A

# Behavioral Subtyping, Specification Inheritance, and Modular Reasoning

GARY T. LEAVENS, University of Central Florida
and DAVID A. NAUMANN, Stevens Institute of Technology

Verification of a dynamically-dispatched method call, $E.m()$, seems to depend on $E$'s dynamic type. To avoid case analysis and allow incremental development, object-oriented program verification uses supertype abstraction. That is, one reasons about $E.m()$ using $m$'s specification for $E$'s static type. Supertype abstraction is valid when each subtype in the program is a behavioral subtype. This paper formalizes supertype abstraction and behavioral subtyping for a Java-like sequential language with mutation, and proves that behavioral subtyping is both necessary and sufficient for the validity of supertype abstraction. Specification inheritance, as in JML, is also formalized and proved to entail behavioral subtyping.

Categories and Subject Descriptors: 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**]: 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, logics of programs, pre- and post-conditions, specification techniques.

General Terms: Specification, Verification

Additional Key Words and Phrases: Behavioral subtyping, supertype abstraction, specification inheritance, modularity, specification, verification, refinement, state transformer, predicate transformer, dynamic dispatch, Eiffel language, JML language.

## 1. INTRODUCTION

In object-oriented (OO) programming, subtyping and dynamic dispatch are both useful and problematic. They are useful because supertypes can abstract away details in the specifications of their subtypes, thus allowing variations in data structures and algorithms to be handled uniformly. They are problematic for modular reasoning because a dynamically-dispatched method call $E.m()$ seems to require a case analysis to deal with all possible dynamic types of $E$'s value. The basic idea of the modular reasoning technique called "supertype abstraction" [Leavens and Weihl 1995] is to use only the specification of $m$ from $E$'s static type to reason about such calls. Reasoning with supertype abstraction is valid when the program's types follow behavioral subtyping.

Behavioral subtyping is a generalization of type checking, since it imposes constraints on the behavior of implementations of $m$ at all subtypes of $E$'s static type. At each subtype, the specification lets $m$ make the same or weaker assumptions, while requiring the same or stronger guarantees. Modular type safety conditions for dynamically-dispatched methods are well-known [Cardelli 1988]. The most well-known definition of behavioral subtyping is a straightforward translation of those co- and contra-variant subtyping conditions into implications between pre- and post-conditions [Liskov and Wing 1994]. While sound, these implications are incomplete, and the translation does not account for object invariants. Better definitions of behavioral subtyping are known and variations are em-

Leavens was supported in part by NSF and CNS 08-08913, and performed some of this work while employed at Iowa State University.
Naumann was supported in part by NSF grants CCR-0208984, CCF-0429894, CNS-0627338, CNS-0708330, and CCF-0915611.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

bodied in some reasoning systems, though either by postulation or with indirect justification. What is lacking is principled criteria to evaluate alternate definitions.

In order to construct specifications that have behavioral subtyping, notions of specification inheritance have been studied and implemented, notably in JML [Leavens et al. 2006]. Again what is lacking is principled criteria to evaluate forms of specification inheritance with respect to the informal goals of soundness and completeness for reasoning and minimal restrictiveness on method implementations.

Our goal is to investigate the soundness and completeness of behavioral subtyping in a way that is independent of particular proof systems, and which connects with operationally sound program semantics as opposed to axiomatic semantics. To this end we formalize supertype abstraction, behavioral subtyping, and specification inheritance semantically, for a Java-like sequential language. Indeed, we formalize two forms of behavioral subtyping. We prove that one form is sound and complete for reasoning with supertype abstraction. The stronger form of behavioral subtyping arises from specification inheritance.

### 1.1. Preview of related work

Remarkably, the literature provides 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 of the topic [Alagic and Kouznetsova 2002; America 1987; America 1991; Bruce and Wegner 1986; Dhara and Leavens 1996; Findler and Felleisen 2001; Findler et al. 2001; Leavens and Weihl 1995; Liskov and Wing 1994; Meyer 1997; Poll 2000] (see [Leavens and Dhara 2000] for a survey, and Sect. 3). Some of the current understanding of behavioral subtyping and supertype abstraction is embodied in program logics [Müller 2002; Oheimb and Nipkow 2002; Parkinson 2005; Pierik 2006; Poetzsch-Heffter and Müller 1999] but is difficult to disentangle from various other complications such as specialized specification languages. Some of the current understanding is embodied in languages and tools such as Eiffel [Meyer 1997; ECMA International 2006], JML [Leavens et al. 2006], ESC/Java [Flanagan et al. 2002], Spec# [Barnett et al. 2005; Barnett et al. 2004], and KeY [Beckert et al. 2007]. But these have unsoundnesses and incompletenesses, some by engineering design and some for lack of adequate theory and methodology. A key source of unsoundness is a naive treatment of object invariants, because a lack of encapsulation of an object's representation can invalidate the simple hierarchical notion of encapsulation on which the standard treatment of invariants [Hoare 1972; Liskov and Wing 1994] is based.

On one hand, behavioral subtyping has been rigorously studied in various applicative and abstract models [Leavens and Weihl 1995; Liskov and Wing 1994; Poll 2000]. On the other hand, for conventional object oriented programs and contracts, various embodiments have been implemented in static and runtime verification tools and logics that apply to rich specification and programming languages such as those mentioned above. The latter works cite Leavens and Weihl [1995], Liskov and Wing [1994], and Poll [2000] as standard background for behavioral subtyping but in terms of technical content there is almost no connection. We aim to close the gap by providing a rigorous analysis for a realistic Java-like language and ordinary functional specifications, on which can be based more specialized assessments and justifications of specific tools and logics.

The achievements closest to our aim are soundness and completeness results for logics of Java fragments that embody supertype abstraction in some form. Pierik [2006] and Parkinson [2005] prove soundness of logics with supertype abstraction and requiring behavioral subtyping. But these results assess the reasoning power of an entire proof system, rather than explicating the connection between behavioral subtyping and supertype abstraction *per se*. Soundness (and even completeness) of a logic that includes supertype abstraction does not imply our result, which directly connects behavioral subtyping with supertype abstraction. Their results are also somewhat removed from the axiomatic semantics of some widely used verifiers, which simply postulate soundness of behavioral subtyping by baking supertype abstraction into the axiomatic semantics.

### 1.2. Approach

The key theoretical novelty that leads to our results is a purely semantic formulation of supertype abstraction using two denotational semantics. In one semantics, method calls are statically dispatched; this is used to model reasoning with supertype abstraction. The other semantics is standard and matches the runtime execution of OO languages with dynamic dispatch. Our results also involve

predicate transformer semantics derived from each denotational semantics. A critical element is the intrinsic refinement relation on specifications, defined in terms of satisfying implementations rather than implications between pre- and post-conditions.

We give a formal treatment of many constructs of sequential OO languages, including: classes, interfaces, mutable heap objects, assignment, exceptions, inheritance, reference equality, type casts, type tests, and recursive types. A minimal language for our purposes would contain just classes, interfaces, method calls, mutable objects, and sequential control structures. We have included a few other features that are useful in practice and illustrate the generality of our technical results. Of particular interest are shared references via expressions with differing static types, and runtime type tests and type casts which allow programs to make observations that can distinguish between supertype and subtype objects. We had thought that unrestricted use of such features might invalidate supertype abstraction. However, our work shows that supertype abstraction is valid even in a realistic Java-like language that includes these features without restriction.

Our language does not model concurrency, nested classes, (static) method overloading, reflection, or dynamic class loading. We omit concurrency because it involves a richer notion of behavior[1] and reflection because we know of no systems for specifying much less verifying such programs. The rest involve complications that shed little light on our main topic.

Our results show that the connection between supertype abstraction and behavioral subtyping is tightly linked with method calls. The use of supertype abstraction in the context of other program constructs hinges on those constructs being compositional with respect to correctness, and monotonic with respect to refinement. To make this entirely explicit, we would need to abstract over the particular constructs of the programming language. Given that our main goal is to explicate modular reasoning about conventional OO programs, we choose instead to work with a specific programming language, while pointing out the way in which the results could be generalized.

Our definitions make no commitment to particular specification notations or reasoning systems. Instead, we formulate modular reasoning and supertype abstraction semantically, in a generic way that idealizes what is found in logics and tools. Using an operationally-sound, compositional semantics allows us to provide a foundation that will serve as a point of reference and as a basis for assessment and further development of specification languages and verification tools.

While we motivate many of our examples using Java and JML, the theory we present in this paper is not specific to either language. We mention Java and JML in some examples, because they are well documented and widely understood, and because JML's design matches this paper's theory.

### 1.3. Contributions

This paper makes the following contributions.

— We give a semantic characterization of *supertype abstraction*, which idealizes what is found in logics and verification tools. In contrast to related work, our definition does not rely on derived notions such as substituting one object for another [Leavens 1989; Liskov 1988; Liskov and Wing 1994] that lack a clear connection with ordinary functional specifications and imperative program behavior. We focus on total correctness, while spelling out the similar results hold for partial correctness, because total correctness brings to light the need for satisfiability of specifications in our main results.
— Our characterization of supertype abstraction is built on a notion of *modular correctness* defined using predicate transformer semantics. Modular correctness is essentially the property that is checked by modular static verifiers like ESC/Java and KeY, but surprisingly the property is seldom explicit in the literature on verification theory.
— We formalize *behavioral subtyping* in terms of refinement of functional behavior in a realistic programming model. Surprisingly, refinement does not need to hold between all type/subtype pairs, as in prior work, but only when the subtype is an instantiable class (as opposed to an interface or abstract class, in Java terminology). We also highlight two distinct notions of refinement at a subtype.
— In contrast to standard proof-theoretic definitions [Liskov and Wing 1994], we define refinement (of specifications) intrinsically, in terms of satisfying implementations. Separately, we *characterize*

[1]The addition of bounded nondeterministic choice, however, would be straightforward. All of our main results would still hold, but the semantic definitions would be slightly more complicated.

*refinement* in terms of pre- and postconditions. Our characterization adapts previous work [Chen and Cheng 2000; Naumann 2001] that improves on the overly restrictive standard condition of postcondition implications that are still often used [America 1987; America 1991; Liskov and Wing 1994; Meyer 1997; Findler and Felleisen 2001; Findler et al. 2001] and it justifies the less restrictive conditions used in some works ([Beckert et al. 2007, Section 8.1.3] [Greenberg et al. 2010] [Liskov and Guttag 2001, Section 7.9.1] [ECMA International 2006, Section 8.10.5]) for specifications using two-state postconditions. Our characterization isolates the ways in which completeness depends on the specification language. One surprise is that the usual precondition rule is unnecessarily strong for partial correctness specifications. An outcome of our focus on reasoning about correctness of programs, rather than an abstract model of computation, is that we find abstraction functions are not an integral part of behavioral subtyping (compare, e.g., Leavens and Weihl [1995], Liskov and Wing [1994]).

— We prove *soundness of behavioral subtyping for supertype abstraction*. Even stronger, our main result is a form of *semantic completeness*: that behavioral subtyping holds for a collection of class and interface specifications if and only if supertype abstraction is valid. Soundness has been proved before, as mentioned at the end of Sect. 1.1, though under more restrictive conditions and entangled with particulars of program logics. We know of no prior completeness result.

— We formalize *specification inheritance* [Wills 1992] and show that it ensures behavioral subtyping. Specification inheritance is part of the semantic definition of JML [Leavens 2006] and it embodies the proof obligations whereby some logics achieve behavioral subtyping [Müller 2002; Oheimb and Nipkow 2002; Parkinson 2005; Pierik 2006; Poetzsch-Heffter and Müller 1999; Beckert et al. 2007]. We identify several desiderata for specification inheritance and several variations on specification inheritance. The variation that appears most useful ensures a form of behavioral subtyping that is stronger than the one that is equivalent to supertype abstraction. The two forms of behavioral subtyping derive from the two notions of refinement at a subtype. In fact there are four forms. The other dimension is whether refinement is required at interface subtypes: though it is not necessary for supertype abstraction, it does yield stronger reasoning.

### 1.4. Organization and conventions

The next section summarizes the key ideas, followed by related work in Sect. 3. Next is the programming language syntax and semantics (Sect. 4) followed by the semantic formalization of specifications (Sect. 5), including the characterization of refinement in terms of pre/post conditions. The short Sect. 6 formalizes modular reasoning from specifications, making forward reference to the rather long Sect. 7 that derives the weakest-precondition predicate transformer semantics. Sect. 8 gives the central definitions of the paper —formalization of supertype abstraction and behavioral subtyping— together with the main theorem which says they are equivalent. Sect. 9 investigates specification inheritance and shows how it ensures behavioral subtyping. Sect. 10 shows how the results carry over to partial correctness. Sect. 11 concludes. Some technical details and proofs are in the appendix. *In this version, an index and table of contents are provided at the end.*

We include considerable detail in proofs, to make it easier to read in depth and also to expose assumptions and techniques so the results can be soundly adapted to particular programming and specification languages. To compensate for the length, we include guideposts for more casual reading.

## 2. SYNOPSIS

In an OO language it is not obvious how to do modular reasoning, because dynamic dispatch selects different methods depending on the exact run time class of an object. What specification should one use to reason about a call $E.m()$, given that the static type of expression $E$, say $T$, is only an upper bound on its dynamic type? While we consider formal specifications and reasoning in this paper, the problem also applies to informal reasoning based on informal specifications. The first expert OO programmers used $T$'s specification of $m$ to reason about such calls. This kind of reasoning, *supertype abstraction* [Leavens and Weihl 1995], is modular in that it does not depend on $E$'s dynamic type, and hence does not have to be changed when subtypes of $T$ are changed in compatible ways or are added to a program [Liskov 1988]. Supertype abstraction supports maintenance and evolutionary programming styles.

However, supertype abstraction is only valid if methods that override $T$'s method $m$ satisfy $T$'s specification for $m$, since that specification is the one used in reasoning about such calls. Making

overrides obey the specification of overridden methods, *specification inheritance* [Dhara and Leavens 1996; Wills 1992], ensures that objects of subtypes of $T$ do not cause surprising behavior when treated as if they are objects of type $T$; that is, it ensures *behavioral subtyping* [America 1987; America 1991; Meyer 1985]. An alternative is to check the given specifications and treat violations of behavioral subtyping as a design error [Findler and Felleisen 2001; Findler et al. 2001]. Either way, careful analysis is needed for soundness and to avoid unnecessary restriction.

### 2.1. Supertype abstraction

An influential discussion of the benefits of these ideas is Liskov's keynote at OOPSLA 1987 [Liskov 1988]. Liskov stated an easily-remembered test for behavioral[2] 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 as originally stated 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.

As a formulation of supertype abstraction, the LSP is not easy to apply to imperative OO languages. It is not clear what it means to substitute one object for another: imperative programs are not referentially transparent, object identity matters, and the state of "an object" often depends on other objects in the heap. (Object identity is also a problem with the algebraic work on applicative languages from which the LSP is drawn [Bruce and Wegner 1986; Leavens 1989].) A more flexible intuition defines observations that are not allowed by the supertype's specification as "surprising," and says that behavioral subtyping prevents surprising behavior.[3] One of the contributions of our paper is to precisely formalize supertype abstraction in a way that captures modular reasoning about imperative OO programs.

In a program logic, supertype abstraction can be embodied by the proof rule for method invocation, which allows deriving the correctness judgment

$$\{pre_m^T[E/\mathsf{self}]\}\ E.m()\ \{post_m^T[E/\mathsf{self}]\} \tag{1}$$

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** $pre_m^T$; **havoc**; **assume** $post_m^T$" (with various optimizations [Leino 2005]). Both techniques aim to produce sound conclusions about the actual semantics of the call, independent of how $m$ is implemented.

Our formalization of supertype abstraction says that properties of an arbitrary 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 formalization is in terms of a denotational semantics that uses static dispatch, written $\mathcal{S}[\![-]\!]$, and a static-dispatch predicate transformer semantics built from it.

To investigate the soundness and completeness of such reasoning, with respect to observable program behavior, we also define a standard dynamic-dispatch semantics. For an arbitrary command $C$, its dynamic-dispatch meaning, $\mathcal{D}[\![C]\!]$, is interpreted in a ***method environment***, $\eta$, that gives a meaning to each method at each type. Thus $\mathcal{D}[\![C]\!](\eta)$ is a function from initial states to final states. The semantics of a program's class declarations, i.e., of its class table, $CT$, is given by a method environment $\mathcal{D}[\![CT]\!]$. Sometimes we abbreviate $\mathcal{D}[\![CT]\!]$ as $\widehat{\eta}$.

We are interested in proving that $C$ satisfies some pre/post specification, *spec*. Modular reasoning proves that $\mathcal{D}[\![C]\!](\widehat{\eta})$ satisfies *spec* by using only method specifications to reason about method calls. In our formalization, such method specifications are kept in a ***specification table***, $ST$, which maps pairs of type and method names to the corresponding method specifications. The specification table entry $ST(T, m)$ models the proof obligations that are imposed on implementation of method $m$ in type $T$. To model that only specifications are used, we use the static dispatch semantics $\mathcal{S}[\![C]\!]$ with an arbitrary environment $\eta$ that satisfies $ST$. Supertype abstraction amounts to reasoning about $\mathcal{S}[\![C]\!](\eta)$ in order to establish properties of $\mathcal{D}[\![C]\!](\widehat{\eta})$.

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

[3] "So programs can manipulate instances of a subtype as if they were instances of that type's supertypes without surprising results." [Leavens 1989, Chapter 9].

What makes supertype abstraction sound is behavioral subtyping. Behavioral subtyping imposes obligations on implementations of $m$ in subtypes of $T$. The obligations can be defined in terms of specifications, and amount to a notion of refinement.

To see how refinement of specifications in the specification table is needed for the soundness of supertype abstraction, consider the following form of reasoning, based on a particular method call rule, which we provisionally call *supertype abstraction for invocations*:

> Suppose that for each type $T$ and method $m$ defined on $T$ objects, its implementation, satisfies $ST(T, m)$, where $ST(T, m) = (pre_m^T, post_m^T)$. Then for all $E : T$, the Hoare triple (1) above holds.

If a subtype, say $K \leq T$, has an implementation of $m$, then that implementation is defined on objects that are of type $K$ and its subtypes, and if the value of $E$ is exactly a $K$ object, then the call $E.m()$ will dispatch to the implementation for class $K$, with denotation $\mathcal{D}[\![CT]\!](K, m)$. The point of behavioral subtyping is to reconcile the potential mismatch between the proof obligation for $\mathcal{D}[\![CT]\!](K, m)$, which is $ST(K, m)$, and the claim (1) based on $ST(T, m)$. The idea is that $ST(K, m)$ may be different than $ST(T, m)$, but it should ensure that every implementation of $m$ at $K$ that satisfies $ST(K, m)$ also satisfies $ST(T, m)$. That is, $ST(K, m)$ refines $ST(T, m)$ in the intrinsic sense: *spec*′ refines *spec* iff each program satisfying *spec*′ also satisfies *spec*.

