considered, with classes and interfaces, recursive types, first-class exceptions and handlers, and dynamically allocated mutable heap objects; the semantics is designed to serve as foundation for the Java Modeling Language (JML), a widely used specification language. Behavioral subtyping is characterized as sound and semantically complete for reasoning with supertype abstraction.

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 [39]. 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. (Here the value of expression E is the receiver object and m is a method name.) The basic idea of modular reasoning uses the specification of m from E's static type to reason about such calls. This technique is called *supertype abstraction* [34]. It is a generalization of typechecking, since it imposes constraints on implementations of at all subtypes of the static type of E. While modular type safety conditions for dynamically-dispatched methods are well-known [20], a straightforward translation into conditions on overriding method specifications, while sound, is more restrictive than necessary. 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 [3, 4, 5, 17, 22, 23, 35, 34, 40, 41, 60] (see [33] for a survey). Some of the current understanding of behavioral subtyping is embodied in program logics [42, 52, 55, 56, 59] but is difficult to disentangle from various other complications. Some of the current understanding is embodied languages and tools such as Eiffel [41], JML [32], ESC/Java [24], and Spec# [12, 11]. But these have unsoundnesses and incompletenesses, some by engineering design and some for lack of adequate theory and methodology.

On one hand, behavioral subtyping has been rigorously studied in restrictive and simplified programming models (e.g., [35, 60]). On the other hand, various embodiments have been implemented in static and runtime

[*]Supported in part by NSF grant CCF-0429567.
[†]Supported in part by NSF grants CCR-0208984 and CCF-0429894.

verification tools and logics that apply to rich specification and programming languages such as Java and JML [32] and Eiffel [41]. 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 [11, 37, 50, 15, 51, 55, 14]— 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 fragments of Java) that embody supertype abstraction in some form (e.g.,[56, 55]). 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).

Our main result is that supertype abstraction is equivalent to behavioral subtyping (Corollary 4.12). The key insight that led to our theorem 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, interface and class types, dynamically allocated mutable heap objects, first-class exceptions, inheritance, reference equality, type tests, and recursive types. The proof that behavioral subtyping implies supertype abstraction goes by structural induction on syntax, making novel and slightly intricate use of weakest preconditions.

To organize the proof we develop a little meta-language for state transformers. Interested readers will see that the theorem could be generalized to other program constructs defined in the little calculus and to other semantic domains. For our purposes it is important to include the somewhat intricate features of Java-like languages listed just above. We are interested to know whether our result can be generalized to a framework like that being developed by Power and Plotkin [58, 61]. The denotational semantics used here does have the virtue of being so concrete and operationally transparent that adequacy with respect to operational semantics is obvious. But a higher level of abstraction could facilitate study of variations —e.g., to model garbage collection and its impact [19]— and extensions such as concurrency.

We make no commitment to particular specification notations or reasoning system but rather formulate modular reasoning semantically in a way that idealizes what is found in logics and tools including those based on first-order assertion languages [2, 12, 24, 56], those that allow higher order assertions [18, 29], and more exotic logics [14, 64].

This paper makes the following contributions.

- We give a semantic definition of supertype abstraction, which idealizes what is found in logics and verification tools for Java-like languages. In contrast to previous work, our definition does not rely on derived notions such as substituting one object for another [35, 39, 40], nor is it tied to a proof system [42, 52, 55, 56, 59].

- We formalize behavioral subtyping in terms of refinement of observable behavior in a realistic programming model. 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 [40], 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 it is a separate issue. Known results [21, 45, 53] can be used to characterize refinement of specifications of appropriate form, improving on the overly restrictive condition found in much work on behavioral subtyping [4, 5, 23, 40, 41].

- 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., [35, 40]).

- We characterize behavioral subtyping as being sound and semantically complete for supertype abstraction (Corollary 4.12). It was by seeking completeness that we were led to the surprising findings noted in the preceding items.

Some inessential proofs and other details are included in appendices.

Related work and synopsis. An influential discussion of the benefits of supertype abstraction is Liskov's invited talk at OOPSLA 1987 [39]. 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" [35, Chapter 1].

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 in S. Specifications for all methods are collected in what we call a *specification table*.[2] 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 (or nominal [34]) 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. 4.4.)

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., [36]). 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 is a property of the specification table.

Several authors have offered definitions of behavioral subtyping, but the most influential definition has been Liskov and Wing's [40]. Their "constraint rule" (from their Figure 4, page 1823), says that "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 [20] hold, and for all subtype objects $\mathsf{self}: T$,

$$U\text{'s precondition for } m \text{ implies } T\text{'s precondition, i.e. } pre_m^U(\mathsf{self}) \Rightarrow pre_m^T(\mathsf{self}) \quad (1)$$

$$\text{and } T\text{'s postcondition implies } U\text{'s, i.e. } post_m^T(\mathsf{self}) \Rightarrow post_m^U(\mathsf{self}). \quad (2)$$

This 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. In particular, the postcondition rule (2) is stronger (i.e., less flexible) than necessary for the soundness of supertype abstraction [22]: it is an inexact approximation of refinement between specifications. It is refinement that explains the equivalence between behavioral subtyping and supertype abstraction, as our main result (Corollary 4.12) shows. Our formulation of behavioral subtyping (Def. 4.1) is in terms of the intrinsic refinement order on specifications. As we discuss later, characterizations of specification refinement in terms pre- and post-conditions are sensitive both to the form of specifications and to some features of program semantics such as nondeterminacy.

[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, [51, 56]), this technique requires justification with respect to the actual program semantics.

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 [27] is 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 (see Sect. 5) —this is modeled by our specification tables.

Liskov and Wing's paper [40] 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. 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 a programming language, and include pointer data structures, whereas they use a model with only atomic values and top-level references.

Leavens and Dhara [33] survey a lot of older work on behavioral subtyping, including the pioneering work of America [4, 5] and Meyer [41]. 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 in our sense [42, 55, 56, 59]. 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 [42] 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 [55] is based on separation logic [51]. Parkinson says "behavioral subtyping" for the standard implications (1) and (2) and "specification compatibility" for a proof-theoretic formulation that is closer to the intrinsic refinement ordering [55, Def. 3.5.1]. Pierik [56] 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 (cf. [45]). 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.

2 A programming language and its semantics

The technical development uses an idealized object-oriented language that models a large fragment of the class-based languages like Java and C#. Constructors are omitted but can be treated as syntactic sugar. 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 [10] which in turn draws on [28] for formalization of syntax including the class table. Because the language is essentially defunctionalized [8, 63], a denotational semantics can be given using a straightforward hierarchy of inductively defined domains; there are no non-trivial domain equations to solve. However, the semantics is not compositional at the level of classes; see Sect. 5.

Three features of the semantics streamline the formal development but may be unexpected. 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 include 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 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 $\perp$ or a state. But that state includes a special variable exc, to encode a disjoint sum: the value of exc is either null, which signifies normal termination, or a reference to the exception object. Variable exc is not allowed to occur in the program text but is used in specifications (which models the **signals** clause in JML). Manipulation of res and exc in the semantics corresponds transparently to operational semantics using a stack of activation frames. The second and third features lessen the number of domain definitions, which is especially helpful for working with logical relations

T	$::=$	$C \mid I \mid$ **bool** $\mid$ **int**	data type
$msig$	$::=$	$m(\overline{x} : \overline{T}) : T$	method signature
$mdec$	$::=$	$msig$ { S }	method declaration
S	$::=$	$x := E \mid x.f := x$	assign to variable, to field
	$\mid$	**var** $x : T$ **in** S	local variable block
	$\mid$	$S;\ S \mid$ **if** x **then** S **else** S	sequence, conditional
	$\mid$	**throw** $x \mid$ **try** S **catch**$(x : T)\ S$	throw, handle exception
	$\mid$	**try** S **finally** S	finalize regardless of exception
E	$::=$	$x \mid$ **null** $\mid$ **true** $\mid 0 \ldots$	variable, literals
	$\mid$	$x.f \mid x = x$	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

Figure 1: Grammar. Bold keywords and punctuation marks including "{" and "}" are terminal symbols.

(compare [10] which uses a similar language). Relations are not used in this paper but the semantics is also being used in other studies of JML.

The metatheory is standard set theory and we often use partial functions, treated as sets of pairs. Unqualified, the term *function* means total function. 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. We tend to refrain from unnecessary parentheses around arguments. Finite mappings are used for typing contexts and variable stores, written like $[x : C, y : \textbf{int}]$ 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 $[g \mid c : z]$.

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

$C \in ClassName$	names of declared classes
$I \in InterfaceName$	names of declared interfaces
x, y, f	variable names (for parameters, fields, and locals)
m	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;". There is one distinguished interface name, Thr, and three distinguished class names: Object, NullDeref and ClassCast. The last two implement Thr, the supertype for all exceptions and therefore the type of the special variable exc. Since other classes can be subtypes of Thr, exception objects can have arbitrary references to and from other objects.

Class and interface declarations have the following forms:

$$\textbf{class}\ C\ \textbf{extends}\ C\ \textbf{implements}\ \overline{I}\ \{\ \overline{f : T}\quad \overline{mdec}\ \} \tag{3}$$

$$\textbf{interface}\ I\ \textbf{extends}\ \overline{I}\ \{\ \overline{f : T}\quad \overline{msig}\ \} \tag{4}$$

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" [65], i.e., subexpressions in various constructs are restricted to be variables. To avoid loss of expressiveness, let-expressions are added; e.g., a general equality test $E_1 = E_2$ can be desugared $(-^o)$ by the rule $(E_1 = E_2)^o =$ **let** x **be** E_1^o **in let** y **be** E_2^o **in** $x = y$ which preserves order of evaluation and propagation of exceptions.

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 $RefType = ClassName \cup InterfaceName$. 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 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 $\mathsf{self}, \mathsf{res}$). Although it is not explicit in the judgements, typing depends on the whole class table, owing to recursive class declarations. Several functions access parts of the class table. Suppose $CT(C)$ is as in (3), then we define $\mathit{super}\, C = D$ and $\mathit{superinterfaces}\, C = \overline{I}$. For a method declaration $m(\overline{x}:\overline{T}):T_1\ \{\ S\ \}$ in class C, define $\mathit{mtype}(C,m) = \overline{x}:\overline{T}\to T_1$. 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 $\mathit{superinterfaces}\, I = \overline{I}$ and $\mathit{mtype}(I,m)$ is defined the same as for classes. For declared fields we define $\mathit{dfields}\, C = \overline{f:T}$ and similarly $\mathit{dfields}\, I$. To include inherited fields, 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 $\mathit{fields}\, I = \mathit{dfields}\, I \cup \mathit{fields}(\mathit{superinterfaces}\, I)$. We define $\mathit{Meths}\ T = \{m \mid \mathit{mtype}(T,m) \text{ is defined }\}$. Note that $\mathit{mtype}(T,m)$ is defined just when T is a ref type that declares or inherits m.

The subtype relation $\leq$ is defined inductively by (a) $C \leq D$ if $\mathit{super}\, C = D$ (note that $\mathit{super}\, \mathsf{Object}$ is undefined), (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 $\leq$. 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, according to Java, but the usual contra/covariant rule would cause no problem here. Rules that define $\Gamma \vdash S$ are straightforward (see Appendix A). Here is the rule for assignment and two of the rules for expressions.

$$\frac{\Gamma \vdash E:T \qquad T \leq \Gamma\, x \qquad x \neq \mathsf{self}}{\Gamma \vdash x := E} \qquad\qquad \frac{\Gamma \vdash E:T \qquad \Gamma, x:T \vdash E1:U}{\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ E\ \mathbf{in}\ E1:U} \tag{5}$$

$$\frac{\Gamma \vdash x:T \qquad \mathit{mtype}(T,m) = \overline{z}:\overline{T}\to U \qquad \Gamma \vdash \overline{y}:\overline{V} \qquad \overline{V} \leq \overline{T}}{\Gamma \vdash x.m(\overline{y}):U} \tag{6}$$

Semantic domains. 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 (and not interface 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 finite partial functions. 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'.

The domains encode important invariants: well typed values and absence of dangling references. (We could easily add that self is not null and is immutable, but need not.)

For data type T the domain of values is defined by cases on T:

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

We use a PVS-like [54] notation for dependent function spaces or dependent pairs. For example, stores are dependent functions from variables to type-correct and allocated values:

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

What this means is that for any $r \in \mathit{Store}(\Gamma,\rho)$, the domain of r is $\mathit{dom}\,\Gamma$ 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.

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

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:

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

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

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

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

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

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 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.

A command in context $\Gamma \vdash S$ will denote a function from method environments to $SemCommand(\Gamma)$ and $\Gamma \vdash E : T$ will denote a function from method environments to $SemExpr(\Gamma, T)$. A complete program CT denotes a method environment obtained as a least fixed point in the straightforward way: $State(\Gamma)$ is ordered discretely, $STrans(\Gamma, \Gamma')$ is ordered pointwise relative to the flat, $\bot$-lifted order on $State(\Gamma')$, and method environments are ordered pointwise. The command and expression semantics is easily shown to be continuous in both constituents and the method environment.