Instantiating a specification, as in (1) above, is the most basic way to reason about a method call, but other conclusions also follow from the method's specification, as may be obtained by proof rules like Consequence and Invariance [Apt et al. 2009]. Moreover, we are interested in the use of supertype abstraction in proving judgments $\{pre\}C\{post\}$ for arbitrary commands and specifications. We want to prove the soundness of behavioral subtyping for supertype abstraction independently from the way in which $\{pre\}C\{post\}$ is proved, and from the language in which *pre* and *post* are expressed. So we formalize supertype abstraction in purely semantic terms as a semantic consequence relation. The payoff is a clear notion of completeness, different from the completeness of a reasoning system.

Our definition of supertype abstraction considers semantic consequences that hold for each method environment $\eta$ that satisfies the specification table. Whereas the actual program semantics, $\mathcal{D}[\![-]\!]$, uses dynamic dispatch, we formalize supertype abstraction using the static dispatch semantics, $\mathcal{S}[\![-]\!]$. Soundness of supertype abstraction means, roughly: If $\mathcal{S}[\![C]\!](\eta)$ satisfies *spec* for each $\eta$ that satisfies the specification table, and if the program's method environment $\widehat{\eta}$ satisfies the specification table, then the actual semantics $\mathcal{D}[\![C]\!](\widehat{\eta})$ satisfies *spec*. This quantification over all $\eta$ can be avoided by using a single "least refined implementation", which amounts to working at the level of axiomatic semantics (predicate transformers). This is how many verification tools work.

The main result of the paper, Theorem 8.15, says that specifications have behavioral subtyping if and only if they allow supertype abstraction. Furthermore, the two notions of supertype abstraction —using predicate transformers or quantifying over environments— are equivalent, even though one is strictly stronger when applied to a single specification and program. The soundness proof relies on connections between the static- and dynamic-dispatch semantics that are proved by induction over the syntax of the language, and are thus language-dependent, but what matters is that the language constructs are refinement-monotonic. The completeness proof relies on the above-mentioned quantification over satisfying implementations. If supertype abstraction holds for $m$ at types $T$ and $K \leq T$, every implementation that satisfies $ST(K, m)$ must satisfy $ST(T, m)$ —and that is the intrinsic refinement relation between $ST(K, m)$ and $ST(T, m)$ required by our definition of behavioral subtyping.

### 2.2. Defining behavioral subtyping

Several authors have offered definitions of behavioral subtyping, but the most influential definition has been Liskov and Wing's [Liskov and Wing 1994]. We paraphrase part of their "constraint rule" (from their Figure 4, page 1823).[4] For type $S$ to be a behavioral subtype of $T$:

[4]Their rule allows for method renamings, found in Eiffel but few other OO languages, and an abstraction function. Our OO language semantics does not model the former. We do not need abstraction functions because we connect behavioral subtyping to the observable properties of programs. We also ignore the part of the rule about history constraints, since we study a sequential language.

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

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

Fig. 1. The interface Tracker and the class WeightLoss, specified using notations from JML. (Infix operators bind more tightly than ∧ and ∨, which in turn bind more tightly than ⇒.)

— The subtype's invariant must imply the supertype's. That is, whenever $S$'s invariant holds for a subtype object[5], then $T$'s invariant holds:

$$inv^S(\mathsf{self}) \;\Rightarrow\; inv^T(\mathsf{self}), \quad \text{for all } \mathsf{self} : S. \tag{2}$$

— "Subtype methods preserve the supertype method's behavior." That is, if $S$'s method $m$ overrides $T$'s method $m$, then the usual static typing conditions [Cardelli 1988] hold, and for all subtype objects $\mathsf{self} : S$, $T$'s precondition for $m$ implies $S$'s precondition:

$$pre_m^T(\mathsf{self}) \;\Rightarrow\; pre_m^S(\mathsf{self}) \tag{3}$$

and $S$'s postcondition implies $T$'s:

$$post_m^S(\mathsf{self}) \;\Rightarrow\; post_m^T(\mathsf{self}). \tag{4}$$

Liskov and Wing's definition is intended to be part of a "descriptive and informal" presentation (p. 1813), which concentrates on ideas and has only "informal justifications" (p. 1813). However, even at a conceptual level, it can be improved.

Liskov and Wing's postcondition rule (4), though attractively simple and memorable, is stronger (i.e., less flexible) than necessary for the soundness of supertype abstraction [Dhara and Leavens 1996]. To see this, consider Fig. 1. The interface Tracker declares two 'model fields', goal and curr, that are only used in specifications [Cheon et al. 2005; Leino 1995; Leino and Müller 2006]. They stand for the goal of a diet and the current weight. The values of these model fields are given by the represents clauses in class WeightLoss. The clause "**represents** goal := self.g" declares part of the object invariant for WeightLoss, which says that goal = self.g. This is a predicate on the protected

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

field and the inherited public model field. This is typical of JML and other specification languages, where the connection between a data representation and an abstraction are expressed by a hidden invariant. Frame conditions (modifies clauses) could be encoded in the postcondition, but for brevity we ignore them in our examples.

The lose method of the class WeightLoss, a subtype of Tracker, illustrates the problem with Liskov and Wing's postcondition rule (4). That rule requires that lose's postcondition implies the postcondition in Tracker. However, this implication does not hold for this example, as can be seen by considering the case where self.goal is strictly larger than the difference self.curr − kg, since exc≠null contradicts exc = null. (The specification variable exc refers to exception results; when exc is null in a post-state, then the method has returned normally.) However, in this example, supertype abstraction works fine, because the lose method of the class WeightLoss obeys Tracker's specification for lose whenever Tracker's precondition holds. Thus clients that reason about calls to lose using Tracker's specification will not be surprised, and so WeightLoss can be a behavioral subtype of Tracker. One way of expressing the most flexible sound rule for $S$ to be a behavioral subtype of $T$ is to require that for all subtype objects self : $S$, the precondition rule (3) holds and

$$\textbf{old}(pre_m^T(\mathsf{self})) \wedge post_m^S(\mathsf{self}) \;\Rightarrow\; post_m^T(\mathsf{self}). \tag{5}$$

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

These conditions are just approximations of refinement and it is refinement that explains the equivalence between behavioral subtyping and supertype abstraction (Sect. 2.1 and Theorem 8.15). Our formulation of behavioral subtyping (Def 8.1) is in terms of the intrinsic refinement order on specifications. In Sect. 5.4 we confirm that (3) and (5) are sufficient for proving refinement and thus can be used to check behavioral subtyping. Disentangling the necessary semantic concepts that rely on an intrinsic notion of refinement from proof-theoretic conditions for checking behavioral subtyping is important since the proof-theoretic conditions are sensitive to the form of specifications, as discussed in Sects. 5.4, 9.1, and 10. There are also some technical issues that relate to the changing type of self in subtypes, which are unexpectedly subtle.

In the case of ordinary specifications where the precondition is a predicate on program states and the postcondition is a "two-state predicate" on the initial and final program state, the conditions (3) and (5) are close to a complete characterization of refinement. All that is missing is to conjoin "self **is** $S$" to $pre_m^T$ in (5). The purpose of **old** expressions is to refer the initial state. Occasionally this form of specification is inadequate and it is preferable to use ***specification variables*** scoped over both pre and post condition but not occurring in code. In particular, Hoare logics and axiomatic semantics are often presented using only one-state predicates. For specifications using one-state predicates and specification variables, (5) is incomplete; a complete characterization is given in Sect. 5.4.

Perhaps most surprisingly, if specifications are interpreted in the sense of partial correctness (i.e., not requiring termination), then (3) is not necessary for refinement. This is explored in Sect. 10. It is unclear that there is much practical impact.

We emphasize that model fields and other forms of data abstraction are important in practice, but the ultimate purpose of specifications is to prescribe the program's observable behavior which involves concrete state. Supertype abstraction is a means to reason about observable behavior, so behavioral subtyping is essentially about refinement of behavior *in terms of the same data representation.* One cannot override a public method that has an integer parameter by one with a string parameter, or even one that interprets the integer in pounds instead of kilograms —or rather, doing so would render supertype abstraction unsound. There is a role for data refinement and simulations: for encapsulated data structures and module-scoped or private methods. But the ultimate purpose of data refinement is to establish ordinary refinement of behavior on the observed data representation, as is emphasized in the literature on data refinement (e.g., [Morgan 1994; de Roever and Engelhardt 1998; Banerjee and Naumann 2005]).

### 2.3. Invariants and behavioral subtyping

Object invariants are predicates that can be assumed, in pre-states, by the method implementation and must be established in method post-states —while being hidden in the sense that invariants are not explicit in the contracts used by callers [Hoare 1972]. In Liskov and Wing's abstract model

```
interface Tracker {
  // same as in Fig. 1
}

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

class WeightLoss extends Object implements Track {
  // same as in Fig. 1
}
```

Fig. 2. The interfaces Tracker, Track and the concrete class WeightLoss.

of computation method calls make atomic transitions between states and pointers are not values. Thus there is no direct way to model sharing or reentrancy. These are two features of conventional OO languages that render naive reasoning about object invariants unsound [Noble et al. 1998]. Recently, several methodologies have been proposed, and some implemented, for sound reasoning about invariants (e.g., [Müller 2002; Müller et al. 2006; Naumann and Barnett 2006]). From invariant declarations provided by the programmer, they derive an effective invariant that can soundly be assumed in pre-states of method calls, as the proof obligation on a method implementation, without being explicitly required for callers.

Given sound invariants, the invariant rule of Liskov and Wing turns out to be sound. To be precise, let $Inv$ be a map from types to predicates, satisfying the rule of Liskov and Wing, i.e., $S \leq T$ implies $Inv(S)(\mathsf{self}) \Rightarrow Inv(T)(\mathsf{self})$ in accord with (2). Let $ST$ be a specification table that has behavioral subtyping. Suppose that by some means we ensure that $Inv(T)(\mathsf{self})$ holds at every method call boundary, where $T$ is the exact type of self. Then $ST'$ has behavioral subtyping, where $ST'$ is obtained from $ST$ by conjoining $Inv(S)$ as pre- and post-condition for each method specification in each type $S$.

This result has been proved on the basis of the formalization in this paper. It accounts for supertype abstraction in conjunction with hidden invariants. The details will be presented in a separate paper.

### 2.4. On interface and class types

The Liskov and Wing rule for behavioral subtyping [Liskov and Wing 1994] is applied to all pairs of types and their immediate supertypes. Unlike their work, in this paper we distinguish between interfaces and classes. For supertype abstraction it turns out to be sound and complete to require behavioral subtyping only for instantiable class types. To illustrate this, consider Fig. 2. Suppose Abby develops the interface Tracker first for a trusted client. Later, Ben decides to include something like Tracker in a library for untrusted clients, and thus develops the interface Track, which is specified to throw an exception if Track's precondition for the lose method is not met. Finally, `Carla` implements the Track (and thus also the Tracker) interface in the class WeightLoss. Since Track is a subtype of Tracker, the Liskov and Wing rule would require that the specification of the lose method of Track satisfy conditions (3) and (4) with respect to the specification given in Tracker. However, neither of these conditions is satisfied by the specification of Track. For the precondition of lose, the specification in Track is opposite to that in Tracker. Similarly for the postcondition of lose, the specification in Track requires an exception to be thrown, while that in Tracker ensures the opposite. On the other hand, it is still valid to use supertype abstraction when reasoning about calls to the lose method on variables whose static type is either Track or Tracker, since the lose method in the concrete class WeightLoss refines both specifications.

Dropping the refinement condition for sub-interfaces is important for the completeness of behavioral subtyping with respect to supertype abstraction. However, there is a good reason to require

refinement in every instance of subtyping. Consider reasoning about a call t.lose(5) where t has static type Track. Due to behavioral subtyping, any implementation of lose at a subtype of Track will satisfy the specification of lose in Tracker. So reasoning about t using only the specification given in Track is incomplete. Strengthening that specification by inheriting the one declared in Tracker has no cost, as it imposes no additional constraint on implementations of lose. (See Prop. 9.14.)

### 2.5. Ensuring behavioral subtyping by specification inheritance

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

Such repetition can be avoided using specification inheritance [Dhara and Leavens 1996; Leavens 2006; Wills 1992]. In JML, the programmer declares two specifications, one for the supertype, and one for the subtype. The effective specification for the subtype is derived using specification inheritance, as described below. For methods, an overriding method declaration inherits the specifications of each declaration of the method that it overrides. Its effective specification is the join (least upper bound) of the explicit specification with the specifications of that method in all supertypes. In JML the join of a method specification with the specifications of overridden methods is indicated by the keyword **also**. Using this convention one could rewrite the specification of the lose method in class WeightLoss as follows, with the initial **also** reminding the reader about the join with the inherited specification:

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

In JML, the effective method specification derived from such a specification has as its precondition the disjunction of the explicit precondition and those inherited (which in this example is **true**), and as its postcondition a conjunction of implications, which says that if a precondition was satisfied, then the corresponding postcondition must hold. This corresponds to the semantics of specification inheritance that we formalize in Sect. 9. And it is exactly what is already explicit in the specification of lose in Fig. 1.

A number of tools and logics enforce behavioral subtyping through equivalent means, though this is not always explicit. Theorem 9.10 says specification inheritance ensures behavioral subtyping.

## 3. RELATED WORK

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. Their paper is famous because they clearly present the main ideas of behavioral subtyping and several interesting examples. They present informal arguments why behavioral subtyping ensures their subtype requirement. The reasoning they permit involves only invariants and history constraints, because their model of computation allows concurrency. By contrast, we use a sequential language that allows Hoare-style reasoning using invariants as well as pre- and postconditions, addressing both partial and total correctness. Furthermore, we strengthen their soundness claim to an equivalence, and we rigorously prove our soundness and completeness results.

Leavens and Dhara [2000] survey older work on behavioral subtyping, including the pioneering work of America [1987; 1991] and Meyer [1985] [1997]. Much of it is similar to Liskov and Wing's and has similar limitations.

Early versions of this paper [Leavens and Naumann 2006b; Leavens and Naumann 2006a] gave our definition of supertype abstraction as well as the result that supertype abstraction for method call is equivalent to behavioral subtyping, and a separate result deriving supertype abstraction for commands in general. Based on the wording of a number of subsequent citations, it seems that our results were convincing. Unfortunately, there is a fatal flaw in our proofs of the second result just mentioned. (It only shows up in the case of sequential composition, which had seemed too obvious to check in full detail.) In the present paper we use an entirely different proof technique and give detailed proofs.

Several logics have been given for sequential fragments of Java which incorporate supertype abstraction [Müller 2002; Parkinson 2005; Pierik 2006; Poetzsch-Heffter and Müller 1999]. These logics mostly achieve behavioral subtyping by requiring that each overriding method implementation in a type satisfies the corresponding specification in each of its supertypes. This has the same effect as our definition of specification inheritance, which explicitly constructs an "effective specification" for a method equivalent to all the specifications of that method in each supertype. 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 [2002] considers interface types, but 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. He does not claim or prove completeness, but does prove soundness of his modular techniques (on the assumption that the underlying logic is sound, a plausible conjecture (p97) based on the soundness proof of [Poetzsch-Heffter and Müller 1999] for a closely related logic).

Apt et al. [2012; 2009] address soundness and completeness of Hoare logics for object-based programs but do not address inheritance.

KeY [Beckert et al. 2007] and Spec# [Barnett et al. 2005] are two mature static verifiers that deal with inheritance and provide at least limited forms of supertype abstraction. Spec# features a sound modular treatment of invariants. KeY doesn't enforce a particular invariant methodology but does facilitate use of ownership- and visibility-based methodologies; its treatment of behavioral subtyping is also flexible. Some of the works cited below include prototype verification tools but tools are outside the scope of this paper; we focus on reasoning techniques.

Parkinson's [2005] work is based on separation logic [O'Hearn et al. 2009] and focuses on encapsulation based on a novel form of opaque predicate [Bierman and Parkinson 2005], related to higher order quantification [Biering et al. 2005] and JML's model fields. The proof rules embody controlled scope of the definitions of predicates used, by name, in contracts. Parkinson says "behavioral subtyping" for the standard implications (3) and (4), and "specification compatibility" for a proof-theoretic approximation of the intrinsic refinement ordering [Parkinson 2005, Def. 3.5.1] which retains (3) although as we show this is unnecessarily strong because the logic is for partial correctness.

Parkinson and Bierman [2008] show the benefits, for modular reasoning about all kinds of inheritance, of such opaque predicates together with separation logic. (Such predicates and separation logic fit neatly into our formal model of specifications.) They also promote the use of "static" specifications for methods, which can be used to reason about statically-dispatched super calls and calls to final or private methods. Their treatment does not attempt to disentangle the foundational issues of modular reasoning from their verification logic. They do connect behavioral subtyping with refinement, citing an early version of the present paper. Parkinson and Bierman [2008] point out that by using exact type tests in preconditions, one can maintain behavioral subtyping while allowing arbitrary changes of behavior in subclasses. The point is that instances of a proper subclass are not constrained by a specification with a precondition that says self has exactly the superclass type. They give examples to show that this has practical value as a way to reuse code via ad hoc inheritance. We explore this in detail in Example 8.5. A variant on the technique is to use exact type tests in postconditions (e.g., [Pierik and de Boer 2005a]). This technique can be deployed in languages like JML using pure methods that get re-defined at subtypes; this has the advantage of avoiding the need for exact type tests, which are not a built-in feature in Java-like languages and which are problematic to encode in open programs.

Chin et al. [2008] give results that are similar to those of Parkinson and Bierman [2008], but also explicitly considering invariants. They use a notion of "intersection" of specifications to model specification inheritance, and have a notion of "specification subsumption" corresponding to our notion of refinement, though specialized for the specific setting of separation logic. Like Parkinson and Bierman, they do not seek to analyze behavioral subtyping separately from their particular logic. While they present several examples demonstrating the power of their specification language, they do not address its completeness formally.

Pierik [2006] gives a more conventional proof system, in particular a proof outline logic with a first-order assertion language. As we do in Sect. 5.4, Pierik explicitly connects specification refinement with adaptation rules; and unlike the other partial correctness logics discussed here, his characterization of refinement does not impose the unnecessary strong condition (3). Supertype abstraction and

behavioral subtyping are present but intertwined with many other details. Pierik [2006] and Parkinson [2005] both prove soundness of their partial-correctness logics, and Pierik proves completeness.