In the formulation of 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

$$XMethEnv \ = \ (T : RefType) \times (m : Meths\ T) \rightarrow SemMeth(T, m) \quad .$$

Semantics of expressions, commands, and class table. The semantic definitions for expressions and commands are straightforward and mostly relegated to Appendix B. To streamline the definitions, our meta-language includes notation for the lift monad: $\mathsf{lets}\ x = \alpha\ \mathsf{in}\ \beta$ abbreviates $\mathsf{if}\ \alpha = \bot\ \mathsf{then}\ \bot\ \mathsf{else}\ \mathsf{let}\ x = \alpha\ \mathsf{in}\ \beta$ (and let has its ordinary mathematical meaning). We are a bit pedantic in using keyword lets since 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 (5)):

$$\begin{array}{lcl}
[\![\Gamma \vdash x := E]\!](\mu)(\rho, h, r) & = & \mathsf{lets}\ (\rho_1, h_1, r_1) = [\![\Gamma \vdash E : T]\!](\mu)(\rho, h, r)\ \mathsf{in} \\
& & \mathsf{if}\ r_1\,\mathsf{exc} = null\ \mathsf{then}\ (\rho_1, h_1, [[r \mid x : r_1\,\mathsf{res}],\ \mathsf{exc}: null]) \\
& & \mathsf{else}\ (\rho_1, h_1, [r,\ \mathsf{exc}: r_1\,\mathsf{exc}])
\end{array} \tag{7}$$

If E yields $\bot$ then so does the assignment (owing to lets). If E throws no exception then its value, $r_1\,\mathsf{res}$, is assigned to x and the store is also is extended with exc mapped to $null$. Otherwise, the final state is extended with exc mapped to the exception, $r_1\,\mathsf{exc}$.

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 based on the dynamic type of the receiver. Let $T = \Gamma\, x$ and $\overline{z}: \overline{T} \rightarrow U = mtype(m,\, T)$ as in the typing rule (6). Define

$$\begin{array}{lcl}
\mathcal{D}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\mu)(\rho, h, r) & = & \mathsf{if}\ r\,x = null\ \mathsf{then}\ except(\rho, h, U, \mathsf{NullDeref}) \\
& & \mathsf{else\ let}\ r_1 = [\mathsf{self}: r\,x,\ \overline{z}: r\,\overline{y}]\ \mathsf{in}\ \mu(\rho(r\,x), m)(\rho, h, r_1)
\end{array} \tag{8}$$

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 helping function $except$ simply builds a state with exc set to a new $\mathsf{NullDeref}$ object (see Appendix B).

The *static dispatch* 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 $\dot{\mu}$:

$$\mathcal{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\dot{\mu})(\rho, h, r) \;=\; \begin{array}{l}\textsf{if } r\,x = \mathit{null} \textsf{ then } \mathit{except}(\rho, h, U, \textsf{NullDeref})\\ \textsf{else let } r_1 = [\textsf{self} : r\,x, \overline{z} : r\,\overline{y}] \textsf{ in } \dot{\mu}(T, m)(\rho, h, r_1)\end{array} \tag{9}$$

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, applied to the preceding approximation. In operational terms, the ith element in the chain gives the correct semantics for executions with method call stack bounded in length by i. Details are in Appendix B.

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 (cf. [6]). 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*, notation $\Gamma \leadsto \Gamma'$, for specifications of state transformers in $\mathit{STrans}(\Gamma, \Gamma')$.

Definition 3.1 A *simple specification of type* $\Gamma \leadsto \Gamma'$ is a pair $(\mathit{pre}, \mathit{post})$ such that pre is a subset of $\mathit{State}(\Gamma)$ and post is a subset of $\mathit{State}(\Gamma')$

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

A *method specification of type* (T, m) is one of type $[\textsf{self} : T, \overline{x} : \overline{T}] \leadsto [\textsf{res} : U, \textsf{exc} : \textsf{Thr}]$ where $\mathit{mtype}(m, T) = \overline{x} : \overline{T} \to U$. And a Γ*-specification* is one of type $\Gamma \leadsto [\Gamma,\ \textsf{exc} : \textsf{Thr}]$.

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

Example 3.2 Given a two-state postcondition $Q \subseteq \mathit{State}(\Gamma) \times \mathit{State}(\Gamma')$ and $P \subseteq \mathit{State}(\Gamma)$ one obtains a general specification $(J, \mathit{pre}, \mathit{post})$ by taking J to be $\mathit{State}(\Gamma)$ and defining pre_s and post_s, for $s \in \mathit{State}(\Gamma)$, by $\mathit{pre}_s = \{t \mid t = s \wedge s \in P\}$ and $\mathit{post}_s = \{t \mid (s, t) \in Q\}$.

Definition 3.3 ($\models$) Let $(\mathit{pre}, \mathit{post})$ be a specification of type $\Gamma \leadsto \Gamma'$ and σ be in $\mathit{STrans}(\Gamma, \Gamma')$. Then σ *satisfies* $(\mathit{pre}, \mathit{post})$, written $\sigma \models (\mathit{pre}, \mathit{post})$, iff $\forall s \cdot s \in \mathit{pre} \Rightarrow \sigma\, s \in \mathit{post}$. For general $(J, \mathit{pre}, \mathit{post})$, let $\sigma \models (J, \mathit{pre}, \mathit{post})$ iff $\sigma \models (\mathit{pre}_i, \mathit{post}_i)$ for all $i \in J$.

A *specification table*, ST, contains a method specification $\mathit{ST}(T, m)$ of type $\mathit{mtype}(T, m)$ for each ref type T and each $m \in \mathit{Meths}\ T$. 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 [38, 42], specification inheritance [22, 31, 67, 68], invariant disciplines [11, 37, 43, 50], etc. (See Sect. 5).

Definition 3.4 Let ST be a specification table. An extended method environment $\dot{\mu}$ *satisfies* ST, written $\dot{\mu} \models \mathit{ST}$, iff $\dot{\mu}(T, m) \models \mathit{ST}(T, m)$ for all ref types T and $m \in \mathit{Meths}\ T$.

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

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 $\mathit{ST}(C, m)$ 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.

The behavioral subtyping property is expressed in terms of a refinement ordering on specifications: it 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. (This is explicit in the proof obligation for method bodies in proof systems, e.g. [42, 57].)

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

Definition 3.5 **($\Vdash^T$)** Let $(pre, post)$ be a specification of type $\Gamma \rightsquigarrow \Gamma'$ and let $T \leq \Gamma\,\mathsf{self}$. Define $(pre, post){\downharpoonleft}T$ to be the specification $(pre', post)$, of type $[\Gamma \mid \mathsf{self} : T] \rightsquigarrow \Gamma'$, where pre' is defined by $(\rho, h, r) \in pre' \iff r\,\mathsf{self} \leq T \wedge (\rho, h, r) \in pre$. For a general specification, $(J, pre, post){\downharpoonleft}T$ is $(J, pre', post)$ where $pre'_i = pre_i{\downharpoonleft}T$ for all $i \in J$.

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

Definition 3.6 **($\sqsupseteq^T$)** 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 $\sigma \models spec_1 \Rightarrow \sigma \Vdash^T spec_0$ for all $\sigma \in STrans([\Gamma \mid \mathsf{self} : T], \Gamma')$.

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 3.7 (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.[3]

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 is given. Then supertype abstraction for commands is defined. Finally, the two are proved equivalent.

Definition 4.1 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 sub-classes 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 4.2 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 $spec$ 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 4.3 A satisfiable specification table ST has behavioral subtyping iff it has supertype abstraction for method specifications.

[3]To see that the satisfiability condition is necessary, let $spec_1$ be the simple specification $(pre_1, post_1)$ where $pre_1 = true$ and $post_1$ says 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 $spec_2 \not\sqsupseteq^U spec_0$.

Supertype abstraction for commands. Reasoning systems use logic and axiomatic semantics to prove that commands satisfy specifications. A proof that $\Gamma \vdash S$ satisfies *spec* is considered *modular* provided 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.

Definition 4.4 Let ST be a specification table for CT. *Supertype abstraction is valid for* ST, CT iff for all $\Gamma \vdash S$ and all general Γ-specifications *spec*, (10) implies (11), where

$$\forall \dot{\mu} \in \mathit{XMethEnv} \cdot \dot{\mu} \models ST \;\Rightarrow\; \mathcal{S}[\![\Gamma \vdash S]\!](\dot{\mu}) \models \mathit{spec} \tag{10}$$

$$\forall \mu \in \mathit{MethEnv} \cdot \mu \models ST \;\Rightarrow\; \mathcal{D}[\![\Gamma \vdash S]\!](\mu) \models \mathit{spec} \tag{11}$$

The idea is that a modular reasoner establishes (10) but it is then a consequence that S satisfies *spec* in the sense of the standard semantics, i.e., $\mathcal{D}[\![\Gamma \vdash S]\!](\widehat{\mu}) \models \mathit{spec}$ 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). In fact, owing to modularity of reasoning about satisfaction as described by (10), the stronger conclusion (11) can be drawn.

Theorem 4.5 For any satisfiable ST the following are equivalent.
(a) ST has supertype abstraction for method specifications (Def. 4.2).
(b) supertype abstraction is valid for ST, CT (Def. 4.4).

We sketch the argument here and return to it after laying some groundwork. For (b) $\Rightarrow$ (a) we instantiate S in Def. 4.4 with suitable method call and unfold the semantics thereof.

For (a) $\Rightarrow$ (b) we 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 structural induction. There are three kinds of cases: For method call, the semantics is used to reduce implication (10) $\Rightarrow$ (11) to the implication given by the supertype abstraction property. For other primitive expressions and commands, the semantics is used to prove (11) directly from (10), which is easy since the semantics are identical, not involving the method environment. The third kind is compound commands and expressions. For these we appeal to the induction hypothesis for the constituent expressions and commands; to obtain suitable specifications for this purpose we analyze the semantic definitions using a little meta-language. It is important that the quantifiers are arranged as they are in Def. 4.4, so that the induction hypothesis is of the form "for all *spec* and S, (10) $\Rightarrow$ (11)", because for a given S of the third kind we need multiple instantiations of *spec* and S.

A little calculus of state transformers. Consider any well formed expression $\Gamma \vdash E : T$ other than a method call. 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 Given 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 $(\sigma_0;\sigma_1)\, s \;=\; (\textsf{if}\ \sigma_0\, s = \bot\ \textsf{then}\ \bot\ \textsf{else}\ \sigma_1(\sigma_0\, s))$.

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

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

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

[4]Store-pairing seems ad hoc. It is tempting to extend the little calculus of state transformers to include products in a general form, with pairing $\Gamma \rightsquigarrow \Gamma' \times \Gamma''$ etc. Then the corresponding predicate transformers would be available from the categorical theory of predicate transformers [25, 44]. But to prove the theorem what we need is to decompose specifications; succumbing to the temptation would require us to introduce specifications of type $\Gamma \rightsquigarrow \Gamma' \times \Gamma''$ or else a nontrivial encoding of pairs of heaps into a single heap (as done in [48]) —at best, this would obscure the connection with JML and extant logics for reasoning about Java programs.

For constructs other than method call, $\mathcal{D}[\![-]\!]$ and $\mathcal{S}[\![-]\!]$ are defined as identical functions of the semantics of their constituent expressions/commands. For example, $\mathcal{D}[\![\Gamma \vdash x := E]\!](\mu)$ is a function of $\mathcal{D}[\![\Gamma \vdash E : T]\!](\mu)$ and $\mathcal{S}[\![\Gamma \vdash x := E]\!](\dot{\mu})$ is exactly the same function of $\mathcal{S}[\![\Gamma \vdash E : T]\!](\dot{\mu})$. To see this, recall the semantics (7). We can write $[\![\Gamma \vdash x := E]\!](\mu)$ as the following sequence of state transformers:

$$\Gamma \xrightarrow{\langle\langle([\![\Gamma \vdash E : T]\!](\mu)\,;\, \mathit{rename}) \mid \mathit{id}\rangle} [\Gamma, \mathsf{res}' : T, \mathsf{exc} : \mathsf{Thr}] \xrightarrow{\mathit{assg}} [\Gamma, \mathsf{exc} : \mathsf{Thr}] \tag{12}$$

Here $\mathit{rename} : [\mathsf{res} : T, \mathsf{exc} : \mathsf{Thr}] \rightsquigarrow [\mathsf{res}' : T, \mathsf{exc} : \mathsf{Thr}]$ just renames res to give a context[5] disjoint from Γ so we can use store-pairing. And assg is the state transformer that updates x with the value of res' if exc is null and in either case 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 4.6 (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. Similarly for the following.

Lemma 4.7 (conditional decomposition) Suppose σ_i has type $\Gamma \rightsquigarrow \Gamma'$, for $i = 0, 1$, and $P \subseteq \mathit{State}(\Gamma)$. Let $(\mathit{pre}, \mathit{post})$ be a simple specification of type $\Gamma \rightsquigarrow \Gamma'$. Define $\mathit{spec}_0 = (P \cap \mathit{pre}, \mathit{post})$ and $\mathit{spec}_1 = (\mathit{pre} - P, \mathit{post})$. Then

$$\text{IF } P \text{ THEN } \sigma_0 \text{ ELSE } \sigma_1 \models (\mathit{pre}, \mathit{post}) \quad \text{iff} \quad \sigma_0 \models \mathit{spec}_0 \text{ and } \sigma_1 \models \mathit{spec}_1$$

Moreover, IF P THEN σ_0' ELSE $\sigma_1' \models (\mathit{pre}, \mathit{post})$ for any σ_0', σ_1' that satisfy $\mathit{spec}_0, \mathit{spec}_1$.

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 [25, 44].

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 4.8 (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 $\langle\sigma \mid \mathit{id}\rangle \models (P, Q)$ iff $\sigma \models (J, \mathit{pre}, \mathit{post})$. Moreover, $\langle\sigma' \mid \mathit{id}\rangle \models (P, Q)$ for any σ' that satisfies $(J, \mathit{pre}, \mathit{post})$.

Proving the theorem.