Pierik and de Boer [2005a] investigate various notions of completeness of Hoare logics in the presence of behavioral subtyping. They formalize notions closely related to what we call modular correctness. Their definitions do not technically apply to our work, since we investigate semantic notions of reasoning instead of particular Hoare logics.

Dovland et al. [2008] also present a proof system in their paper "Lazy Behavioral Subtyping." The main innovation in their proof system is to distinguish between "requirements," which are specifications used to verify method calls that call other methods defined in the same class, and "specifications". Both requirements and specifications are formalized as pre- and postcondition pairs. A method's specification is only used to specify a particular implementation; in particular the implementation of a method must satisfy its declared specification, and the declared specification must be sufficient to prove that all requirements placed on that method hold. In lazy behavioral subtyping, subtypes inherit requirements for methods in the sense that an overriding method $m$ must satisfy all the requirements placed on $m$ in all ancestor classes. Supertype abstraction is still used for reasoning about method calls (when the receiver is not **this**); however, such calls are verified not with respect to the declared specification of a method, but with respect to the set of requirements remembered for the receiver's static type. So method specifications are not used directly in verification of client code, and thus the system is unusually flexible. Their proof system can be modeled in our work by populating the specification table for each method with the join of the requirements placed on that method's implementation. While the authors present soundness results, they do not attempt to prove completeness nor do they separate their results from their particular proof system.

Findler and Felleisen [2001] is a foundational study of runtime assertion checking for pre/post specifications without invariants. Findler et al. [2001] directly addresses behavioral subtyping. These papers are the first to describe how to use runtime assertion checking to properly check supertype abstraction, since they check method calls using the contracts associated with the static type of the receiver.

The other issue that is important in these papers by Findler et al. is specification inheritance. This is an important design decision in an OO specification language. Many of the above cited works use specification inheritance to force behavioral subtyping. This decision prevents redundancy in specifications and avoids the need for subtypes to see all specifications in supertypes, some of which may be invisible (such as private invariants). The downside of this decision is that the join used in specification inheritance can lead to unsatisfiable specifications, which will manifest themselves during testing as assertion violations. Findler and Felleisen say that such assertion violations will be confusing to programmers; instead they advocate having tools check for instances of what they call an "erroneous contract formulation" [Findler et al. 2001, Section 3] and signal them unambiguously as "hierarchy violations." Thus they require specifiers to write out the entire contract for each method. They point out that specification inheritance can cause precondition violations to be suppressed (when a precondition in a subtype's overriding method is weaker than that of the method it overrides). More importantly, they note a problem with the way that specification inheritance causes "blame" to be assigned for postcondition violations. The problem occurs when an overriding method's specification has a postcondition that contradicts the postcondition specified for the method it overrides; in this case a test will cause one postcondition to be violated, and thus blame will be assigned to the implementation of the method (instead of to the specifier), which Findler and Felleisen believe is confusing. Instead, they argue that such problems should be pointed out to programmers as "hierarchy violations," which their runtime assertion checks can detect. They point out that specification languages that use specification inheritance cannot detect hierarchy violations. Their contract checker checks for hierarchy violations using Liskov and Wing's pre/post rules, (3) and (4). However, conditions (3) and (4) are unnecessarily restrictive as we have explained above.

Our formalization of specifications, which keeps a table of specifications for each method in each type, can model both specification languages that use specification inheritance and those that do not, like Findler and Felleisen's. In the former case the specification table stores the specifications constructed by specification inheritance; in the latter case the specification table stores the specifications written directly by the specifier.

## 4. PROGRAMMING LANGUAGE

Our technical development uses an idealized object-oriented language that models a large fragment of the class-based languages like Java and C#. It includes interfaces, mutually recursive classes, exceptions as first-class objects, type tests and casts, inheritance and dynamic binding, and strong binary methods.[6] Constructors are omitted but can be treated as a syntactic sugar. For issues like evaluation order and the semantics of null casts, where reasonable languages may differ on semantic details, we follow Java.

We chose this language because it closely resembles Java with JML specifications. Both Java and JML are of interest to a large group of programmers, including those working in C# and the Spec# system. Modeling exceptions seems necessary to have a realistic treatment of runtime type casts and of total correctness in a Java-like language, and we are pleased to model them in full generality, although they are not central to our results. Exceptions slightly complicate the definitions in this section of the paper, and also the derivations in Sect. 7, but do not otherwise obtrude. Because standard details about the language are relegated to the appendix, the language features do not significantly lengthen the paper.

The semantics is adapted from Banerjee and Naumann [2005] which in turn draws on Igarashi et al. [2001] for formalization of syntax including the class table. A version of the semantics is formalized in Leavens et al. [2006], building on Naumann [2005]. Because the language is essentially defunctionalized [Banerjee et al. 2001; Reynolds 1972], a denotational semantics can be given using a straightforward hierarchy of inductively defined domains; there are no interesting domain equations to solve. Denotational semantics is well suited to our purposes, both because it supports a semantic notion of modular reasoning and because the semantics of commands is compositional.[7]

Three features of the semantics streamline the formal development but may be unexpected. First, although the conventional distinction is made between expressions $E$ and commands $C$, both may have effects —and we consider pre/post specifications for both. 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. For a command, the final state space is the initial one extended with a distinguished variable, exc, for exceptions. For an expression, the initial state space is the one in which the expression is evaluated and the final state space has just two variables, res for the normal result and exc for exceptions. For a method, the initial variables are the parameters and the receiver variable self; the final state has res and exc as in the case of expressions. A state transformer of a given type is a mapping from initial states to final states or $\perp$, the latter modeling divergence.

The third notable feature is an encoding of exceptions that is easy to work with. 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 final states include the aforementioned variable exc that, in effect, encodes another disjoint sum: the value of exc is either null, which indicates normal termination, or a reference to the exception object. Variable exc is not allowed to occur in the program text but can be used in specifications (exc is used in modeling the **signals** clause in JML).[8]

These features of the semantics streamline not only the semantic definitions but also formulation of the main results of the paper. Nonetheless, an operationally sound semantics for a language of this complexity is inevitably complicated. 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.

### 4.1. Notations

We often use partial functions, treated as sets of pairs. Unqualified, the term ***function*** means "total function"; in the semantics we use the distinguished value $\perp$ so that state transformers are functions.

[6] *Strong binary methods* are those where a method acts directly on the private fields of more than one object, a commonly-used feature that does not fit with some denotational models.

[7] The semantics is not compositional at the level of classes. A denotational semantics that is compositional at the level of classes, such as Reus [2003], is conceptually attractive but mathematically less elementary, and not needed for our results.

[8] Some works make elegant use of monads to encode exceptions and monads to encode state, but our exceptions are first-class in the sense that there may be arbitrary pointers to and from an exception at the time it is thrown. We are not aware of monadic formalizations in this generality.

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 sometimes parenthesize arguments when it is not necessary, for visual clarity.

A raised dot separates a variable binding from its scope, as in $\forall x : \mathsf{int} \cdot a[x] > 0$.

Finite mappings are used for typing contexts and variable stores, written like $[x : K, y : \mathsf{int}]$ or with brackets omitted. The ***extension*** of a finite mapping $g$ to also map $b$ to $z$, where $b \notin dom\, g$, is written $[g\ ,\ b : z]$ with colon binding more tightly than the comma (and less tightly than function application). Note that in square brackets, the comma forms a union of disjoint functions, whereas in parentheses the ordinary comma means tupling as usual. To ***override*** the mapping for an element $c \in dom\, g$ we write $[g \mid c : z]$ for the map that sends $c$ to $z$ and any other $b$ in $dom\, g$ to $g\, b$. The distinction between override and extension helps clarify some definitions, and the use of comma fits with conventional notation for typing contexts.

To streamline the definitions, our meta-language includes notation for the "strict let" of the lift monad: **lets** $x = \alpha$ **in** $\beta$ abbreviates **if** $\alpha = \bot$ **then** $\bot$ **else let** $x = \alpha$ **in** $\beta$, and **let** has its ordinary mathematical meaning.

### 4.2. Syntax

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

| | |
|---|---|
| $K, L \in \mathit{ClassName}$ | names of declared classes |
| $I \in \mathit{InterfaceName}$ | names of declared interfaces |
| $x, y, z, f$ | variable names (for parameters, locals, and fields) |
| $m$ | 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 "$C$; **return** res;". Using res fits with JML and lets us omit return statements.

There is one distinguished interface name, Thr (for "throwable"). There are 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, there can be arbitrary references between exceptions and other objects.

Class and interface declarations have the following forms:

$$\textbf{class}\ \ K\ \ \textbf{extends}\ \ L\ \ \textbf{implements}\ \ \overline{I}\ \{\ \overline{f : T}\quad \overline{mdec}\ \} \tag{6}$$

$$\textbf{interface}\ \ I\ \ \textbf{extends}\ \ \overline{I}\ \{\ \overline{f : T}\quad \overline{msig}\ \} \tag{7}$$

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 marked as "**ghost**". Our results apply to programs that are properly annotated for ghost (specification-only) fields, but the distinction is not needed for our results.

The remaining syntactic categories are defined in Fig. 3. The syntax is in "A-normal form", i.e., subexpressions in various constructs are restricted to be variables. To avoid loss of expressiveness, let-expressions are included. Desugaring transformations are not difficult to define, e.g., a general equality test $E_1 = E_2$ can be desugared (notation $-^o$) by the rule $(E_1 = E_2)^o = \textbf{let}\ x\ \textbf{be}\ E_1^o\ \textbf{in let}\ y\ \textbf{be}\ E_2^o\ \textbf{in}\ x = y$ which preserves order of evaluation and propagation of exceptions. For exception handling, we omit try-finally since it can be desugared using try-catch.[9] Try-catch statements with multiple catch clauses can be desugared to nested try-catch statements.[10]

In Java there is a return type, "void", for methods called only for their effect. Such a method can be desugared to one that always returns false, and a call desugared using an assignment to a fresh local variable.

A ***ref type*** is any non-primitive data type, i.e., a class or interface name, and we define

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

[9] The type test expression is convenient, although at the level of commands it can also be desugared using cast and try/catch.

[10] Albeit in complicated cases extra Boolean flags are needed to explicitly track control flow, see e.g. Van Roy and Haridi [2004, Sec. 2.7.2].

| | | | |
|---|---|---|---|
| $T$ | ::= | $K \mid I \mid \mathsf{bool} \mid \mathsf{int}$ | data type ($K \in$ *ClassName* and $I \in$ *InterfaceName*) |
| *msig* | ::= | $m(\overline{x:T}):T$ | method signature |
| *mdec* | ::= | **meth** *msig* { $C$ } | method declaration |
| $C$ | ::= | $x := E \mid x.f := x$ | assign to variable, to field |
| | $\mid$ | **var** $x:T$ **in** $C$ | local variable block |
| | $\mid$ | $C;\ C \mid$ **if** $x$ **then** $C$ **else** $C$ | sequence, conditional |
| | $\mid$ | **throw** $x \mid$ **try** $C$ **catch**$(x:T)\ C$ | throw, handle exception |
| $E$ | ::= | $x \mid \mathsf{null} \mid \mathbf{true} \mid 0\ldots$ | variable, literals |
| | $\mid$ | $x.f \mid x = y$ | field access, equality test |
| | $\mid$ | $x$ **is** $T \mid (T)\ x \mid$ **new** $K()$ | type test, type cast, object construction |
| | $\mid$ | $x.m(\overline{x}) \mid$ **let** $x$ **be** $E$ **in** $E$ | method call, sequenced local binding |

Fig. 3. Grammar. Bold keywords and punctuation marks including "{" and "}" are terminal symbols.

We omit arithmetic and Boolean operators as their treatment is straightforward, essentially like equality test. Other constants are treated just like **true** and 0.

A complete program is a class table together with a command. A ***class table*** $CT$ is a mapping, with domain the union of disjoint sets *ClassName* and *InterfaceName*, that sends each class name $K$ or interface name $I$ to its declaration.

### 4.3. Typing

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 self and res). Although it is not explicit in the typing judgments, typing depends on the whole class table, owing to mutually-recursive class declarations. Several functions access parts of the class table. Let $CT(K)$ be the class declaration shown in formula (6) above; then we define $\mathit{super}\,K = L$ and $\mathit{superinterfaces}\,K = \overline{I}$.

For a method declaration

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

in class $K$, define its declared method type $\mathit{dmtype}(K,m) = \overline{x:T}\to T_1$. Similarly, for an interface $I$, the notation $\mathit{dmtype}(I,m)$ stands for the type of the signature of $m$ appearing in $I$. Note that method type $\overline{x:T}\to T_1$ is parsed as $(\overline{x:T})\to T_1$ and it records parameter names as well as their types, a minor technical convenience that loses no generality. Method declarations do not list exceptions; any may be thrown (as in C#).

To include inherited methods, function *mtype* is defined as: $\mathit{mtype}(K,m) = \overline{x:T}\to T_1$ iff either $\mathit{dmtype}(K,m) = \overline{x:T}\to T_1$ or $\mathit{mtype}(\mathit{super}\,K,m) = \overline{x:T}\to T_1$ or $\mathit{mtype}(I,m) = \overline{x:T}\to T_1$ for some $I \in \mathit{superinterfaces}\,K$.[11] In a well-formed class table (see below), this will be well-defined. Thus $\mathit{mtype}(K,m)$ is defined iff $m$ is declared or inherited in $K$, whereas $\mathit{dmtype}(K,m)$ is defined only if $m$ is declared in $K$.

Similarly for interfaces: If $I$ extends $\overline{I}$, as in (7), then we define $\mathit{superinterfaces}\,I = \overline{I}$ and $\mathit{mtype}(I,m) = \overline{x:T}\to T_1$ iff either $\mathit{dmtype}(I,m) = \overline{x:T}\to T_1$ or $\mathit{mtype}(I',m) = \overline{x:T}\to T_1$ for some $I' \in \overline{I}$.

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 $m$; this includes the case that $T$ is an interface with a superinterface that declares $m$.

We use the auxiliary function $\mathit{formals}(T,m)$ to pick out the argument names of a method type; $\mathit{formals}(T,m)$ is defined as $\overline{z}$ when $\mathit{mtype}(T,m) = \overline{z:T}\to U$.

For fields, if $CT(K)$ is as in (6), then its list of declared fields is $\mathit{dfields}\,K = \overline{f:T}$ and similarly for $\mathit{dfields}\,I$. To include inherited fields, define $\mathit{fields}\,K = \mathit{fields}\,L \cup \mathit{dfields}\,K \cup (\bigcup\{\mathit{fields}\,I \mid I \in$

[11] Although not necessary, the reader can also assume that Object, Thr, NullDeref and ClassCast have no defined methods and no fields.

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

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

Fig. 4. Selected typing rules.

*superinterfaces* $K$}). In a well formed class table (see below) each of these unions will be disjoint. For interfaces we similarly define $\mathit{fields}\, I = \mathit{dfields}\, I \cup (\bigcup\{\mathit{fields}\, I' \mid I' \in \mathit{superinterfaces}\, I\})$.

The subtype relation $\leq$ on data types is defined inductively by

— $K \leq L$ if $\mathit{super}\, K = L$ (note that $\mathit{super}\, \mathsf{Object}$ is undefined),
— $T \leq I$ if $I \in \mathit{superinterfaces}\, T$,
— $I \leq \mathsf{Object}$ for all $I$, and
— $\leq$ is reflexive and transitive.

Thus for primitive types (int and bool), $T \leq U$ holds just if $T$ is $U$.

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

(W1) The subtype relation, $\leq$, is antisymmetric.
(W2) Any ref type that appears as a field type, superclass, local variable type, cast type, etc. is either declared in $CT$ or is Object, Thr, NullDeref, or ClassCast.
(W3) Field names are not shadowed; that is, for each ref type $T$, if $f : U$ is in $\mathit{dfields}\, T$, then (a) $f$ is not in $\mathit{dom}(\mathit{fields}(\mathit{super}\, T))$, and (b) $f$ is not in $\mathit{dom}(\mathit{fields}\, I)$, for all $I \in \mathit{superinterfaces}\, T$.
(W4) Method types are invariant; i.e., if $T \leq U$ and both $\mathit{dmtype}(T, m)$ and $\mathit{dmtype}(U, m)$ are defined, then $\mathit{dmtype}(T, m) = \mathit{dmtype}(U, m)$.
(W5) For each $K \in \mathit{dom}(CT)$, every method declaration $\textbf{meth}\ m(\overline{x : T}) : T\ \{C\}$ in $CT(K)$ is typable, in the sense that $\Gamma \vdash C$ where $\Gamma$ is the assignment of types to variables $[\mathsf{self} : K, \mathsf{res} : T, \overline{x : T}]$, and moreover exc does not occur in $\overline{x}$. Rules that define the typing judgment $\Gamma \vdash C$ are standard; some appear in Fig. 4 and all can be found in Fig. 10 and Fig. 11 in Sect. A.1. The rules refer to partial function *mtype*, which is well defined given the preceding conditions (W1)–(W4).
(W6) For any $K$, any $I \in \mathit{superinterfaces}\, K$, and any method signature for a method $m$ name declared or inherited in $I$, there is a declared or inherited method in $K$ for $m$ with the same signature.

The symbol $=$ denotes strong equality, when used in connection with partial functions. For example the antecedent of the typing rule for method call is that $\mathit{mtype}(T, m)$ is defined and moreover it equals $\overline{z : U} \to U$.

Although we assume that the type of methods is invariant in subtypes (condition (W4)), we believe that our results would be unchanged if this assumption were relaxed to allow covariance in method return types (as recently added to Java).

Henceforth we assume $CT$ is a fixed, well formed class table.

### 4.4. Semantic domains

The domains encode important invariants: well typed values, absence of dangling references, and non-nullity of self. Fig. 5 is a guide to the domains.