Lemma 4.9 (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 $\Gamma \rightsquigarrow [\mathsf{res} : T, \mathsf{exc} : \mathsf{Thr}]$ we have that (13) implies (14) where

$$\forall \dot{\mu} \in \mathit{XMethEnv} \cdot \dot{\mu} \models ST \;\Rightarrow\; \mathcal{S}[\![\Gamma \vdash E : T]\!](\dot{\mu}) \models \mathit{spec} \tag{13}$$

$$\forall \mu \in \mathit{MethEnv} \cdot \mu \models ST \;\Rightarrow\; \mathcal{D}[\![\Gamma \vdash E : T]\!](\mu) \models \mathit{spec} \tag{14}$$

[5] We are only interested in Γ that has res in its domain since it is present for any expression and command in a method body; cf. the typing rule for method body.

Proof By structural induction on E. For the case $\Gamma \vdash x : T$, the result follows from a much stronger property: $\mathcal{S}[\![\Gamma \vdash x : T]\!](\dot{\mu}) = \mathcal{D}[\![\Gamma \vdash x : T]\!](\mu)$ for all μ and all $\dot{\mu}$. 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

$$\forall \dot{\mu} \in \mathit{XMethEnv} \cdot \dot{\mu} \models ST \ \Rightarrow\ \mathcal{S}[\![\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ E\ \textbf{in}\ E1 : U]\!](\dot{\mu}) \models \mathit{spec} \tag{15}$$

for *spec* of type $\Gamma \rightsquigarrow [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thr}]$. Without loss of generality we can assume *spec* is a simple specification. (For a general one, apply the argument to any $(\mathit{pre}_i, \mathit{post}_i)$.) 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 (15) and satisfiability of ST there is some $\dot{\mu}$ such that

$$\mathcal{S}[\![\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ E\ \textbf{in}\ E1 : U]\!](\dot{\mu}) \models \mathit{spec} \tag{16}$$

Here is the semantics of rule (5), written using $[\![-]\!]$ and μ' to stand for either $\mathcal{D}[\![-]\!]$ and a normal method environment or $\mathcal{S}[\![-]\!]$ and an extended method environment:

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

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

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

Here *rename* just renames res, as in (12). And *alt* is

$$\text{IF ``}\mathsf{exc} = \textbf{null}\text{'' THEN } \mathit{init}\,;\ [\![\Gamma, x : T \vdash E1 : U]\!](\mu') \text{ ELSE } \mathit{propogate}$$

where $\mathit{init} : [\Gamma, \mathsf{res}' : T, \mathsf{exc} : \mathsf{Thr}] \rightsquigarrow [\Gamma, x : T]$ sets x to the value of res' and $\mathit{propogate} : [\Gamma, \mathsf{res}' : T, \mathsf{exc} : \mathsf{Thr}] \rightsquigarrow [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thr}]$ is the (primitive) transformer that sets res to *default* U and exc to the exception (from E). Thus by the decomposition lemmas there are specifications spec_E of type $\Gamma \rightsquigarrow [\mathsf{res} : T, \mathsf{exc} : \mathsf{Thr}]$, spec_{E1} of type $[\Gamma, x : T] \rightsquigarrow [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thr}]$, and $\mathit{spec}_{\mathit{propogate}}$, $\mathit{spec}_{\mathit{init}}$, $\mathit{spec}_{\mathit{rename}}$ of the evident types, such that (16) holds iff each of the component transformers satisfies its specification.

Since assumption (15) 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 $\dot{\mu}$. As a consequence, we may appeal to the induction hypothesis for E, spec_E and for $E1, \mathit{spec}_{E1}$. 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 *rename* are the 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 (9) for methods. Suppose $\mathit{mtype}(m, T) = \overline{z} : \overline{T} \to U$ as in the typing rule for method call. 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 (15).) By decomposition we obtain spec' of type $[\mathsf{self} : T, \overline{z} : \overline{T}] \to U$ 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 (15) 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 (8) 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. 4.2), $ST(T, m) \sqsupseteq^T \mathit{spec}'$ 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 $\mu(C, m) \models^C \mathit{spec}'$. 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 the semantics 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 (13) implies (14) 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 of (13) but not (14).

The following result amounts to the (a) $\Rightarrow$ (b) part of Theorem 4.5.

Lemma 4.10 (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 $\Gamma \rightsquigarrow [\Gamma,\ \mathsf{exc}:\mathsf{Thr}]$ 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 4.9. 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 4.9 for E. □

To prove the (b) $\Rightarrow$ (a) part of Theorem 4.5, (b) can be specialized to method call.

Lemma 4.11 If ST is satisfiable then it has supertype abstraction for method specifications provided that (17) implies (18) for all T and all $m \in \mathit{Meths}\ T$ with $\mathit{mtype}(m,T) = \overline{x} : \overline{T} \rightarrow U$, where

$$\forall \dot{\mu} \in \mathit{XMethEnv} \cdot \dot{\mu} \models ST \ \Rightarrow\ \mathcal{S}[\![\mathsf{self}: T, \overline{y}: \overline{T}, z: U \vdash z := \mathsf{self}.m(\overline{y})]\!](\dot{\mu}) \models \mathit{spec} \qquad (17)$$

$$\forall \mu \in \mathit{MethEnv} \cdot \mu \models ST \ \Rightarrow\ \mathcal{D}[\![\mathsf{self}: T, \overline{y}: \overline{T}, z: U \vdash z := \mathsf{self}.m(\overline{y})]\!](\mu) \models \mathit{spec} \qquad (18)$$

Putting the Theorem 4.5 together with lemma 4.3 we obtain the following.

Corollary 4.12 (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 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 $\mathcal{D}[\![\Gamma \vdash S]\!](\widehat{\mu})$ satisfies spec. Moreover, such reasoning is sound only if ST has behavioral subtyping.

5 Discussion

The main result, Corollary 4.12, 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 or a means to obtain suitable specifications by constructions from arbitrary specifications declared by the programmer.

The most common formulation of behavioral subtyping uses the implications (1) and (2) 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. So Hoare [26] proposed an “adaptation rule”; but his not complete and some subsequent proposals were found to have subtle unsoundness in connection with auxiliary variables in specifications (corrected in [6]). By now, sound and complete adaptation rules are known for some programming languages [16, 30, 53] and the connection with specification refinement has been made clear [21, 45, 56]. Our work seems to be the first to disentangle the characterization of refinement from the use of refinement to define and reason about behavioral subtyping.

In the special case of two-state postconditions (Example 3.2), $(\mathit{pre}', \mathit{post}') \sqsupseteq (\mathit{pre}, \mathit{post})$ holds just when $\mathit{pre} \subseteq \mathit{pre}'$ and $\mathit{old}(\mathit{pre}) \cap \mathit{post}' \subseteq \mathit{post}$; other characterizations can be found in [22] and [21]. More complicated conditions are needed for general specifications, even with auxiliary variables ranging over data types in the programming language rather than arbitrary index sets [16, 30, 56]. Completeness of such a condition depends on the program semantics, since the definition of $\sqsupseteq^T$ quantifies over all program meanings; some early proposals become complete if arbitrary angelic and demonic choice are added to the programming language. An elegant way to study the issues is to interpret specifications as monotonic predicate transformers and study factorization in subcategories of predicate transformers [45].

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* [22, 31, 67, 68].

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 [43]. 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 [11, 37, 43, 50] (see [49] for a survey that includes separation logic). In these and other works [55], invariants are thus folded into method specifications.

Future work. Our theorem is given using a specific programming language and denotational model developed in the context of a project on foundations for the JML specification language including relational reasoning to deal with purity, modifies clauses, etc (e.g. [13, 46]). The semantic model has been encoded in the PVS theorem prover and type soundness proved, building on previous work [47] involving a logical relation for information security. 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 (e.g., [1, 7, 66]).

Our semantics is not compositional at the level of classes: 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 perfectly suited 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 all the interface/class names that appear in its field declarations, casts, and type tests. These may as well be formalized as a closed world consisting of an arbitrary class table for which the method under consideration is well formed. Indeed, since there may be infinitely many interfaces and classes in a class table, we can also define a universal class table by enumerating all possible declarations. Nonetheless it is conceptually appealing to have compositionality at the level of classes. Reus [62] gives a denotational model of that kind, for an untyped language, and considers associated reasoning principles. We refrained from using such a model since it entails additional complexity in proofs and also requires more justification with respect to operational semantics. But it would also be interesting to explore behavioral subtyping and supertype abstraction in such a model and using more abstract semantics of commands as mentioned in Sect. 1.

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, Freidrich Steimann, Joseph Kiniry, Andreas Podelski, and Erik Poll for comments on earlier drafts.

References

[1] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*, pages 69–83, 2006.

[2] W. Ahrendt, Th. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[3] Suad Alagic and Svetlana Kouznetsova. Behavioral compatibility of self-typed theories. In Boris 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.

[4] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean 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.

[5] Pierre 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.

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

[7] Andrew W. Appel, Paul-André Melliès, Chrostopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *POPL*, 2007. to appear.

[8] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In *Intl. Symp. on Theoretical Aspects of Computer Software*, pages 420–447, October 2001. LNCS 2215.

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

[10] Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, November 2005. Extended version of [9].

[11] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram 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.

[12] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian 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.

[13] Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. Reading, writing and relations. In *Programming Languages and Systems (APLAS)*, volume 4279 of *LNCS*, pages 114–130, 2006.

[14] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstra ction. Technical Report ITU-TR-2005-69, IT University of Copenhagen, 2005.

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

[16] A. Bijlsma, P.A. Matthews, and J.G. Wiltink. A sharp proof rule for procedures in wp semantics. *Acta Informatica*, 26:409–419, 1989.

[17] Kim B. Bruce and Peter Wegner. An algebraic model of subtypes in object-oriented languages (draft). *ACM SIGPLAN Notices*, 21(10), October 1986.

[18] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: a developer-oriented approach. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe*, volume 2805 of *LNCS*, pages 422–439. Springer, September 2003.

[19] Criastiano Calcagno, Peter O'Hearn, and Richand Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Computer Science*, 298(3):557–581, 2003.

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

[21] Yonghao Chen and Betty H. C. Cheng. A semantic foundation for specification matching. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 91–109. Cambridge University Press, New York, NY, 2000.

[22] Krishna Kishore Dhara and Gary 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, March 1996.

[23] Robert Bruce Findler and Matthias 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, October 2001.

[24] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM Conf. on Program. Lang. Design and Implementation (PLDI)*, pages 234–245, 2002.

[25] Paul H.B. Gardiner, Clare E. Martin, and Oege de Moor. An algebraic construction of predicate transformers. *Science of Computer Programming*, 22:21–44, 1994.

[26] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*. Springer-Verlag, 1971.

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

[28] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–459, May 2001.

[29] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes (preliminary report). In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *SIGPLAN*, pages 329–340. ACM, October 1998.

[30] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11:541–566, 1999.

[31] Gary T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Zhiming Liu and He Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *LNCS*, pages 2–34, New York, NY, November 2006. Springer-Verlag.

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

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

[34] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[35] Gary Todd Leavens. Verifying object-oriented programs that use subtypes. Technical Report 439, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1989. The author's Ph.D. thesis.

[36] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

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

[38] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, 2002.

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

[40] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

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

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

[43] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, October 2006.

[44] David A. Naumann. A categorical model for higher order imperative programming. *Mathematical Structures in Computer Science*, 8(4):351–399, August 1998.

[45] David A. Naumann. Calculating sharp adaptation rules. *Information Processing Letters*, 77:201–208, 2001.

[46] David A. Naumann. Observational purity and encapsulation. In *Fundamental Aspects of Software Engineering (FASE)*, pages 190–204, 2005.

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

[48] David A. Naumann. From coupling relations to mated invariants for secure information flow. In *European Symposium on Research in Computer Security (ESORICS)*, number 4189 in LNCS, pages 279–296, 2006.

[49] David A. Naumann. On assertion-based encapsulation for object invariants and simulations. *Formal Aspects of Computing*, 2006. Special issue: Applicable Research on Formal Verification and Development, to appear.

[50] David A. Naumann and Mike 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.

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

[52] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *LNCS*, pages 89–105. Springer, 2002.

[53] Ernst-Rüdiger Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Computer Science*, 24:337–347, 1983.

[54] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[55] Matthew J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, November 2005. Dissertation.

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

[57] Cees Pierik and Frank S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 2005. to appear.

[58] Gordon D. Plotkin and John Power. Notions of computation determine monads. In *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 342–356, 2002.

[59] 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.

[60] Erik 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.

[61] John Power. Semantics for local computational effects. In *Mathematical Foundations of Program Semantics*, pages 355–371, 2006.

[62] Bernhard Reus. Modular semantics and logics of classes. In Matthias Baaz and Johann A. Makowsky, editors, *Computer Science Logic (CSL)*, volume 2803 of *LNCS*, pages 456–469, 2003.

[63] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, volume 2, pages 717–740, New York, 1972. ACM.

[64] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

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

[66] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. In *POPL*, pages 63–74, 2005.

[67] Alan Wills. Specification in Fresco. In Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z*, Workshops in Computing, chapter 11, pages 127–135. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[68] Jeannette Marie 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.
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 $f : T$ is in *fields* T and *super* $T = U$ then f is not in *dfields* U.
4. Method types (including parameter names, for technical convenience) 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 $\Gamma = [\mathsf{self} : C, \mathsf{res} : T, \overline{x} : \overline{T}]$. Also exc does not occur in S. Rules that define $\Gamma \vdash S$ appear in Fig. 2 and Fig. 3.
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.

$$\Gamma \vdash 0 : \textbf{int} \qquad \Gamma \vdash x : \Gamma x \qquad \Gamma \vdash \textbf{new}\ C() \qquad \frac{T \in \mathit{RefType}}{\Gamma \vdash \textbf{null} : T}$$

$$\frac{\Gamma \vdash E : T \qquad [\Gamma,\ x : T] \vdash E1 : U}{\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ E\ \textbf{in}\ E1 : U} \qquad \frac{\Gamma \vdash x : T_1 \qquad \Gamma \vdash y : T_2}{\Gamma \vdash x = y : \textbf{bool}}$$

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

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

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

Figure 2: Typing rules for expressions.