We assume a given set *Ref* of ***references*** —abstract addresses of objects. A ***ref context*** is a finite partial function $r$ that maps references to class names (and not interface names). The idea is that if $o \in \mathit{dom}\, r$ then $o$ is allocated and moreover $o$ points to an object of type $r\, o$. We define the set of reference contexts:

$$\mathit{RefCtx} = \mathit{Ref} \rightharpoonup \mathit{ClassName}$$

where $\rightharpoonup$ denotes finite partial functions. For $r$ and $r'$ in *RefCtx*, the set inclusion $r \subseteq r'$ serves to express that the domain of $r'$ includes at least the objects in $r$ and, for objects allocated in $r$, the types are the same in $r'$.

| | domain | description | metavariables |
|---|---|---|---|
| | $\mathit{RefCtx}$ | typing of allocated refs | $r$ |
| | $\mathit{Val}(T, r)$ | value of type $T$ (in ref context $r$) | $o, v$ |
| | $\mathit{Store}(\Gamma, r)$ | stores for $\Gamma$ | $s, t$ |
| $*$ | $\mathit{Obrecord}(K, r)$ | fields of $K$-objects | |
| | $\mathit{Heap}(r)$ | heaps | $h$ |
| | $\mathit{State}(\Gamma)$ | states for $\Gamma$ | $\sigma, \tau$ |
| | $\mathit{STrans}(\Gamma_1, \Gamma_2)$ | state transformers | $\varphi, \psi$ |
| $*$ | $\mathit{SemExpr}(\Gamma, T)$ | semantic expression | |
| $*$ | $\mathit{SemCommand}(\Gamma)$ | semantic command | |
| $*$ | $\mathit{SemMeth}(K, m)$ | semantics of method $m$ in $K$ | |
| | $\mathit{MethEnv}$ | method environment | $\eta$ |
| | $\mathit{XMethEnv}$ | extended method environment | $\dot{\eta}$ (sometimes $\eta$) |

Fig. 5. Guide to domains. Those marked with $*$ are special cases of the others as explained in the text.

For data type $T$ its domain of values in a reference context $r$ is defined by cases on $T$:

$$\begin{array}{lll} \mathit{Val}(\mathsf{int}, r) & = \mathbb{Z} & \\ \mathit{Val}(\mathsf{bool}, r) & = \{\mathit{true}, \mathit{false}\} & \\ \mathit{Val}(K, r) & = \{\mathit{null}\} \cup \{o \mid o \in \mathit{dom}\, r \wedge r\, o \leq K\}, & \text{if } K \in \mathit{ClassName} \\ \mathit{Val}(I, r) & = \{\mathit{null}\} \cup \{o \mid \exists K \cdot K \leq I \wedge o \in \mathit{Val}(K, r), \} & \text{if } I \in \mathit{InterfaceName} \end{array}$$

A ***store*** for a context $\Gamma$ is a dependent function from variables in scope to type-correct and allocated values. That is, a store $s$ is an element of the dependent function space $(x : \mathit{dom}\,\Gamma) \to \mathit{Val}(\Gamma\, x,\, r)$. What this means is that the domain of $s$ is $\mathit{dom}\,\Gamma$ and $s\, x$ is an element of $\mathit{Val}(\Gamma\, x,\, r)$ for each $x \in \mathit{dom}\,\Gamma$. The domain of stores is also defined to impose an additional invariant of the semantics, namely that $\mathsf{self}$ is never null. Thus we define the set $\mathit{Store}(\Gamma, r)$ by

$$s \in \mathit{Store}(\Gamma, r) \iff s \in ((x : \mathit{dom}\,\Gamma) \to \mathit{Val}(\Gamma\, x,\, r)) \;\wedge\; (\mathsf{self} \in \mathit{dom}\,\Gamma \;\Rightarrow\; s(\mathsf{self}) \neq \mathit{null}) \qquad (8)$$

A ***heap*** $h$ maps each allocated reference $o$ to a store, $h\, o$, of the object's current field values:

$$\begin{array}{lll} \mathit{Obrecord}(K, r) & = & \mathit{Store}(\mathit{fields}\, K,\, r) \\ \mathit{Heap}(r) & = & (o : \mathit{dom}\, r) \to \mathit{Obrecord}(r\, o,\, r) \end{array}$$

A ***state*** is a ref context, $r$, together with heap and store that are well formed in $r$:

$$\mathit{State}(\Gamma) \;=\; (r : \mathit{RefCtx}) \times \mathit{Heap}(r) \times \mathit{Store}(\Gamma, r)$$

This is isomorphic to a semantics where the dynamic type of each object is stored in a distinguished, immutable field.

The most important domain is ***state transformers***. An element of $\mathit{STrans}(\Gamma, \Gamma')$ is a function $\varphi$ that maps each state $\sigma$ in $\mathit{State}(\Gamma)$ to either $\bot$ or a state $\varphi\, \sigma$ in $\mathit{State}(\Gamma')$, with a possibly extended heap, subject to some additional conditions.

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

The conditions help streamline some definitions in Sect. 5. Relation *extState* is used to say that the ref context of the initial state is extended by the ref context of the final state:

$$\mathit{extState}((r, h, s), (r', h', s')) \iff r \subseteq r'.$$

Relation *imSelf* says that $\mathsf{self}$, if present, is immutable:

$$\mathit{imSelf}((r, h, s), (r', h', s')) \iff (\mathsf{self} \in (\mathit{dom}\, s \cap \mathit{dom}\, s') \;\Rightarrow\; s(\mathsf{self}) = s'(\mathsf{self})). \qquad (9)$$

The domain of state transformers subsumes meanings for expressions, commands, and methods, which are defined as follows:

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

We are usually interested in initial state spaces that include self but for expressions and method meanings self is absent from the final state space.

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

$$MethEnv \;=\; (K : ClassName) \times (m : Meths\ K) \to SemMeth(K, m).$$

A normal method environment $\eta$ is defined on pairs $(K, m)$ where $K$ is a class with method $m$; and $\eta(K, m)$ is a state transformer suitable to be the meaning of a method of type $mtype(K, m)$. In case $m$ is inherited in $K$ from class $L$, $\eta(K, m)$ will be the restriction of $\eta(L, m)$ to receiver objects of type $K$.

In our formulation of modular reasoning based on static types, we need to associate 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 $Val(I, r)$). The set of ***extended method environments*** is defined by

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

The metavariable $\eta$ ranges over normal method environments and $\dot{\eta}$ is used to range over extended method environments —the dot is mnemonic for i in "interface". Unqualified, the term "method environment" means normal unless context indicates otherwise.

A command in context, $\Gamma \vdash C$, denotes a function, defined later, from method environments to $SemCommand(\Gamma)$. An expression in context $\Gamma \vdash E : T$ denotes a function from method environments to $SemExpr(\Gamma, T)$. A class table denotes a normal method environment obtained as a least fixed point.

### 4.5. Semantics of expressions and commands

The semantics of expressions and commands is defined by recursion on typing derivations, which are unique.[12] For any derivable typing $\Gamma \vdash E : T$ and any method environment $\eta$, the semantics, $[\![\Gamma \vdash E : T]\!](\eta)$, is an element of $SemExpr(\Gamma, T)$. For any derivable typing $\Gamma \vdash S$ the semantics, $[\![\Gamma \vdash S]\!](\eta)$, is in $SemCommand(\Gamma)$.

In case $E$ has subexpressions, the definition uses nomenclature from the corresponding typing rule. For example, here is the semantics of let-expression, as typed in Fig. 4:

$$\begin{array}{l}
[\![\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ E\ \textbf{in}\ E1 : U]\!](\eta)(r, h, s)\ = \\
\quad \textbf{lets}\ (r_0, h_0, s_0) = [\![\Gamma \vdash E : T]\!](\eta)(r, h, s)\ \textbf{in} \\
\quad \textbf{if}\ s_0\,\mathsf{exc} \neq \mathit{null}\ \textbf{then}\ (r_0, h_0, [\mathsf{res} : \mathit{default}\ U, \mathsf{exc} : s_0\,\mathsf{exc}]) \\
\quad \textbf{else let}\ s_1 = [s\,,\ x : s_0\,\mathsf{res}]\ \textbf{in}\ [\![\Gamma, x : T \vdash E1 : U]\!](\eta)(r_0, h_0, s_1).
\end{array} \tag{10}$$

An example of the command semantics is the semantics of assignment:

$$\begin{array}{rcl}
[\![\Gamma \vdash x := E]\!](\eta)(r, h, s) & = & \textbf{lets}\ (r_1, h_1, s_1) = [\![\Gamma \vdash E : T]\!](\eta)(r, h, s)\ \textbf{in} \\
& & \textbf{if}\ s_1\,\mathsf{exc} = \mathit{null}\ \textbf{then}\ (r_1, h_1, [[s \mid x : s_1\,\mathsf{res}]\,,\ \mathsf{exc} : \mathit{null}]) \\
& & \textbf{else}\ (r_1, h_1, [s\,,\ \mathsf{exc} : s_1\,\mathsf{exc}]).
\end{array} \tag{11}$$

If $E$ yields $\perp$ then so does the assignment (owing to **lets**). If $E$ throws no exception then its value, $s_1\,\mathsf{res}$, 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, $s_1\,\mathsf{exc}$.

The preceding definitions use the notation $[\![-]\!]$ for the semantics function. This abbreviates two definitions, both 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. Define

$$\begin{array}{rcl}
\mathcal{D}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\eta)(r, h, s) & = & \textbf{if}\ s\,x = \mathit{null}\ \textbf{then}\ \mathit{except}(r, h, U, \mathsf{NullDeref}) \\
& & \textbf{else let}\ K = r(s\,x)\ \textbf{in let}\ \overline{z} = \mathit{formals}(K, m)\ \textbf{in} \\
& & \quad \textbf{let}\ s_1 = [\mathsf{self} : s\,x,\ \overline{z : s\,y}]\ \textbf{in}\ \eta(K, m)(r, h, s_1).
\end{array} \tag{12}$$

The exception helper function, *except*, is discussed later. By well-formedness of the class table, $\mathit{formals}(K, m) = \mathit{formals}(T, m)$ where $T$ is the type of $x$ in $\Gamma$ in accord with the typing rule. The receiver object is $s\,x$, thus the dynamic type of the object is given by $K = r(s\,x)$. To look up in

[12] Unique modulo the minor issue that the expression null can sometimes be given more than one type; the semantics can be shown to be independent of this small variation in typing derivations.

method environment $\eta$ the meaning of the dynamically dispatched method we write $\eta(K, m)$. It is applied to state $(r, h, s_1)$ containing the arguments. Since the argument expressions $\overline{y}$ are variables we can write $\overline{s\,y}$ for the sequence of their values.[13] Because the dynamic type of the receiver is a class (specifically, $K$), this semantics is based on a normal method environment.

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 $\dot{\eta}$:

$$\begin{array}{rcl} \mathcal{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\dot{\eta})(r,h,s) & = & \textbf{if } s\,x = \mathit{null} \textbf{ then } \mathit{except}(r,h,U,\mathsf{NullDeref}) \\ & & \textbf{else let } T = \Gamma\,x \textbf{ in let } \overline{z} = \mathit{formals}(T,m) \textbf{ in} \\ & & \quad \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } \dot{\eta}(T,m)(r,h,s_1). \end{array} \tag{13}$$

In keeping with our use of nomenclature from the typing rules, there is no need for the explicit binding $T = \Gamma\,x$ here, but we include it for contrast with the dynamic semantics.

Notation in the method call semantics has been chosen to cater for later formulations. An alternative formulation may be given using $\mathit{dispatch}(T, m, \eta)$ defined by

$$\mathit{dispatch}(T,m,\eta)(r,h,s) = \eta(r(s(\mathsf{self})),m)(r,h,s) \qquad \text{for any } r,h,s. \tag{14}$$

Here we assume $\mathit{mtype}(T, m) = \overline{z : T} \to U$, and then $\mathit{dispatch}(T, m, \eta)$ is a state transformer of type $[\mathsf{self} : T, \overline{z : T}] \to [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thr}]$. In the alternative formulation, the difference between static and dynamic semantics boils down to $\dot{\eta}(T, m)(r, h, s_1)$ versus $\mathit{dispatch}(T, m, \eta)(r, h, s_1)$.

Semantics for the remaining expression and command forms is in Sect. A.1.

It is straightforward but not trivial to prove that the semantic clauses are all well defined and yield elements of the appropriate semantic domains. In fact this property is a strong form of type soundness. In the rest of the paper, we often use without mention various consequences of type soundness. For example, the result value of an expression is in the right domain, which implies that it is allocated. The invariants about self are occasionally important too.

### 4.6. Semantics of class tables

A well formed class table $CT$ denotes a method environment, $\mathcal{D}[\![CT]\!]$, defined to be the least upper bound of an ascending chain of method environments—the ***approximation chain***—each of which is given using the dynamic dispactch command semantics for method bodies, applied to the preceding approximation. In operational terms, the $i$th element in the chain, $\eta_i$, gives the correct semantics for executions with method call stack bounded in length by $i$. (With $\eta_0(K, m)$ being everywhere $\bot$.) In case class $K$ declares $m$ with body $C$, the meaning $\eta_i(K, m)$ is defined as $\mathcal{D}[\![C]\!](\eta_{i-1})$. In case class $K$ declares $m$ from $L$, the meaning $\eta_i(K, m)$ is defined to be $\eta_i(L, m)$, or rather the restriction of $\eta_i(L, m)$ to states with self of type $K$. The details are in Appendix Sect. A.2.

## 5. SPECIFICATIONS AND REFINEMENT

This section formalizes method specifications and their satisfaction by state transformers, in the sense of total correctness (Sect. 5.1). On this basis we define the intrinsic refinement relation between specifications (Sect. 5.2). For refinement by a specification at a subtype, two different notions are defined; one is critical for the equivalence between supertype abstraction and behavioral subtyping, the other is needed for specification inheritance. Sect. 5.3 gives some technical results. Sect. 5.4 characterizes refinement in terms of pre- and postconditions, cf. Eq. (5) in Sect. 2.

### 5.1. Specifications and satisfaction

In practice, most method specifications are written using *two-state* postconditions over program state, with special syntax like "**old**(x)" to refer to the initial state, together with a frame condition ("modifies clause") that delimits what part of the program state is allowed to be changed. A specification of this kind can be desugared into one where the pre- and post-conditions are one-state predicates, using specification variables universally quantified over the Hoare triple [Apt 1981] as is made explicit in JML's "forall" notation (cf. Example 5.6 below). In this paper we consider specifications of arbitrary commands, for which one-state postconditions are convenient, as in axiomatic semantics and in Hoare logics (including [Pierik 2006; Bierman and Parkinson 2005; Dovland et al.

[13] The notation $\overline{z : s\,y}$ means the sequence $z_1 : s\,y_1, \ldots, z_n : s\,y_n$, where $\overline{z}$ and $\overline{y}$ both have length $n$.

2008]). For our purposes, it is convenient to distinguish specification variables from ordinary program variables by considering indexed families of pre/post predicate pairs. A ***predicate*** is just a set of states. We also use the notion of ***state transformer type***, notation $\Gamma \rightsquigarrow \Gamma'$, for specifications of state transformers in $STrans(\Gamma, \Gamma')$.

*Definition* 5.1 (*state transformer specification*). A ***simple specification of type*** $\Gamma \rightsquigarrow \Gamma'$ is a pair $(pre, post)$ such that $pre$ is a subset of $State(\Gamma)$ and $post$ is a subset of $State(\Gamma')$

A ***general specification of type*** $\Gamma \rightsquigarrow \Gamma'$ is a triple $(J, pre, post)$ consisting of a non-empty set $J$ and indexed families of predicates: $pre \in J \to \wp(State(\Gamma))$ and $post \in J \to \wp(State(\Gamma'))$.

A ***method specification of type*** $(T, m)$ is a general specification of type $[\mathsf{self} : T, \overline{x : T}] \rightsquigarrow [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thr}]$ where $mtype(T, m) = \overline{x : T} \to U$.

A $\Gamma$***-specification*** is a general specification of type $\Gamma \rightsquigarrow [\Gamma\,,\ \mathsf{exc} : \mathsf{Thr}]$.

Unqualified, "specification" means general specification unless the context indicates otherwise. A simple specification $(p, q)$ lifts to a general specification with singleton index set: $(\{0\}, \{(0, p)\}, \{(0, q)\})$.

*Remark* 5.2. The requirement that the index set $J$ is non-empty is a technical convenience that loses no generality. A specification with empty index set would impose no constraint at all. The same effect can be achieved using an empty precondition, e.g., take $J$ to be the singleton $\{0\}$, $pre_0$ to be the empty set, and $post_0$ arbitrary. This is satisfied by all state transformers, including the everywhere-divergent $\lambda\sigma \cdot \bot$.

*Definition* 5.3 (*satisfaction by state transformer,* $\models$). Let $(pre, post)$ be a simple specification of type $\Gamma \rightsquigarrow \Gamma'$ and $\varphi$ be in $STrans(\Gamma, \Gamma')$. Then $\varphi$ ***satisfies*** $(pre, post)$, written $\varphi \models (pre, post)$, iff

$$\forall \sigma \cdot \sigma \in pre \;\Rightarrow\; \varphi(\sigma) \in post. \tag{15}$$

For general specification $(J, pre, post)$, define $\varphi \models (J, pre, post)$ iff $\varphi \models (pre_i, post_i)$ for all $i \in J$.

Because postconditions are sets of states, and $\bot$ is not a state, this notion of satisfaction is total correctness, i.e., termination is required.[14]

We introduce some terminology and notation for specifications given in terms of a two-state postcondition.

*Definition* 5.4 (*specification in two-state form,* $\langle\!\langle -, - \rangle\!\rangle$). A ***specification in two-state form***, of type $\Gamma \rightsquigarrow \Gamma'$, is a pair $(P, R)$ with precondition $P \subseteq State(\Gamma)$ and postcondition $R \subseteq State(\Gamma) \times State(\Gamma')$. Given $(P, R)$ we define the general specification $\langle\!\langle P, R \rangle\!\rangle$ to be $(J, pre, post)$ where $J = State(\Gamma)$ and $pre_\sigma = \{\tau \mid \tau = \sigma \wedge \sigma \in P\}$ and $post_\sigma = \{\tau \mid (\sigma, \tau) \in R\}$ for all $\sigma \in State(\Gamma)$.

For a given $\sigma$, $pre_\sigma$ is either empty or a singleton. Furthermore, a state transformer satisfies $\langle\!\langle P, R \rangle\!\rangle$ just if it satisfies $(P, R)$ in the usual sense that $\sigma \in P \;\Rightarrow\; (\sigma, \varphi(\sigma)) \in R$ for all $\sigma$. This encoding is similar to what is done by some verification-condition generators, which interpret **old** in terms of an auxiliary variable that "snapshots" the entire program state.