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

$$\frac{\Gamma \vdash x : U \qquad (f : T) \in \mathit{dfields}\ V \qquad U \leq V \qquad \Gamma \vdash y : T1 \qquad T1 \leq T}{\Gamma \vdash x.f := y}$$

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

$$\frac{\Gamma \vdash x : T \qquad T \leq \mathsf{Thr}}{\Gamma \vdash \textbf{throw}\ x} \qquad \frac{\Gamma \vdash S_1 \qquad [\Gamma,\ x : T] \vdash S_2 \qquad T \leq \mathsf{Thr}}{\Gamma \vdash \textbf{try}\ S_1\ \textbf{catch}(x : T)\ S_2} \qquad p\frac{\Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash \textbf{try}\ S_1\ \textbf{finally}\ S_2}$$

Figure 3: Typing rules for commands.

B Semantics

The semantics of commands and expressions are defined in figures 4 and 5. The semantics is defined with respect to an arbitrary allocator. An *allocator* is just a choice function for unused references, i.e., a function *fresh* that maps a pair (ρ, h), with $h \in \mathit{Heap}(\rho)$, to a reference such that $\mathit{fresh}(\rho, h) \notin \mathit{dom}\,\rho$.[6]

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 $C \leq \mathsf{Thr}$, and any type T we define $\mathit{except}(\rho, h, T, C)$ to be an element of $\mathit{State}([\mathsf{res} : T, \mathsf{exc} : \mathsf{Thr}])$ as follows.

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

This is similar to the semantics of **new** $C()$, but the new object is assigned to exc rather than to res. A similar function is used in the semantics of commands. Given (ρ, h, r) in $\mathit{State}(\Gamma)$ and classname $C \leq \mathsf{Thr}$ we define $\mathit{except}(\rho, h, r, C)$ to be an element of $\mathit{State}([\Gamma,\ \mathsf{exc} : \mathsf{Thr}])$ as follows.

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

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

$$
\begin{array}{l}
[\![\Gamma \vdash x : T]\!](\mu)(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : r\,x, \mathsf{exc} : \mathit{null}] \\
[\![\Gamma \vdash \mathbf{true} : \mathbf{bool}]\!](\mu)(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : \mathit{true}, \mathsf{exc} : \mathit{null}]) \\
[\![\Gamma \vdash 0 : \mathbf{int}]\!](\mu)(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : 0, \mathsf{exc} : \mathit{null}]) \\
[\![\Gamma \vdash \mathbf{null} : T]\!](\mu)(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : \mathit{null}, \mathsf{exc} : \mathit{null}]) \\
[\![\Gamma \vdash x = y : \mathbf{bool}]\!](\mu)(\rho, h, r) \;= \\
\quad \mathsf{let}\ v = (\mathsf{if}\ (r\,x = r\,y)\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ \mathit{false})\ \mathsf{in}\ (\rho, h, [\mathsf{res} : v, \mathsf{exc} : \mathit{null}]) \\
[\![\Gamma \vdash \mathbf{new}\ C() : C]\!](\mu)(\rho, h, r) \;= \\
\quad \mathsf{let}\ o = \mathit{fresh}(\rho, h)\ \mathsf{in}\ \mathsf{let}\ \rho_o = [\rho,\ o : C]\ \mathsf{in}\ \mathsf{let}\ h_o = [h,\ o : \mathit{defaultObrecord}\ C]\ \mathsf{in} \\
\quad (\rho_0, h_0, [\mathsf{res} : o, \mathsf{exc} : \mathit{null}]) \\
[\![\Gamma \vdash x.f : T]\!](\mu)(\rho, h, r) \;= \\
\quad \mathsf{if}\ r\,x \neq \mathit{null}\ \mathsf{then}\ (\rho, h, [\mathsf{res} : h(r\,x).f, \mathsf{exc} : \mathit{null}])\ \mathsf{else}\ \mathit{except}(\rho, h, T, \mathsf{NullDeref}) \\
[\![\Gamma \vdash (U)\ x : U]\!](\mu)(\rho, h, r) \;= \\
\quad \mathsf{if}\ r\,x = \mathit{null} \lor \rho(r\,x) \leq U\ \mathsf{then}\ (\rho, h, [\mathsf{res} : r\,x, \mathsf{exc} : \mathit{null}])\ \mathsf{else}\ \mathit{except}(\rho, h, U, \mathsf{ClassCast}) \\
[\![\Gamma \vdash x\ \mathbf{is}\ U : \mathbf{bool}]\!](\mu)(\rho, h, r) \;= \\
\quad \mathsf{let}\ v = \mathsf{if}\ r\,x \neq \mathit{null} \land \rho(r\,x) \leq U\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ \mathit{false}\ \mathsf{in}\ (\rho, h, [\mathsf{res} : v, \mathsf{exc} : \mathit{null}])
\end{array}
$$

Figure 4: Semantics of expressions other than method call and **let**. Read $[\![-]\!]$ as either $\mathcal{D}[\![-]\!]$ or $\mathcal{S}[\![-]\!]$ throughout.

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.

A class table denotes a method environment obtained as a fixpoint. The first step is to define semantics of method declaration $mdec$ of the form $m(\overline{x} : \overline{T}) : T\ \{\ S\ \}$ in some class C. We define $[\![mdec]\!]$ to be a function from method environments to $SemMeth(C, m)$, i.e.,

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

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 (ρ, h, r) in $State([\mathsf{self} : C, \overline{x} : \overline{T}])$, define

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

Now define an ascending chain $\mu \in \mathbb{N} \to MethEnv$ of method environments as follows.

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

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 $C \leq B$ implies $[\![C]\!] \subseteq [\![B]\!]$ which in turn induces an inclusion for stores.[7]

Method environments are ordered by $\mu \leq \mu'$ iff $\mu(C, m) \leq \mu'(C, m)$ for all C, m. This refers to the usual ordering on state transformers: For σ and τ in $STrans(\Gamma, \Gamma')$, define $\sigma \leq \tau$ 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. We refrain from proving that for any $\Gamma \vdash S$, the semantics $[\![\Gamma \vdash S]\!]$ is a continuous function from method environments to state transformers. Similarly, the semantics of a method declaration is continuous in the method environment. It follows that $i \leq j \Rightarrow \mu_i \leq \mu_j$ for the approximation chain. The semantics $\widehat{\mu}$ is defined to be the least upper bound of the approximation chain. (A similar semantics is used in [10]; a characterization of least upper bounds is very easy and machine-checked proofs of the continuity properties etc. appear in [47].)