*Example* 5.5. Consider a method gain in a type Tracker with signature lbs : int→int. A specification for this method in Eiffel or JML that uses **old** might look like the following:

**requires** $0 < \mathsf{lbs}$; **ensures** $\mathsf{self.curr} = \mathbf{old}(\mathsf{self.curr} + \mathsf{lbs}) \wedge \mathsf{res} = \mathsf{self.curr} \wedge \mathsf{exc} = \mathsf{null}$;

In this example, the initial and final state spaces are $\Gamma_0 = [\mathsf{self} : \mathsf{Tracker}, \mathsf{lbs} : \mathsf{int}]$ and $\Gamma'_0 = [\mathsf{res} : \mathsf{int}, \mathsf{exc} : \mathsf{Thr}]$. Mention of self in the postcondition can be interpreted with respect to its value in the initial state, because self is immutable. This is spelled out in the next Example.

*Example* 5.6. In JML, one can write out a specification that is something like our general specification by declaring specification variables. Again consider a method gain in a type Tracker with signature lbs : int→int. A specification with explicit specification variable declarations would look like the following, where the scope of ocurr and olbs extends over both the pre- and postcondition:

[14] Our language could be extended to include bounded nondeterministic choice, using state transformers that return finite sets (the Smyth powerdomain), and then this definition of satisfaction would use $\varphi(\sigma) \subseteq post$ in place of $\varphi(\sigma) \in post$. Key results such as the characterization of refinement, Proposition 5.18, would be unchanged.

**forall** ocurr, olbs: int;
**requires** $0 < \mathsf{lbs} \wedge \mathsf{ocurr} = \mathsf{self.curr} \wedge \mathsf{olbs} = \mathsf{lbs}$;
**ensures** $\mathsf{self.curr} = \mathsf{ocurr} + \mathsf{olbs} \wedge \mathsf{res} = \mathsf{self.curr}$;

As an alternative to the encoding of Def. 5.4, one can directly use the product of value domains declared for the specification variables as the index set in the general specification. In this example, the product space is $\mathbb{Z} \times \mathbb{Z}$, so the corresponding specification is $(\mathbb{Z} \times \mathbb{Z}, pre, post)$ where for $(oc, ol)$ in $\mathbb{Z} \times \mathbb{Z}$ we define $pre_{(oc,ol)} = \{(r, h, s) \mid 0 < s(\mathsf{lbs}) \wedge oc = h(s\,\mathsf{self})(\mathsf{curr}) \wedge ol = s(\mathsf{lbs})\}$ and $post_{(oc,ol)} = \{(r', h', s') \mid h'(s\,\mathsf{self})(\mathsf{curr}) = (oc + ol)) \wedge s'\mathsf{res} = h'(s\,\mathsf{self})(\mathsf{curr}) \wedge s'\mathsf{exc} = null\}$. Note that we use $s\,\mathsf{self}$, not $s'\,\mathsf{self}$.

*Example* 5.7. Specification languages such as JML feature a "modifies clause" that expresses frame conditions. These can be interpreted as two-state postconditions that say any pre-existing location not explicitly mentioned in the clause has the same final value as initially. Many tools desugar the modifies clause by means of a ghost variable of type "heap" or "state" that snapshots the initial state as in Def. 5.4.

*Example* 5.8. In separation logic, preconditions and postconditions denote sets of states of the form $(s, h)$ where $s$ is a store as in our semantics but $h$ is a partial heap, i.e., its objects may have dangling pointers (and so may $s$). Let us ignore the store and sketch one encoding of such a specification in our formalism. Let $p$ and $q$ be sets of partial heaps. To encode the specification with precondition $p$ and postcondition $q$, let $h\#k$ mean that the domains of $h$ and $k$ are disjoint, and for such heaps let $h * k$ be their union. Our specification is $(J, pre, post)$ where $J$ is the set of all partial heaps. For $h \in J$, define $pre_h = \{k * h \mid k \in p \wedge h\#k \wedge closed(k * h)\}$ and similarly $post_h = \{k * h \mid k \in q \wedge h\#k \wedge closed(k * h)\}$. By restricting to closed heaps, i.e., without dangling pointers, we get well formed states according to our definitions. By fixing $h$ between pre- and postcondition, we express the implicit frame conditions of the "tight interpretation" of Hoare triples in separation logic [O'Hearn et al. 2001].

Data refinement mechanisms like model fields are ultimately means to define predicates on concrete state. Such mechanisms involve notations that may desugar differently in different contexts (see Example 8.5). For our purposes we model such specifications in terms of multiple concrete specifications, in a specification table (see Sect. 6).

### 5.2. Refinement of specifications and satisfaction at a subtype

The behavioral subtyping property is expressed in terms of a refinement ordering on specifications: It says that if $S$ is a subtype of $T$ then, for each method $m$ of $T$, the specification of $m$ in $S$ is stronger than that in $T$. Being "stronger" means that any method satisfying the specification in $S$ also satisfies the one in $T$. This intrinsic ordering on specifications, written $\sqsupseteq$, is determined by the nature of command denotations and the definition of satisfaction. Some care needs to be taken with the details, because if $S$ is a class, a method in class $S$ is defined to act on receiver objects of type $S$ whereas a specification of type $(T, m)$ imposes a requirement on state transformers for target type $T$. Owing to the semantics of dynamic dispatch, it is sound for a method in class $S$ to assume a strengthened precondition saying that the receiver object has type $S$. (This is explicit in the proof obligation for method bodies in some proof systems and tools, e.g., [Müller 2002; Pierik and de Boer 2005b; Beckert et al. 2007; Bierman and Parkinson 2005].)

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

It turns out that there are important uses for both restriction to an exact type and restriction to all subtypes. To refer to the exact type of self, in a state with self, we define

$$selftype(r, h, s) = r(s(\mathsf{self})) \tag{16}$$

*Definition* 5.9 (*subtype restriction,* $\lfloor T$, $\lfloor^* T$). Let $(pre, post)$ be a simple specification of type $\Gamma \rightsquigarrow \Gamma'$, where $\mathsf{self} \in dom\,\Gamma$, and let $T \leq \Gamma\,\mathsf{self}$. The ***exact restriction*** of $pre$ to $T$, written $pre\lfloor T$, is defined by

$$\sigma \in pre\lfloor T \iff selftype(\sigma) = T \wedge \sigma \in pre.$$

Define $(pre, post)\lfloor T$ to be the specification $(pre\lfloor T, post)$, of type $[\Gamma \mid \mathsf{self}: T] \rightsquigarrow \Gamma'$. Define $(J, pre, post)\lfloor T$ to be $(J, pre', post)$ where $pre'_i = pre_i\lfloor T$ for all $i \in J$.

The ***downward restriction*** $\lfloor^* T$ uses "$\leq T$" in place of "$= T$", i.e.,

$$\sigma \in pre\lfloor^* T \iff selftype(\sigma) \leq T \wedge \sigma \in pre.$$

Recall that self is never null, according to the definition of states, cf. Eq. (8). Thus the condition $selftype(\sigma) = T$, expressing that the exact type of self is $T$, is well defined, as is condition $selftype(\sigma) \leq T$ (which is denoted by the program expression self **is** $T$).

The exact type of an object is always a class, so $pre\lfloor T$ is empty in case $T$ is an interface type. Thus $\lfloor^* T$ is interesting for any ref type $T$ but $\lfloor T$ is only interesting when $T$ is a class. Also, in the case that $T$ is $\Gamma(\mathsf{self})$ we have $pre\lfloor^* T = pre$ but in general only $pre\lfloor T \subseteq pre$.

*Definition* 5.10 (*specification refinement* $\sqsupseteq$*; at a subtype* $\sqsupseteq^T, \sqsupseteq^{*T}$). Let $spec_0$ and $spec_1$ be specifications of type $\Gamma \rightsquigarrow \Gamma'$. Then $spec_1$ ***refines*** $spec_0$, written $spec_1 \sqsupseteq spec_0$, if and only if

$$\varphi \models spec_1 \Rightarrow \varphi \models spec_0 \quad \text{for all } \varphi \in STrans(\Gamma, \Gamma').$$

Let $spec_2$ be of type $[\Gamma \mid \mathsf{self}: T] \rightsquigarrow \Gamma'$ where $T \leq \Gamma\,\mathsf{self}$. ***Refinement at exact subtype*** $T$, written $\sqsupseteq^T$, is defined by

$$spec_2 \sqsupseteq^T spec_0 \iff spec_2 \sqsupseteq spec_0\lfloor T.$$

The ***refinement at a downward subtype***, $\sqsupseteq^{*T}$, is defined by

$$spec_2 \sqsupseteq^{*T} spec_0 \iff spec_2 \sqsupseteq spec_0\lfloor^* T.$$

We take care never to omit the superscripts on the refinement symbol, so $\sqsupseteq$ by itself always implies both sides have the same type. In the definitions of $\sqsupseteq^T$ and $\sqsupseteq^{*T}$, the specifications on the right are at type $T$, so if $T < \Gamma\,\mathsf{self}$, we have $spec_2 \sqsupseteq^T spec_0$ iff

$$\varphi \models spec_2 \Rightarrow \varphi \models spec_0\lfloor T$$

for all $\varphi$ of type $[\Gamma \mid \mathsf{self}: T] \rightsquigarrow \Gamma'$. *Mutatis mutandis* for $\sqsupseteq^{*T}$

If $T = \Gamma\,\mathsf{self}$ then $\sqsupseteq^{*T}$ is the same as $\sqsupseteq$. That is not the case for $\sqsupseteq^T$ (because $pre\lfloor T \subseteq pre$ is a proper inclusion in general). We also have

$$spec_0\lfloor^* T \sqsupseteq spec_0\lfloor T \tag{17}$$

for $spec_0$ and $T$ as in Def. 5.10, by precondition weakening and the fact that $P\lfloor^* T \supseteq P\lfloor T$ for any predicate $P$. Formally, (17) is a consequence of Proposition 5.18 below.

Leaving subtyping aside, we note that $\sqsupseteq$ is not antisymmetric. We write $(J, pre, post) = (J', pre', post')$ to mean equality in the metatheory, i.e., $J$ is $J'$, $pre$ is $pre'$, and $post$ is $post'$. It can happen that two distinct specifications are equivalent in the sense that they are satisfied by the same state transformers, and are thus mutually refined.

*Example* 5.11. As an example of distinct but equivalent specifications, suppose $x$ and $y$ are distinct variables of type int. Let $J = J' = \mathbb{N}$. Let $pre_i$ be the states where $x = i \wedge i \geq 1$ and $post_i$ be the states where $y = i$. Let $pre'_i = pre_i$ and $post'_i = \{\sigma \mid \sigma \in post_i \wedge i \geq 1\}$. So $post \neq post'$ but the specifications are satisfied by the same state transformers.

Another example of equivalent but distinct specifications is given by changing Def. 5.4 to use $post_\sigma = \{\tau \mid \sigma \in P \wedge (\sigma, \tau) \in R\}$.

One could make $\sqsupseteq$ antisymmetric by equating $spec$ with $spec'$ if $spec \sqsupseteq spec'$ and $spec' \sqsupseteq spec$. But little would be gained. It is enough that $\sqsupseteq$ is reflexive and transitive, as is easily shown. We occasionally write $\simeq$ for the associated equivalence relation.

Transitivity for refinement at a subtype is delicate and is considered in the next subsection. Reflexivity is not too interesting for refinement at a subtype: $spec \sqsupseteq^T spec$ and $spec \sqsupseteq^{*T} spec$ are not defined unless $T = \Gamma\,\mathsf{self}$, in which case both are true.

The relation $\sqsupseteq^{*T}$ is stronger than $\sqsupseteq^T$ in the following sense.

LEMMA 5.12. Suppose $spec_0$ has type $\Gamma \rightsquigarrow \Gamma'$ and $spec_1$ has type $[\Gamma \mid \mathsf{self}: T]$ with $T \leq \Gamma\,\mathsf{self}$ Then $spec_1 \sqsupseteq^{*T} spec_0$ implies $spec_1 \sqsupseteq^T spec_0$.

PROOF. Observe that $(spec_1 \sqsupseteq^{*T} spec_0) \iff (spec_1 \sqsupseteq spec_0 \lfloor^* T) \Rightarrow (spec_1 \sqsupseteq spec_0 \lfloor T) \iff (spec_1 \sqsupseteq^T spec_0)$, where the middle step is by Eq. (17) and transitivity of $\sqsupseteq$. □

For an example that the implication is strict, suppose $T$ is a class and there is at least one proper subclass $K$ of $T$. Let the precondition of $spec_0$ be true and the precondition of $spec_1$ say self has exactly type $T$. Let $spec_0$ and $spec_1$ have the same, nontrivial postcondition. Then $spec_1 \sqsupseteq^T spec_0$, because this considers only the exact type restriction $spec_0 \lfloor T$, but $spec_1 \not\sqsupseteq^K spec_0$ and so $spec_1 \not\sqsupseteq^{*T} spec_0$.

### 5.3. Some properties of refinement

This and the next subsection contain technical results on refinement. Some readers may prefer to skip to Sect. 6 and refer back as needed.

LEMMA 5.13. Suppose *spec* has type $\Gamma \rightsquigarrow \Gamma'$. For any type $T$ and class $K$, if $T \leq \Gamma\,\mathsf{self}$ and $K \leq T$ then $spec \lfloor^* T \lfloor K = spec \lfloor K$. Moreover $spec \lfloor^* T \lfloor^* K = spec \lfloor^* K$.

PROOF. By definitions, the equality $(spec \lfloor^* T) \lfloor K = spec \lfloor K$ boils down to the property that $P \lfloor^* T \lfloor K = P \lfloor K$ for any $\Gamma$-predicate $P$, which is easily proved from the definitions. Similarly for $spec \lfloor^* T \lfloor^* K = spec \lfloor^* K$. □

Note that it is not that case that $spec \lfloor T \lfloor K = spec \lfloor K$. If $K$ is a strict subtype of $T$ then $P \lfloor T \lfloor K$ is empty because $P \lfloor T$ contains only states where self is exactly $T$.

Although ordinary refinement ($\sqsupseteq$) is transitive, the situation is more delicate for refinement at a subtype.

LEMMA 5.14 (QUASI-TRANSITIVITY OF $\sqsupseteq^T$). Suppose $spec_0$ and $spec_1$ are specifications of type $\Gamma \rightsquigarrow \Gamma'$, and $spec_2$ is of type $[\Gamma \mid \mathsf{self} : U] \rightsquigarrow \Gamma'$ with $U \leq \Gamma\,\mathsf{self}$. If $spec_2 \sqsupseteq^U spec_1$ and $spec_1 \sqsupseteq spec_0$ then $spec_2 \sqsupseteq^U spec_0$, provided $spec_1$ is satisfiable.

PROOF. To show $spec_2 \sqsupseteq^U spec_0$, assume $\varphi \in STrans([\Gamma \mid \mathsf{self} : U], \Gamma')$ and $\varphi \models spec_2$. We must show that $\varphi \models spec_0 \lfloor U$. From $spec_2 \sqsupseteq^U spec_1$ we get $\varphi \models spec_1 \lfloor U$; but this does not yield $\varphi \models spec_1$ which is not well-defined (because $\varphi$ is only defined on states with $\mathsf{self} : U$). Define $\psi \in STrans(\Gamma, \Gamma')$ by

$$\psi\,\sigma \quad = \quad \textbf{if } selftype(\sigma) = U \textbf{ then } \varphi\,\sigma \textbf{ else } \tau,$$

where $\tau$ is an arbitrarily chosen state that satisfies $spec_1$ for initial state $\sigma$, and *selftype* is defined in Eq. (16). From $\varphi \models spec_1 \lfloor U$ we get $\psi \models spec_1$ (using the definition of $\models$). Then by $spec_1 \sqsupseteq spec_0$ we get that $\psi \models spec_0$. Now $\varphi \models spec_0 \lfloor U$ follows from $\psi \models spec_0$ by definitions of $\psi$, $\models$, and $\lfloor U$. □

Satisfiability ensures that $\tau$ exists in the preceding proof. 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 $STrans(\Gamma, \Gamma')$ satisfies $spec_1$, because self is immutable (recall Eq. (9)). Owing to unsatisfiability we have $spec_1 \sqsupseteq spec_0$ 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 $spec_2 \not\sqsupseteq^U spec_0$.

LEMMA 5.15 (QUASI-TRANSITIVITY OF $\sqsupseteq^{*T}$). For $spec_0$ of type $\Gamma \rightsquigarrow \Gamma'$, satisfiable $spec_1$ of type $[\Gamma \mid \mathsf{self} : T] \rightsquigarrow \Gamma'$ where $T \leq \Gamma\,\mathsf{self}$, and $spec_2$ of type $[\Gamma \mid \mathsf{self} : U] \rightsquigarrow \Gamma'$ with $U \leq T$, we have

$$spec_2 \sqsupseteq^{*U} spec_1 \wedge spec_1 \sqsupseteq^{*T} spec_0 \;\Rightarrow\; spec_2 \sqsupseteq^{*U} spec_0.$$

The proof is very similar to the preceding one, and still requires satisfiability of $spec_1$.

If we replace $\sqsupseteq^{*U}$ and $\sqsupseteq^{*T}$ in the antecedent by $\sqsupseteq^U$ and $\sqsupseteq^T$, we do not get $\sqsupseteq^U$ as consequent (e.g., consider $U \neq T$). Also, if we replace $\sqsupseteq^{*U}$ and $\sqsupseteq^{*T}$ by $\sqsupseteq^{*U}$ and $\sqsupseteq^T$, we get neither $\sqsupseteq^{*U}$ nor $\sqsupseteq^U$ as consequent.

By definitions, a specification is satisfiable if it is refined by a satisfiable specification. Satisfiable specifications can be characterized as follows.

LEMMA 5.16 (SATISFIABILITY). A specification $(J, pre, post)$ of type $\Gamma \rightsquigarrow \Gamma'$ is satisfiable iff $\forall \sigma \in State(\Gamma) \cdot (\exists j \in J \cdot \sigma \in pre_j) \Rightarrow \exists \tau \in State(\Gamma') \cdot (\forall j \in J \cdot \sigma \in pre_j \Rightarrow \tau \in post_j)$.

PROOF. (sketch) By mutual implication. If $\varphi$ satisfies the specification, then $\varphi(\sigma)$ provides a witness for $\tau$ in the condition. Conversely, if the condition holds then a satisfying state transformer

$\varphi$ can be defined as follows: For any $\sigma$, if $\sigma$ is in some $pre_j$ then define $\varphi(\sigma)$ to be some chosen $\tau$ that satisfies the condition; otherwise, $\varphi(\sigma)$ can be any state or $\bot$. $\square$

Consider a specification given by formulas, as in Example 5.6, where the index $j$ is a **forall** variable and $pre_j$, $post_j$ are formulas in which $j$ may occur. Assuming all mutable state is explicitly represented by variables, as in the encoding used by a verification tool (and by contrast with notations like separation logic or JML), the condition above can be written

$$\forall \overline{x} \cdot (\exists j \cdot pre_j) \Rightarrow \exists \overline{x'} \cdot (\forall j \cdot pre_j \Rightarrow post_j[\overline{x'}/\overline{x}])$$

where $\overline{x}$ spans the free variables other than $j$. The point is that satisfiability of a specification is reduced to validity of a closed formula.

Def. 5.4 gives a representation of two-state specifications in terms of general ones. The next result goes in the opposite direction. Thus, in terms of semantics, two-state specifications are equivalent in expressive power to general specifications. In practice, the distinction often makes a difference because specification languages are based on first order formulas over expressions in the programming language, sometimes augmented with mathematical data types, but not always including the ability to refer to semantic entities like complete states (as in Def. 5.4).

For any $(J, r, s)$ we define a two-state specification $(J, r, s)\dagger$ as follows:

$$(J, r, s)\dagger = (P, R) \text{ where } P = (\cup j \cdot r_j) \text{ and } R = \{(\sigma, \tau) \mid \forall j \cdot \sigma \in r_j \Rightarrow \tau \in s_j\} \tag{18}$$

Now $\dagger$ and $\langle\!\langle - \rangle\!\rangle$ are exact inverses in one direction, and up to equivalence in the other direction.

LEMMA 5.17. (a) For any specification in two-state form, $(Q, S)$, we have $\langle\!\langle Q, S \rangle\!\rangle\dagger = (Q, S)$. (b) For any $(J, pre, post)$ we have $\langle\!\langle\, (J, pre, post)\dagger \,\rangle\!\rangle \simeq (J, pre, post)$.

PROOF. For (a), straightforward use of the definitions yields the equality.

For (b), we certainly do not have equality. Let $(P, R)$ be given by (18). Let $(J', pre', post')$ be $\langle\!\langle P, R \rangle\!\rangle$ as given by Def. 5.4. So $J'$ is the set of all states (of appropriate type), quite possibly a much different set from $J$ which might even be finite. By definitions we have that for any state $\rho$,

$$pre'_\rho = \{\tau \mid \tau = \rho \wedge \exists j \cdot \rho \in pre_j\} \quad \text{and} \quad post'_\rho = \{\tau \mid \forall j \cdot \rho \in pre_j \Rightarrow \tau \in post_j\}$$

To show (b), consider any $\varphi$. We have $\varphi \models (J', pre', post')$ iff $\forall \rho, \sigma \cdot \sigma \in pre'_\rho \Rightarrow \varphi(\sigma) \in post'_\rho$, by definition of $\models$. This is equivalent to $\forall \sigma \cdot (\exists j \cdot \sigma \in pre_j) \Rightarrow (\forall j \cdot \sigma \in pre_j \Rightarrow \varphi(\sigma) \in post_j)$. Which is logically equivalent to $\forall \sigma, j \cdot \sigma \in pre_j \Rightarrow \varphi(\sigma) \in post_j$, i.e., $\varphi \models (J, pre, post)$. $\square$

### 5.4. Characterizing refinement

The most common formulation of behavioral subtyping uses the implications (3) and (4) in Sect. 2 that correspond to the rule of consequence in Hoare logic which, in effect, derives a weaker specification from a stronger one. Even in Hoare logic for simple procedures these implications are incomplete. Hoare proposed an "adaptation rule" which is not complete and some subsequent proposals were found to have subtle unsoundness in connection with specification variables (corrected in [America and de Boer 1990]). By now, sound and complete rules have been found and the connection with specification refinement has been made clear if not widely known [Olderog 1983; Kleymann 1999; Chen and Cheng 2000; Naumann 2001; Pierik 2006].

The following result characterizes refinement of general specifications in terms of pre- and post-conditions. Similar results appear in some of the work cited in preceding paragraphs but perhaps not easily connected with our formalism, and in some cases additional restrictions are imposed.

PROPOSITION 5.18 (CHARACTERIZATION OF REFINEMENT). Suppose that $(J, pre', post')$ and $(I, pre, post)$ are specifications of type $\Gamma \rightsquigarrow \Gamma'$. If $(J, pre', post')$ is satisfiable then the following are equivalent:

(a) $(J, pre', post') \sqsupseteq (I, pre, post)$
(b) $\forall i \in I, \sigma \in State(\Gamma) \cdot \sigma \in pre_i$
$\quad \Rightarrow (\exists j \in J \cdot \sigma \in pre'_j) \wedge (\forall \tau \in State(\Gamma') \cdot (\forall j \in J \cdot \sigma \in pre'_j \Rightarrow \tau \in post'_j) \Rightarrow \tau \in post_i)$

We refer to (b) as the ***characteristic formula*** for the refinement in (a).

The proof is a non-trivial exercise in predicate calculus and is given in Appendix A.3. It uses only the definitions and Lemma 5.16. Note that (b) implies (a) regardless of satisfiability, since an unsatisfiable specification refines any specification.

We get Eq. (17) by instantiating the Proposition with $J := I$, $pre' := pre\lfloor^* T$, $pre := pre\lfloor T$.

Using the definitions of $\sqsupseteq^T$ and $\sqsupseteq^{*T}$ we immediately get the following, which differs from Prop. 5.18 only in restricting $pre_i$ to states where self has type $T$.

COROLLARY 5.19 (CHARACTERIZATION OF REFINEMENT AT A SUBTYPE). Suppose $T \leq \Gamma\,\mathsf{self}$ and let $\Gamma_T = [\Gamma \mid \mathsf{self} : T]$. Suppose $(I, pre, post)$ is of type $\Gamma \rightsquigarrow \Gamma'$ and $(J, pre', post')$ is of type $\Gamma_T \rightsquigarrow \Gamma'$. If $(J, pre', post')$ is satisfiable then the following are equivalent:

(a) $(J, pre', post') \sqsupseteq^T (I, pre, post)$
(b) $\forall i \in I, \sigma \in State(\Gamma) \cdot \sigma \in pre_i\lfloor T$
$\Rightarrow (\exists j \in J \cdot \sigma \in pre'_j) \wedge (\forall \tau \in State(\Gamma') \cdot (\forall j \in J \cdot \sigma \in pre'_j \Rightarrow \tau \in post'_j) \Rightarrow \tau \in post_i)$

*Mutatis mutandis* with $\sqsupseteq^{*T}$ and $pre_i\lfloor^*$.

To handle specifications in two-state form (Def. 5.4), it is convenient to define a semantic version of the **old** operator: for any $\Gamma$ and $\Gamma'$, if $P$ is a subset of $State(\Gamma)$ then define $old(P) \subseteq (\Gamma) \times State(\Gamma')$ by $(\sigma, \tau) \in old(P) \iff \sigma \in P$.

COROLLARY 5.20 (CHARACTERIZATION OF REFINEMENT OF SPECIFICATIONS IN TWO-STATE FORM). Consider specifications $(P, R)$ and $(P', R')$ in two-state form, of type $\Gamma \rightsquigarrow \Gamma'$. Then $\langle\langle P', R' \rangle\rangle \sqsupseteq \langle\langle P, R \rangle\rangle$ iff

$$P \subseteq P' \;\wedge\; old(P) \cap R' \subseteq R \tag{19}$$

provided that $\langle\langle P', R' \rangle\rangle$ is satisfiable.

PROOF. Let $\langle\langle P, R \rangle\rangle$ be $(J, pre, post)$ and let $\langle\langle P', R' \rangle\rangle$ be $(J', pre', post')$. Recall from Def. 5.4 we have $J = J' = State(\Gamma)$. We calculate

$$\begin{array}{ll}
 & (J', pre', post') \sqsupseteq (J, pre, post) \\
\iff & \quad \text{Proposition 5.18, with } \sigma, \rho, \upsilon \text{ ranging over } State(\Gamma) \\
 & \forall \rho, \sigma \cdot \sigma \in pre_\rho \\
 & \quad \Rightarrow (\exists \upsilon \cdot \sigma \in pre'_\upsilon) \wedge (\forall \tau \in State(\Gamma') \cdot (\forall \upsilon \cdot \sigma \in pre'_\upsilon \Rightarrow \tau \in post'_\upsilon) \Rightarrow \tau \in post_\rho) \\
\iff & \quad \text{Def. 5.4 for } pre, post \text{ and for } pre', post' \\
 & \forall \rho, \sigma \cdot \rho = \sigma \wedge \sigma \in P \\
 & \quad \Rightarrow (\exists \upsilon \cdot \upsilon = \sigma \wedge \sigma \in P') \wedge (\forall \tau \cdot (\forall \upsilon \cdot \upsilon = \sigma \wedge \sigma \in P' \Rightarrow (\upsilon, \tau) \in R') \Rightarrow (\rho, \tau) \in R) \\
\iff & \quad \text{predicate calculus (one-point rule thrice)} \\
 & \forall \sigma \cdot \sigma \in P \Rightarrow \sigma \in P' \wedge (\forall \tau \cdot (\sigma \in P' \Rightarrow (\sigma, \tau) \in R') \Rightarrow (\sigma, \tau) \in R) \\
\iff & \quad \text{predicate calculus} \\
 & (\forall \sigma \cdot \sigma \in P \Rightarrow \sigma \in P') \wedge (\forall \sigma, \tau \cdot \sigma \in P \wedge (\sigma \in P' \Rightarrow (\sigma, \tau) \in R') \Rightarrow (\sigma, \tau) \in R) \\
\iff & \quad \text{predicate calculus: use } \sigma \in P \Rightarrow \sigma \in P'), \text{ and } x \wedge (y \Rightarrow z) \equiv x \wedge z \text{ when } x \Rightarrow y \\
 & (\forall \sigma \cdot \sigma \in P \Rightarrow \sigma \in P') \wedge (\forall \sigma, \tau \cdot \sigma \in P \wedge (\sigma, \tau) \in R' \Rightarrow (\sigma, \tau) \in R) \\
\iff & \quad \text{set theory, definition of } old(P) \\
 & P \subseteq P' \;\wedge\; old(P) \cap R' \subseteq R
\end{array}$$

□

Dhara and Leavens [1996] give a condition that is equivalent to (19) and similar to the join of specifications investigated later (see Lemma 9.7):

$$P \subseteq P' \;\wedge\; (\neg old(P') \cup R') \subseteq (\neg old(P) \cup R)$$

where we write $\neg$ for the complement with respect to the set of all states or state pairs of appropriate type. Note that $\neg old(P) = old(\neg P)$.

Taking the type of self into account we have the following, which refers to the following predicates:

$$exact^T = \{\sigma \mid selftype(\sigma) = T\} \qquad \text{and} \qquad is^T = \{\sigma \mid selftype(\sigma) \leq T\}$$

COROLLARY 5.21. Under the assumptions of Corollary 5.20 but with the types as in Corollary 5.19, we have $(J', pre', post') \sqsupseteq^{*T} (J, pre, post)$ iff

$$P \cap is^T \subseteq P' \ \wedge \ old(P \cap is^T) \cap R' \subseteq R$$

and $(J', pre', post') \sqsupseteq^{T} (J, pre, post)$ iff $P \cap exact^T \subseteq P' \ \wedge \ old(P \cap exact^T) \cap R' \subseteq R$.

The proofs are straightforward adaptations of the proof of the Corollary 5.20.

In terms of the Corollary, it is clear that if $P$ implies $exact^U$ for $U \neq T$ then the refinement holds trivially, because then $P \cap exact^T = \emptyset$. This is consistent with the observation following Lemma 5.12.

## 6. MODULAR CORRECTNESS AND MODULAR VERIFICATION

This section lays the groundwork for supertype abstraction, by formalizing the way modular verifiers and proof systems focus on one method at a time, relative to assumed specifications for all others.

We are concerned with a class table $CT$ together with a table of specifications for its methods. A ***specification table***, $ST$, is a function such that $ST(T, m)$ is a method specification of type $mtype(T, m)$ for each ref type $T$ and each $m \in Meths\ T$. Our primary use for specification tables is to model what might be called the "effective specification" against which an implementation at type $T$ would be verified. Such specifications are typically obtained from declared specifications by means of context-dependent interpretation of modifies clauses [Leino and Nelson 2002; Müller 2002], combinations of specification cases [Leavens et al. 2006; Wing 1983], unpacking of abstract predicates [Bierman and Parkinson 2005], specification inheritance (the topic of Sect. 9), invariant disciplines [Barnett et al. 2004; Müller et al. 2006], model fields (see Example 8.5), etc.

*Definition* 6.1 (*satisfaction by method environment*). Let $ST$ be a specification table. An extended method environment $\dot{\eta}$ ***satisfies*** $ST$, written $\dot{\eta} \models ST$, iff $\dot{\eta}(T, m) \models ST(T, m)$ for all ref types $T$ and $m \in Meths\ T$.

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

Note that $ST(K, m)$ is the entire proof obligation imposed on the implementation of $m$ in class $K$.

Let us consider satisfaction for the method environment $\widehat{\eta}$ denoted by a class table. If $m$ is inherited in $K$ from superclass $L$, the semantics $\widehat{\eta}(K, m)$ is defined to be $\widehat{\eta}(L, m)$ (or rather the restriction thereof to self of type $K$). This suggests that the implementation in $L$ must actually satisfy both $ST(K, m)$ and $ST(L, m)$. However, if $m$ is inherited in $K$ the programmer need not explicitly give a specification for $m$ in $K$. Instead, one can take $ST(K, m)$ to be $ST(L, m)\downarrow K$, so the implementation really has a single specification. (Unless there is some $I$ in $superinterfaces\ K$ that declares $m$ and is not in $superinterfaces\ L$.) This is pursued further in Sects. 9.2 and 9.3.

Modular verification statically checks that a command, such as a method body, satisfies its specification using the specifications of called methods. This is modular in that it does not use the code of called methods to verify calls to these methods. It introduces some inherent incompleteness, in that method specifications may be inadequate (just as declared loop invariants may be). But there is an important compensating advantage: verification need not be re-done when the code for called methods changes. Similarly, modular verification uses static type information to conservatively approximate what methods will be called by dynamic dispatch at runtime. This again introduces some inherent incompleteness, but again has a compensating advantage in that the verification need not be repeated when new subtypes are added to the program. Modular verification is widely used in tools, but only a few theoretical works formalize it semantically [Harel et al. 1977; O'Hearn et al. 2009].

The ultimate goal of verification is to establish correctness of a complete program, which for main program $C$ and specification $spec$ we write

$$\mathcal{D}[\![\Gamma \vdash C]\!](\widehat{\eta}) \models spec$$

where $\widehat{\eta}$ is the semantics of $CT$ (i.e., $\widehat{\eta} = \mathcal{D}[\![CT]\!]$). Modular verification establishes the stronger property that $C$ is correct in any correct environment. This property is formalized as follows.

*Definition* 6.2 (*modular correctness*). For command $\Gamma \vdash C$ and $\Gamma$-specification $spec$, we say $C$ ***modularly satisfies*** $spec$ ***w.r.t.*** $ST$, and write

$$ST, (\Gamma \vdash C) \models^{\mathcal{D}} spec$$

if and only if

$$\forall \eta \in \mathit{MethEnv} \cdot \eta \models ST \;\Rightarrow\; \mathcal{D}[\![\Gamma \vdash C]\!](\eta) \models \mathit{spec} \tag{20}$$

We also say $C$ is ***modularly correct***, when $ST$ and *spec* are understood.

For expression $\Gamma \vdash E : T$ and *spec* of type $\Gamma \rightsquigarrow [\mathsf{res} : T, \mathsf{exc} : \mathsf{Thr}]$ we define
$ST, (\Gamma \vdash E : T) \models^{\mathcal{D}} \mathit{spec}$ iff $\forall \eta \in \mathit{MethEnv} \cdot \eta \models ST \;\Rightarrow\; \mathcal{D}[\![\Gamma \vdash E : T]\!](\eta) \models \mathit{spec}$.

For brevity we shall sometimes let identifier $\mathcal{P}$ range over program ***phrases-in-context***, i.e., derivable typing judgments for commands and expressions. In that notation, $ST, \mathcal{P} \models^{\mathcal{D}} \mathit{spec}$ means $\forall \eta \in \mathit{MethEnv} \cdot \eta \models ST \;\Rightarrow\; \mathcal{D}[\![\mathcal{P}]\!](\eta) \models \mathit{spec}$. Despite use of the same font, $\mathcal{D}$ and $\mathcal{S}$ name fixed functions whereas $\mathcal{P}$ is a metavariable.

Condition (20) directly expresses that $C$ is correct under the hypothesis that its environment satisfies its specification $ST$. The straightforward syntactic counterpart would be a judgment of correctness under hypotheses, as formalized in variants of Hoare logic [Harel et al. 1977; Reynolds 1982; O'Hearn et al. 2009; Banerjee and Naumann 2013]. Whereas (20) refers to the actual program semantics, and thus directly expresses a desirable property, our interest is in reasoning based on static types, which we formalize by replacing $\mathcal{D}$ in Def. 6.2 by $\mathcal{S}$, as follows.

*Definition* 6.3 (*modular correct under static dispatch*). Phrase-in-context $\mathcal{P}$ ***modularly satisfies*** *spec* ***w.r.t.*** $ST$ ***under static dispatch***, written

$$ST, \mathcal{P} \models^{\mathcal{S}} \mathit{spec}$$

if and only if

$$\forall \eta \in \mathit{XMethEnv} \cdot \eta \models ST \;\Rightarrow\; \mathcal{S}[\![\mathcal{P}]\!](\eta) \models \mathit{spec}$$

We also say $\mathcal{P}$ is ***modularly correct under static dispatch***.

Müller [2002] (p111) uses the term "modular correctness" for a similar notion, but defined in terms of proofs rather than semantics.[15]

Let us exercise the definitions by considering modular verification of a complete program. In some way or other, a logic or verification tool establishes (20) for each method body, i.e., checks correctness of each method body $C^{K,m}$ with respect to its specification:

$$ST, (\Gamma \vdash C^{K,m}) \models ST(K, m) \tag{21}$$

where $m$ is a method with implementation $C^{K,m}$ declared in class $K$ and $\Gamma$ declares the parameter and $\mathsf{res}$. From this one hopes to obtain that $\widehat{\eta} \models ST$ (where $\widehat{\eta} = \mathcal{D}[\![CT]\!]$); because then from $ST, (\Gamma^{main} \vdash C^{main}) \models \mathit{spec}$ it follows immediately that $\mathcal{D}[\![\Gamma^{main} \vdash C^{main}]\!](\widehat{\eta}) \models \mathit{spec}$. The hope is easily realized for partial correctness specifications. Recall the approximation chain $\eta$ in Sect. 4.6 from which $\widehat{\eta}$ is obtained. For partial correctness, $\eta_0$ satisfies any $ST$. Using (21), an easy induction would then yield that $\eta_i$ satisfies $ST$, in the sense of partial correctness, for all $i$. So we would get that $\widehat{\eta}$ satisfies $ST$ for partial correctness.