[7]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 $T \geq C$.

$[\![\Gamma \vdash x.f := y]\!](\mu)(\rho, h, r) \;=\; \text{if } r\,x \neq \mathit{null} \text{ then } (\rho, [h \mid r\,x.f : r\,y], r) \text{ else } \mathit{except}(\rho, h, r, \mathsf{NullDeref})$
$[\![\Gamma \vdash \textbf{if } x \textbf{ then } S_1 \textbf{ else } S_2]\!](\mu)(\rho, h, r) \;=$
  $\text{if } r\,x = \mathit{true} \text{ then } [\![\Gamma \vdash S_1]\!](\mu)(\rho, h, r) \text{ else } [\![\Gamma \vdash S_2]\!](\mu)(\rho, h, r)$
$[\![\Gamma \vdash \textbf{var } x : T \textbf{ in } S]\!](\mu)(\rho, h, r) \;=$
  $\text{lets } (\rho_1, h_1, r_1) = [\![\Gamma, x : T \vdash S]\!](\mu)(\rho, h, [r,\; x : \mathit{default}\ T]) \text{ in } (\rho_1, h_1, r_1 - x)$
$[\![\Gamma \vdash S_1;\; S_2]\!](\mu)(\rho, h, r) \;=$
  $\text{lets } (\rho_1, h_1, r_1) = [\![\Gamma \vdash S_1]\!](\mu)(\rho, h, r) \text{ in}$
  $\text{if } r_1\,\mathsf{exc} = \mathit{null} \text{ then } [\![\Gamma \vdash S_2]\!](\mu)(\rho_1, h_1, r_1 - \mathsf{exc}) \text{ else } (\rho_1, h_1, r_1)$
$[\![\Gamma \vdash \textbf{throw } x]\!](\mu)(\rho, h, r) \;=\; \text{if } r\,x \neq \mathit{null} \text{ then } (\rho, h, [r,\; \mathsf{exc} : r\,x]) \text{ else } \mathit{except}(\rho, h, r, \mathsf{NullDeref})$
$[\![\Gamma \vdash \textbf{try } S_1 \textbf{ catch}(x : T)\; S_2]\!](\mu)(\rho, h, r) \;=$
  $\text{lets } (\rho_1, h_1, r_1) = [\![\Gamma \vdash S_1]\!](\mu)(\rho, h, r) \text{ in}$
  $\text{if } r_1\,\mathsf{exc} = \mathit{null} \vee \rho(r_1\,\mathsf{exc}) \not\leq T \text{ then } (\rho_1, h_1, r_1)$
  $\text{else let } r_3 = [r_1 \mid x : r_1\,\mathsf{res}] - \mathsf{exc} \text{ in}$
    $\text{lets } (\rho_2, h_2, r_2) = [\![\Gamma, x : T \vdash S_2]\!](\mu)(\rho_1, h_1, r_3) \text{ in } (\rho_2, h_2, r_2 - x)$
$[\![\Gamma \vdash \textbf{try } S_1 \textbf{ finally } S_2]\!](\mu)(\rho, h, r) \;=$
  $\text{lets } (\rho_1, h_1, r_1) = [\![\Gamma \vdash S_1]\!](\mu)(\rho, h, r) \text{ in}$
  $\text{lets } (\rho_2, h_2, r_2) = [\![\Gamma \vdash S_2]\!](\mu)(\rho_1, h_1, r_1 - \mathsf{exc}) \text{ in}$
  $\text{if } r_2\,\mathsf{exc} = \mathit{null} \text{ then } (\rho_2, h_2, [r_2,\; \mathsf{exc} : r_1\,\mathsf{exc}]) \text{ else } (\rho_2, h_2, r_2)$

Figure 5: Semantics of commands. Read $[\![-]\!]$ as either $\mathcal{D}[\![-]\!]$ or $\mathcal{S}[\![-]\!]$ throughout.

C Additional proofs

Proof of Lemma 3.7 **Proof** To show $\mathit{spec}_2 \sqsupseteq^U \mathit{spec}_0$, assume $\sigma \in \mathit{STrans}([\Gamma \mid \mathsf{self} : U], \Gamma')$ and $\sigma \models \mathit{spec}_2$ with the aim to prove $\sigma \models\!\!\models^U \mathit{spec}_0$. From $\mathit{spec}_2 \sqsupseteq^U \mathit{spec}_1$ we get $\sigma \models\!\!\models^U \mathit{spec}_1$; but this does not yield $\sigma \models \mathit{spec}_1$ which isn't even defined. Define $\tau \in \mathit{STrans}([\Gamma \mid \mathsf{self} : T], \Gamma')$ by

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

where s is an arbitrarily chosen state that satisfies spec_1 for (ρ, h, r). From $\sigma \models\!\!\models^U \mathit{spec}_1$ we get $\tau \models \mathit{spec}_1$. Then by $\mathit{spec}_1 \sqsupseteq^T \mathit{spec}_0$ we get that $\tau \models\!\!\models^T \mathit{spec}_0$. Now $\sigma \models\!\!\models^U \mathit{spec}_0$ follows from $\tau \models\!\!\models^T \mathit{spec}_0$ by definition of τ and $\models\!\!\models$. ☐

Proof of Lemma 4.3 **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 $ST(T, m) \sqsupseteq \mathit{spec}$ 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 3.7) to get $ST(C, m) \sqsupseteq^C \mathit{spec}$.

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 $\mathit{spec} := ST(U, m)$. Since $ST(U, m) \sqsupseteq^T ST(U, m)$, supertype abstraction yields $ST(C, m) \sqsupseteq^C ST(U, m)$. ☐

Proof of Lemma 4.8. **Proof** Observe that, by definitions, $\sigma \models (J, \mathit{pre}, \mathit{post})$ is equivalent to

$$\forall r \in \mathit{Store}(\Gamma) \cdot \sigma \models (\mathit{pre}_r, \mathit{post}_r)$$

which is equivalent, by definition of satisfaction, to

$$\forall r \in \mathit{Store}(\Gamma) \cdot \forall \rho, h, q \cdot (\rho, h, q) \in \mathit{pre}_r \;\Rightarrow\; \sigma(\rho, h, q) \in \mathit{post}_r$$

By definition of pre and post this is equivalent to

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

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 $\langle \sigma \mid \mathit{id} \rangle \models (P, Q)$. We leave to the reader the argument why $\langle \sigma' \mid \mathit{id} \rangle \models (P, Q)$ for any σ' that satisfies $(J, \mathit{pre}, \mathit{post})$. ☐

Proof of Lemma 4.11. **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 $C \leq T$. 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 4.9, 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 (17) 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 (18) with $spec := ST(T, m)$, which, for any $C \leq T$, boils down to $\forall \sigma \in SemMeth(C, m) \cdot \sigma \models ST(C, m) \Rightarrow \sigma \models^C ST(T, m)$ whence $ST(C, m) \sqsupseteq^C ST(T, m)$. $\square$