For total correctness, $\eta_0$ does not satisfy a non-trivial $ST$ and a proof of $\widehat{\eta} \models ST$ must involve a well-foundedness argument. Most verification systems for OO programs do partial correctness, perhaps in part because a general formulation of well foundedness arguments is complicated in the presence of mutually recursive methods and dynamic dispatch. In the main part of this paper we deal with total correctness (but see Sect. 10). However, for our purposes there is no need to formalize how $\widehat{\eta} \models ST$ is established. Our concern is with modular reasoning to establish modular correctness, i.e., $ST, \mathcal{P} \models^{\mathcal{D}} \mathit{spec}$ for arbitrary $\mathcal{P}$ and *spec*.

One benefit of modular correctness is immediate from Def. 6.2: a verified command need not be re-verified when changes are made to the implementations of methods called in $C$. Another benefit is to decompose the verification task into potentially more tractable subtasks, but that is outside the scope of this paper. Modular correctness avoids the need for re-verification when the class table is extended with new classes and interfaces, provided that specifications of new methods have behavioral subtyping. We explore this topic further in Sect. 9.3.

[15] The term "modular correctness" seems more memorable than "modular satisfaction", but the latter is accompanied by the nice verb "satisfy".

With the preceding definitions, we can introduce our first notion of supertype abstraction, which says that

$$ST, \mathcal{P} \models^{\mathcal{S}} spec \quad \text{implies} \quad ST, \mathcal{P} \models^{\mathcal{D}} spec \tag{22}$$

(It will be restated later as Def. 8.7.) When it holds, this implication lets us prove modular correctness by hypothetical reasoning using static dispatch semantics.

Reasoning under hypotheses is well suited to interactive proof assistants. For more automatic verification tools (and also in some theoretical works [O'Hearn et al. 2009; Banerjee and Naumann 2013]), it is preferable to avoid the quantification over environments in Definition 6.3. Instead, reasoning is done more directly in terms of the specifications in $ST$, which yields the consequent in (20) without the need to consider all implementations of $ST$. In a system based on verification conditions, those conditions are expressed in terms of axiomatic semantics, ultimately resting on predicate transformers. In particular, an invocation of $m$ at static type $T$ is interpreted using assertions and assumptions, in what amounts to a "specification statement" where the specification is $ST(T, m)$. Such statements can be interpreted as weakest precondition predicate transformers [Morgan 1994; Back and von Wright 1998]. We shall define a method environment $\{\!|ST|\!\}$ comprised of such predicate transformers —it can be seen as the least refined environment that satisfies $ST$. We also define a static dispatch predicate transformer semantics $\mathcal{S}\{\!|-|\!\}$ so that $\mathcal{S}\{\!|\Gamma \vdash C|\!\}(\{\!|ST|\!\})$ is the predicate transformer denoted by $C$ in the single environment $\{\!|ST|\!\}$. The technical details of predicate transformer semantics are the topic of Sect. 7. Our semantic formulation of modular verification exploits the fact that satisfaction of a specification can be encoded by refinement of predicate transformers (see Eq. (27) and Def. 7.3 in Sect. 7.3).

*Definition* 6.4 (*modular verification*). For command $\Gamma \vdash C$ and $\Gamma$-specification *spec*, we say $C$ is ***modularly verified for*** *spec* ***w.r.t.*** $ST$, if and only if

$$\mathcal{S}\{\!|\Gamma \vdash C|\!\}(\{\!|ST|\!\}) \sqsupseteq \{\!|spec|\!\} \tag{23}$$

That is, *spec* is satisfied by $C$ under static dispatch and in the least refined environment that satisfies $ST$. Modular verification of an expression $E$ is defined similarly.

This brings us to our second notion of supertype abstraction (restated later as Def. 8.6):

$$\mathcal{S}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\}) \sqsupseteq \{\!|spec|\!\} \quad \text{implies} \quad ST, \mathcal{P} \models^{\mathcal{D}} spec \tag{24}$$

The main result of the paper says that the following are equivalent:

(a) $ST$ has behavioral subtyping
(b) modular correctness under static dispatch implies modular correctness (cf. Eq. (22))
(c) modular verification implies modular correctness (cf. Eq. (24))

This is Thm. 8.15 in Sect. 8. In preparation for that, we assemble the ingredients of Def. 6.4 in Sect. 7. Here is an informal depiction of some key notions and implications between them that are formalized in due course.

$$\begin{array}{ccc} \mathcal{S}\{\!|\Gamma \vdash C|\!\}(\{\!|ST|\!\}) \sqsupseteq \{\!|spec|\!\} & \xRightarrow{\textit{Thm. } 8.8(a)} & ST, \mathcal{P} \models^{\mathcal{S}} spec \\ \Big\Downarrow \textit{Thm. } 8.10^{+} & \searrow^{\textit{Def. } 8.6^{+}} & \Big\Downarrow \textit{Def. } 8.7^{+} \\ \mathcal{D}\{\!|\Gamma \vdash C|\!\}(\{\!|ST|\!\}) \sqsupseteq \{\!|spec|\!\} & \xRightarrow{\textit{Thm. } 8.8(b)} & ST, \mathcal{P} \models^{\mathcal{D}} spec \end{array} \tag{25}$$

The superscript $^{+}$ indicates implications that depend on behavioral subtyping.

*Remark* 6.5. Leino [1995], and Leino and Nelson [2002], explore a notion they call ***modular soundness***: proofs remain valid when a program is extended with additional components. Their work explores this in connection with techniques for data abstraction and information hiding. Müller [2002] (p112) connects modular soundness with behavioral subtyping as well, in the setting of a proof system. To deal with 'open programs' that may be extended, his formalization of modular soundness is expressed in a sort of Kripke semantics that identifies an open program with the set of all closed programs that extend it.

We would like to consider a class table $CT'$ that extends $CT$ with additional classes and interfaces, together with a corresponding $ST'$ that extends $ST$. The expectation is that $ST, \mathcal{P} \models^{\mathcal{D}} spec$ should

imply $ST', \mathcal{P} \models^{\mathcal{D}} \mathit{spec}$ provided that $ST'$ has behavioral subtyping. This is not easy to make precise in our setting, because *spec* is a semantic object defined in terms of states for $CT$, whereas states for $CT'$ may contain objects of the added types. The denotation of $\mathcal{P}$ may also differ between $CT$ and $CT'$. A Kripke-style formulation is possible but beyond the scope of this paper. □

The notion of modular correctness is defined in terms of a specific programming language. We have gone to some lengths to formalize a fairly powerful language. Still, one might wonder whether the main result holds for trivial reasons that have nothing to do with the language. To dispel that doubt, this section concludes by describing a language construct for which behavioral subtyping does not imply supertype abstraction. It breaks supertype abstraction by distinguishing between the subtype relation on which static reasoning is based and a runtime subtype relation on which dynamic dispatch is based.

*Example* 6.6. Consider an extension of the programming language, in which the subtype relation can be extended during computations. The idea is that the subtype relation, $\leq$, is part of the program state and can be changed by the primitive command **make-subtype** $K$ **of** $I$ which makes class $K$ a subtype of interface $I$. Initially, the relation $\leq$ is the one determined by the class table. The effect of **make-subtype** is not only to add the pair $(K, I)$ to relation $\leq$ but also to add the other pairs needed to maintain that $\leq$ is transitive.

This idea can be carried out in several ways. For our purpose here, the idea is to change little else about the language and its semantics. For example, typechecking does not track additions to $\leq$. So the following is well formed, but would not be without the cast:

$$\textbf{var } x : I \textbf{ in } \textbf{make-subtype } K \textbf{ of } I;\ x := (I) \textbf{ new } K();\ x.m()$$

The cast succeeds, as its semantics is now based on the current state of $\leq$, as is method dispatch.

Without changing existing typing rules we retain type soundness as embodied in the denotational semantics. In the semantics of Sect. 4, the set of values of a given reference type $T$ depends on state, because it contains only allocated references. To cater for **make-subtype**, the set of values of type $T$ is also dependent on the current subtype relation. Because we are not considering a construct that undoes subtype relations, the value set for a type $T$ only grows. For typing of **make-subtype** $K$ **of** $I$, one option is to require that $K$ provides (i.e., declares or inherits) all methods declared in $I$ — essentially, $K$ satisfies the conditions needed for it to declare "**implements** $I$". Another alternative is for **make-subtype** $K$ **of** $I$ to be allowed but throw an exception in case $K$ lacks, or has an incompatible signature for, some method of $I$. (The aim is to avoid "method not found".)

The point of this example is that, after making $K$ a subtype of $I$, the call $x.m()$ may dispatch to an implementation of $m$ in $K$. We consider that behavioral subtyping is defined in terms of the initial subtype relation determined by the class table. Assuming that initially $K \not\leq I$, the implementation of $m$ in $K$ need not satisfy the specification of $m$ in $I$, because $ST(K, m)$ is not required to refine $ST(I, m)$. Thus supertype abstraction is unsound for reasoning about the call $x.m()$ above.

Indeed, if we considered behavioral subtyping to be a state-dependent notion, then **make-subtype** $K$ **of** $I$ would appear to break behavioral subtyping. But that strays rather far from the topic of moular reasoning. We interpret the example as a strange language for which behavioral subtyping does not imply supertype abstraction.

Readers who find this language construct implausible are advised to become familiar with the JavaScript language, in which inheritance is based on prototype chains –and in which the prototype relation is mutable.

## 7. PREDICATE TRANSFORMER SEMANTICS

This section uses weakest preconditions to derive a predicate transformer semantics $\{\![-]\!\}$ from the state transformer semantics $[\![-]\!]$. Rather, both static- and dynamic-dispatch predicate transformer semantics, written $\mathcal{S}\{\![-]\!\}$ and $\mathcal{D}\{\![-]\!\}$, are derived from the corresponding state transformer semantics $\mathcal{S}[\![-]\!]$ and $\mathcal{D}[\![-]\!]$ (Sect. 7.2). The predicate transformer semantics applies to method environments containing predicate transformers. Predicate transformer semantics is also defined for specifications, so that refinement of specifications corresponds to refinement of predicate transformers. The semantics of specifications gives a canonical method environment determined by the specification table, written $\{\![ST]\!\}$ (Sect. 7.3). We use $\mathcal{S}\{\![-]\!\}$ and $\{\![ST]\!\}$ to formalize supertype abstraction. We use $\mathcal{D}\{\![-]\!\}$ in proofs. Groundwork is laid in Sect. 7.1.

$$
\begin{array}{lll}
(r,h,s) \in \{\!|\Gamma \vdash x : T|\!\}(\theta)(Q) & \iff & (r,h,[\mathsf{res} : s\,x, \mathsf{exc} : \mathit{null}]) \in Q \\
(r,h,s) \in \mathcal{S}\{\!|\Gamma \vdash x.m(\overline{y}) : U|\!\}(\theta)(Q) & \iff & \textbf{if } s\,x = \mathit{null} \textbf{ then } \mathit{except}(r,h,U,\mathsf{NullDeref}) \in Q \\
 & & \textbf{else let } T = \Gamma\,x \textbf{ in let } \overline{z} = \mathit{formals}(T,m) \textbf{ in} \\
 & & \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } (r,h,s_1) \in \theta(T,m)(Q) \\
(r,h,s) \in \mathcal{D}\{\!|\Gamma \vdash x.m(\overline{y}) : U|\!\}(\theta)(Q) & \iff & \textbf{if } s\,x = \mathit{null} \textbf{ then } \mathit{except}(r,h,U,\mathsf{NullDeref}) \in Q \\
 & & \textbf{else let } K = r(s\,x) \textbf{ in let } \overline{z} = \mathit{formals}(K,m) \textbf{ in} \\
 & & \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } (r,h,s_1) \in \theta(K,m)(Q)
\end{array}
$$

Fig. 6. Predicate transformer semantics: selected cases. Here $\theta$ ranges over predicate transformer method environments, $(r,h,s)$ over states, and $Q$ over sets of states. Read $\{\!|-|\!\}$ in the first clause as either $\mathcal{S}\{\!|-|\!\}$ or $\mathcal{D}\{\!|-|\!\}$. The remaining cases are in Fig. 14 in Appendix Sect. A.6.

### 7.1. Predicate transformers and weakest preconditions

A ***predicate transformer of type*** $\Gamma \rightsquigarrow \Gamma'$ is a function from $\wp(\mathit{State}(\Gamma'))$ to $\wp(\mathit{State}(\Gamma))$ that is monotonic with respect to set inclusion ($\subseteq$). The reversal of direction is because we use predicate transformers for weakest preconditions.

Suppose $\varphi$ is a state transformer of type $\Gamma \rightsquigarrow \Gamma'$ and *post* is a subset of $\mathit{State}(\Gamma')$. The ***weakest precondition*** of $\varphi$ with respect to *post*, written $wp(\varphi)(\mathit{post})$, is the subset of $\mathit{State}(\Gamma)$ defined by

$$wp(\varphi)(\mathit{post}) = \{\sigma \in \mathit{State}(\Gamma) \mid \varphi(\sigma) \in \mathit{post}\}. \tag{26}$$

Note that $wp(\varphi)$ captures total correctness, because predicates are sets of states and $\perp$ is not a state.

For predicate transformers $f, g$ of the same type, we define $f \sqsupseteq g$ —read "$f$ ***refines*** $g$"— by

$$f \sqsupseteq g \iff \forall \mathit{post} \cdot f(\mathit{post}) \supseteq g(\mathit{post}). \tag{27}$$

### 7.2. Deriving the semantics of commands and expressions

Our first step is to lift method environments from state transformers to predicate transformers. For any normal or extended method environment (of state transformers) $\eta$, we define $wp(\eta)$ to be the corresponding method environment given by

$$wp(\eta)(T,m) = wp(\eta(T,m)) \quad \text{for all } T, m \tag{28}$$

Both static and dynamic dispatch predicate transformer semantics are derived directly from the state transformer semantics, in the course of proving the following Lemma. It says that the predicate transformer definitions exactly correspond to taking the *wp* of the state transformer semantics, when the method environment is obtained using *wp*. Selected cases in the semantics are given in Fig. 6 and derived in the proof. Aside from the method call case, we have no need in this paper to work directly with the predicate transformer semantic definitions, so the rest are relegated to the Appendix (Fig. 14 in Sect. A.6). Here and in the sequel we let $\theta$ range over method environments that contain predicate transformers.

Lemma 7.1 (wp-equivalence). Let $\eta$ be any method environment. For any phrase-in-context $\mathcal{P}$, we have

$$wp(\mathcal{D}[\![\mathcal{P}]\!](\eta)) = \mathcal{D}\{\!|\mathcal{P}|\!\}(wp(\eta))$$

and the same for static dispatch semantics: $wp(\mathcal{S}[\![\mathcal{P}]\!](\dot{\eta})) = \mathcal{S}\{\!|\mathcal{P}|\!\}(wp(\dot{\eta}))$ where $\dot{\eta}$ ranges over extended method environments.

Proof. By structural induction on expressions and then by structural induction on commands using the result for expressions. In cases other than method call, the dynamic- and static-dispatch cases are the same so we prove both at once, omitting $\mathcal{D}$ and $\mathcal{S}$ from the notation. We reason in terms of an arbitrary method environment $\eta$ (normal or extended or both, as appropriate).

The first expression case is $x : T$. We have for any $(r, h, s)$ of type $\Gamma$ and any predicate $Q$ of type $[\mathsf{res} : T, \mathsf{exc} : \mathsf{Thr}]$

$$\begin{array}{ll} & (r,h,s) \in wp([\![\Gamma \vdash x : T]\!](\eta))(Q) \\ \iff & \quad \text{definition (26) of } wp \\ & [\![\Gamma \vdash x : T]\!](\eta)(r,h,s) \in Q \\ \iff & \quad \text{definition of } [\![\Gamma \vdash x : T]\!] \\ & (r,h,[\mathsf{res} : s\,x, \mathsf{exc} : \mathit{null}]) \in Q \\ \iff & \quad \text{definition of } \{\![\Gamma \vdash x : T]\!\}, \text{ see below} \\ & (r,h,s) \in \{\![\Gamma \vdash x : T]\!\}(wp(\eta))(Q) \end{array}$$

The last step suggests the semantics, which is given in the first line of Fig. 6 and which applies to any environment $\theta$.

For static dispatch semantics of method call we have

$$\begin{array}{ll} & (r,h,s) \in wp(\mathcal{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\dot\eta))(Q) \\ \iff & \quad \text{definition (26) of } wp \\ & \mathcal{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\dot\eta)(r,h,s) \in Q \\ \iff & \quad \text{definition of } \mathcal{S}[\![-]\!] \\ & (\textbf{if } s\,x = \mathit{null} \textbf{ then } \mathit{except}(r,h,U,\mathsf{NullDeref}) \\ & \textbf{else let } T = \Gamma\,x \textbf{ in let } \overline{z} = \mathit{formals}(T,m) \textbf{ in} \\ & \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } \dot\eta(T,m)(r,h,s_1)) \quad \in Q \\ \iff & \quad \text{logic} \\ & \textbf{if } s\,x = \mathit{null} \textbf{ then } \mathit{except}(r,h,U,\mathsf{NullDeref}) \in Q \\ & \textbf{else let } T = \Gamma\,x \textbf{ in let } \overline{z} = \mathit{formals}(T,m) \textbf{ in} \\ & \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } \dot\eta(T,m)(r,h,s_1) \in Q \\ \iff & \quad \text{definition of } wp \\ & \textbf{if } s\,x = \mathit{null} \textbf{ then } \mathit{except}(r,h,U,\mathsf{NullDeref}) \in Q \\ & \textbf{else let } T = \Gamma\,x \textbf{ in let } \overline{z} = \mathit{formals}(T,m) \textbf{ in} \\ & \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } (r,h,s_1) \in wp(\dot\eta(T,m))(Q) \\ \iff & \quad \text{definition of } wp(\dot\eta) \\ & \textbf{if } s\,x = \mathit{null} \textbf{ then } \mathit{except}(r,h,U,\mathsf{NullDeref}) \in Q \\ & \textbf{else let } T = \Gamma\,x \textbf{ in let } \overline{z} = \mathit{formals}(T,m) \textbf{ in} \\ & \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } (r,h,s_1) \in wp(\dot\eta)(T,m)(Q) \\ \iff & \quad \text{definition of } \mathcal{S}\{\![-]\!\} \\ & (r,h,s) \in \mathcal{S}\{\![\Gamma \vdash x.m(\overline{y}) : U]\!\}(wp(\dot\eta))(Q) \end{array}$$

Again, the definition is chosen to justify the last step (see Fig. 6).

For dynamic dispatch we have a similar calculation.

$$\begin{array}{ll} & (r,h,s) \in wp(\mathcal{D}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\eta))(Q) \\ \iff & \quad \text{definition (26) of } wp \\ & \mathcal{D}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\eta)(r,h,s) \in Q \\ \iff & \quad \text{definition of } \mathcal{D}[\![-]\!] \\ & (\textbf{if } s\,x = \mathit{null} \textbf{ then } \mathit{except}(r,h,U,\mathsf{NullDeref}) \\ & \textbf{else let } K = r(s\,x) \textbf{ in let } \overline{z} = \mathit{formals}(K,m) \textbf{ in} \\ & \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } \eta(K,m)(r,h,s_1)) \quad \in Q \\ \iff & \quad \text{logic} \\ & \textbf{if } s\,x = \mathit{null} \textbf{ then } \mathit{except}(r,h,U,\mathsf{NullDeref}) \in Q \\ & \textbf{else let } K = r(s\,x) \textbf{ in let } \overline{z} = \mathit{formals}(K,m) \textbf{ in} \\ & \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } \eta(K,m)(r,h,s_1) \in Q \\ \iff & \quad \text{definitions of } wp(\eta(K,m)) \text{ and of } wp(\eta) \\ & \textbf{if } s\,x = \mathit{null} \textbf{ then } \mathit{except}(r,h,U,\mathsf{NullDeref}) \in Q \\ & \textbf{else let } K = r(s\,x) \textbf{ in let } \overline{z} = \mathit{formals}(K,m) \textbf{ in} \\ & \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } (r,h,s_1) \in wp(\eta)(K,m)(Q) \\ \iff & \quad \text{definition of } \mathcal{D}\{\![-]\!\} \text{ (see Fig. 6)} \\ & (r,h,s) \in \mathcal{D}\{\![\Gamma \vdash x.m(\overline{y}) : U]\!\}(wp(\eta))(Q) \end{array}$$

Note that this only applies the method environment at class types, not interfaces. □

The cases given above do not illustrate the whole story. The semantics in Sect. 4 is written in a way intended to convince the reader it is operationally accurate with respect to Java-like languages, modulo the idealizations: unbounded integers and unbounded heap and stack size. To facilitate the derivation of predicate transformer semantics for control constructs, **var**, etc., we refactor the semantic definition from Sect. 4.5 using a little algebra of state transformers. The details are in Sect. A.4.

A straightforward part of the proof of Lemma 7.1 is to show that for any $\Gamma \vdash C$ and any $\theta$, both $\mathcal{S}\{\![\Gamma \vdash C]\!\}(\theta)(Q)$ and $\mathcal{D}\{\![\Gamma \vdash C]\!\}(\theta)(Q)$ are monotonic in $Q$, with respect to inclusion of predicates, and hence are predicate transformers.

We order method environments pointwise: we define $\theta \sqsupseteq \theta'$ iff $\theta(T, m) \sqsupseteq \theta'(T, m)$ for all $T, m$. Here $T$ ranges over classes, or over all types, depending on whether normal or extended environments are under consideration.

LEMMA 7.2. For any $\mathcal{P}$, both $\mathcal{S}\{\![\mathcal{P}]\!\}(\theta)$ and $\mathcal{D}\{\![\mathcal{P}]\!\}(\theta)$ are monotonic in $\theta$.

The proof is a straightforward but tedious induction on $\mathcal{P}$, checking that the various constructs are monotonic in their constituent parts and checking that the static and dynamic semantics of method call are monotonic in the environment.

### 7.3. Predicate transformer semantics of specifications and refinement

We have no need to embed specifications in programs as "specification statements" as in refinement calculi [Back 1988; Morgan 1988; Morgan 1994]. However, we do need predicate transformer semantics of specifications —in fact, a semantics slightly different from what might be expected. Here there is no need to distinguish between static and dynamic dispatch, nor are there constituent phrases that depend on a method environment.

*Definition* 7.3. Let $(J, pre, post)$ be a specification of type $\Gamma \rightsquigarrow \Gamma'$. We write $\{\![J, pre, post]\!\}$ for the predicate transformer of the same type, defined for any $\sigma$ and $Q$ by

$$\sigma \in \{\![J, pre, post]\!\}(Q) \iff (\exists j \cdot \sigma \in pre_j) \wedge (\forall \tau \cdot (\forall i \cdot \sigma \in pre_i \Rightarrow \tau \in post_i) \Rightarrow \tau \in Q)$$

Here $j$ and $i$ both range over $J$ and $\tau$ ranges over $\Gamma'$-states.

*Remark* 7.4. One might attempt to use a simpler definition, which corresponds to the semantics of specification statements of Morgan [1988] (where the term "logical constant" is used for what we formalize as indices in the notion of general specification). It has the following form, where we use the notation $\langle\!|J, pre, post|\!\rangle$ for later reference:

$$\sigma \in \langle\!|J, pre, post|\!\rangle(Q) \iff \exists j \cdot \sigma \in pre_j \wedge post_j \subseteq Q$$

This has some but not all of the properties we need, as detailed later in Remark 7.8. It corresponds to an incomplete characterization of refinement for specifications.[16] In the case of two-state postconditions, the two kinds of specification statements are equivalent.

Some readers may find an operational reading helpful: Morgan's specification statement angelically chooses, for a given initial state $\sigma$, some $j$ for which the postcondition can be ensured. Def. 7.3 treats the choice of $j$ demonically, cf. Back and von Wright [1998]. □

In the case that $(J, pre, post)$ is unsatisfiable, there is at least one $\sigma$ in $\{\![J, pre, post]\!\}(\emptyset)$, as a consequence of the definition and Lemma 5.16. That is, $\{\![J, pre, post]\!\}$ is $\emptyset$-strict iff $(J, pre, post)$ is satisfiable. It is also straightforward to show that $\{\![J, pre, post]\!\}$ is positively conjunctive (regardless of satisfiability).

LEMMA 7.5 (REFINEMENT AND SATISFACTION). Let $\varphi$ be a state transformer and *spec* a specification of the same type as $\varphi$. Then $\varphi \models spec$ iff $wp(\varphi) \sqsupseteq \{\![spec]\!\}$.

PROOF. Consider any $\varphi$ and specification $(J, pre, post)$ of the same type. By (27), the definition of refinement, $wp(\varphi) \sqsupseteq \{\![J, pre, post]\!\}$ is equivalent to

$$\forall \sigma, Q \cdot \sigma \in wp(\varphi)(Q) \Leftarrow \sigma \in \{\![J, pre, post]\!\}(Q)$$

[16] For Morgan's refinement calculus this definition is suitable, because specification statements are embedded in programs. Thus programs can denote arbitrary predicate transformers, whereas our programs denote only those satisfying the "healthiness conditions" of Dijkstra [1976].

with $Q$ ranging over state sets. This in turn is equivalent, by definition (26) of *wp* and Def. 7.3, to

$$\forall \sigma, Q \cdot \varphi(\sigma) \in Q \Leftarrow (\exists j \cdot \sigma \in pre_j) \wedge (\forall \tau \cdot (\forall i \cdot \sigma \in pre_i \Rightarrow \tau \in post_i) \Rightarrow \tau \in Q) \tag{29}$$

where both $i$ and $j$ range over $J$ and $\sigma, \tau$ range over states of appropriate type. By definition, $\varphi \models (J, pre, post)$ is equivalent to

$$\forall k, \sigma \cdot \sigma \in pre_k \Rightarrow \varphi(\sigma) \in post_k \tag{30}$$

We complete the proof by mutual implication.

Assume (29). To show (30), observe that for any $k$ in $J$ and any $\sigma$ in $pre_k$ we can instantiate (29) with $post_k$ for $Q$ (and $\sigma$ for $\sigma$) to obtain

$$\varphi(\sigma) \in post_k \Leftarrow (\exists j \cdot \sigma \in pre_j) \wedge (\forall \tau \cdot (\forall i \cdot \sigma \in pre_i \Rightarrow \tau \in post_i) \Rightarrow \tau \in post_k) \tag{31}$$

so to prove $\varphi(\sigma) \in post_k$ it is enough to establish the conjuncts of the antecedent. From $\sigma \in pre_k$ we get $\exists j \cdot \sigma \in pre_j$. For the second conjunct we observe for any $\tau$ that

$$\begin{array}{ll} & \forall i \cdot \sigma \in pre_i \Rightarrow \tau \in post_i \\ \Rightarrow & \quad \text{logic (instantiate } i := k) \\ & \sigma \in pre_k \Rightarrow \tau \in post_k \\ \Rightarrow & \quad \text{using } \sigma \in pre_k \\ & \tau \in post_k \end{array}$$

which concludes the proof of (30).

Now assume (30). To show (29), consider any $\sigma, Q$. Assume the antecedent of (29). The conjunct $\exists j \cdot \sigma \in pre_j$ lets us choose some $j$ such that $\sigma \in pre_j$, and then by (30) we get $\varphi(\sigma) \in post_j$ —which implies that $\varphi(\sigma)$ is not $\bot$. Thus we can instantiate $\tau$ in the second conjunct by the state $\varphi(\sigma)$, to obtain

$$(\forall i \cdot \sigma \in pre_i \Rightarrow \varphi(\sigma) \in post_i) \Rightarrow \varphi(\sigma) \in Q$$

The antecedent of this formula is a direct consequence of (30) so we obtain the consequent $\varphi(\sigma) \in Q$ which concludes the proof of (29). □

It is standard that the set of all predicate transformers of a fixed type, say $\Gamma \rightsquigarrow \Gamma'$, is a complete lattice with meets (resp. joins) given by pointwise intersection (resp. union). In detail, for any set $X$ of predicate transformers, all of type $\Gamma \rightsquigarrow \Gamma'$, we write $(\dot{\sqcap} f \mid f \in X \cdot f)$ for the ***pointwise meet***, i.e., the predicate transformer defined by

$$(\dot{\sqcap} f \mid f \in X \cdot f)(Q) = (\cap f \mid f \in X \cdot f(Q)).$$

The symbol $\dot{\sqcap}$ is intended as a reminder that this is a defined operation. However, it yields ***meets*** (greatest lower bounds) in the lattice of all predicate transformers. That is, for any predicate transformer $g$ and set $X$ of predicate transformers,

$$(\forall f \cdot f \in X \Rightarrow f \sqsupseteq g) \iff (\dot{\sqcap} f \mid f \in X \cdot f) \sqsupseteq g. \tag{32}$$

The following can be proved directly from the definitions (and is, in Sect. A.7).

LEMMA 7.6 (MEET CHARACTERIZATION). For any satisfiable specification $(J, pre, post)$ we have

$$\{\![J, pre, post]\!\} = (\dot{\sqcap} \varphi \mid \varphi \models (J, pre, post) \cdot wp(\varphi)) \tag{33}$$

Satisfiability is necessary for Lemma 7.6. In the case of an unsatisfiable specification $(J, pre, post)$, we have that $(\dot{\sqcap} \varphi \mid \varphi \models (J, pre, post) \cdot wp(\varphi))$ is the function $\lambda Q \cdot true$ sending $Q$ to the empty intersection, *true*. But a state $\sigma$ is in $\{\![J, pre, post]\!\}(Q)$ only if either it is an initial state at which the specification is unsatisfiable (i.e., $\sigma$ is in some $pre_j$ but the postcondition is over-constrained), or the specification allows it but ensures a final state in $Q$.

The next result justifies the use of symbol $\sqsupseteq$ for both specifications and predicate transformers.

LEMMA 7.7. For any $spec_0$ and $spec_1$ of some type $\Gamma \rightsquigarrow \Gamma'$, with $spec_1$ satisfiable, we have $spec_1 \sqsupseteq spec_0$ iff $\{\![spec_1]\!\} \sqsupseteq \{\![spec_0]\!\}$.

PROOF.

$$
\begin{array}{ll}
 & spec_1 \sqsupseteq spec_0 \\
\iff & \quad \text{definition of } \sqsupseteq \text{ for specifications} \\
 & \forall \varphi \cdot \varphi \models spec_1 \;\Rightarrow\; \varphi \models spec_0 \\
\iff & \quad \text{Lemma 7.5} \\
 & \forall \varphi \cdot \varphi \models spec_1 \;\Rightarrow\; wp(\varphi) \sqsupseteq \{\!|spec_0|\!\} \\
\iff & \quad \text{meet property} \\
 & (\dot{\sqcap}\varphi \mid \varphi \models spec_1 \cdot wp(\varphi)) \sqsupseteq \{\!|spec_0|\!\} \\
\iff & \quad \text{Lemma 7.6, } spec_1 \text{ satisfiable} \\
 & \{\!|spec_1|\!\} \sqsupseteq \{\!|spec_0|\!\}
\end{array}
$$

□

In case $spec_1$ is unsatisfiable, we have $spec_1 \sqsupseteq spec_0$ for any $spec_0$, but $\{\!|spec_1|\!\} \sqsupseteq \{\!|spec_0|\!\}$ only if $spec_0$ has no more unsatisfiable initial states than $spec_1$ does.

*Remark* 7.8. Lemma 7.5 holds with Morgan's semantics $\langle\!|spec|\!\rangle$ in place of $\{\!|spec|\!\}$ (see Remark 7.4). But Lemmas 7.6 and 7.7 fail for $\langle\!|spec|\!\rangle$. The refinement $\langle\!|spec|\!\rangle \sqsubseteq (\dot{\sqcap}\varphi \mid \varphi \models spec \cdot wp(\varphi))$ does hold, for the same reasons as in the preceding proof. So we have $\{\!|spec|\!\} \sqsupseteq \langle\!|spec|\!\rangle$ for any satisfiable specification. However, this refinement is an inequality in general. For an example, consider the specification $(\mathbb{Z}, pre, post)$ of type $[r : \mathsf{real}] \rightsquigarrow [k : \mathsf{int}]$ where for any integer $i$ we define

$$pre_i = \{\sigma \mid i \le \sigma(r) \le i + 1\} \quad \text{and} \quad post_i = \{\tau \mid i = \tau(k) \vee i + 1 = \tau(k)\}$$

This has the peculiar feature that for an initial state where $r$ has an integral value, $k$ must take that value. For example, suppose $\sigma_1(r) = 1$, then $\sigma_1$ is in both $pre_0$ and $pre_1$. Any satisfying state transformer must establish $(0 = k \vee 1 = k) \wedge (1 = k \vee 2 = k)$. Taking $Q_1$ to be the states where $k = 1$, we have $\sigma_1 \in \{\!|\mathbb{Z}, pre, post|\!\}(Q_1)$ but $\sigma_1 \notin \langle\!|\mathbb{Z}, pre, post|\!\rangle(Q_1)$.

For another example of this kind, see the discussion of Fig. 8 in Sect. 9.1. □

The following corresponds to $\sqsupseteq^T$ in Def. 5.10 of refinement of specifications at a subtype. As it happens, we do not need the analogous notion for downward subtypes (cf. $\sqsupseteq^{*T}$ in Def. 5.10).

*Definition* 7.9. Let $f$ and $g$ be predicate transformers such that $f$ has type $\Gamma \rightsquigarrow \Gamma'$ and $g$ has type $[\Gamma \mid \mathsf{self} : T] \rightsquigarrow \Gamma'$. Suppose $T \le \Gamma\,\mathsf{self}$. Then $g$ ***refines*** $f$ ***at exact type*** $T$, written $g \sqsupseteq^{\dot{T}} f$, if and only if $\sigma \in g(Q) \Leftarrow \sigma \in f(Q)$ for all $\Gamma$-states $\sigma$ such that $selftype(\sigma) = T$, and all $\Gamma'$-predicates $Q$.

The definition is justified by the following result. The proof expands definitions and uses Lemma 7.7; it is in Appendix A.8.

LEMMA 7.10. Suppose $spec_0$ has type $\Gamma \rightsquigarrow \Gamma'$ and $spec_1$ has type $[\Gamma \mid \mathsf{self} : T] \rightsquigarrow \Gamma'$ and $T \le \Gamma\,\mathsf{self}$. If the specifications are satisfiable then $spec_1 \sqsupseteq^T spec_0$ iff $\{\!|spec_1|\!\} \sqsupseteq^T \{\!|spec_0|\!\}$.

The "only if" direction does not require satisfiability, but we need the "if" direction later.

### 7.4. The canonical method environment

By using the predicate transformer semantics of each specification, we can define a method environment $\{\!|ST|\!\}$ pointwise, that is:

$$\{\!|ST|\!\}(T, m) = \{\!|ST(T, m)|\!\} \quad \text{for all } T, m$$

We write $\{\!|ST|\!\}$ for either the extended method environment given for all types $T$ or for the normal one for just classes. Context should disambiguate, as in the following Lemma where we compare like kinds of environments.

LEMMA 7.11. For $\eta$ a normal or extended method environment, $\eta \models ST$ iff $wp(\eta) \sqsupseteq \{\!|ST|\!\}$.

PROOF. Let $T$ range over classes (for normal environments) or all ref types (for extended ones). Observe:

$$\begin{array}{ll} & wp(\eta) \sqsupseteq \{\!|ST|\!\} \\ \iff & \quad \text{definition of } \sqsupseteq \text{ for method environments} \\ & \forall T, m \cdot wp(\eta)(T,m) \sqsupseteq \{\!|ST|\!\}(T,m) \\ \iff & \quad \text{definition of } wp \text{ for method environments, definition of } \{\!|ST|\!\} \\ & \forall T, m \cdot wp(\eta(T,m)) \sqsupseteq \{\!|ST(T,m)|\!\} \\ \iff & \quad \text{Lemma 7.5} \\ & \forall T, m \cdot \eta(T,m) \models ST(T,m) \\ \iff & \quad \text{definition of } \models \text{ for method environments} \\ & \eta \models ST \end{array}$$

□

## 8. BEHAVIORAL SUBTYPING AND ITS EQUIVALENCE TO SUPERTYPE ABSTRACTION

This section formalizes behavioral subtyping (Sect. 8.1) and supertype abstraction (Sect. 8.2) Then they are proven to be equivalent (Sect. 8.3). The implication from left to right means that if the specification table has behavioral subtyping then it is sound to reason about any command using, for method calls, only the specification of the receiver object's static type. The converse means that our notion of behavioral subtyping is complete: it is necessary for modular reasoning. Along the way we consider interesting variations on each notion.

### 8.1. Behavioral subtyping

Behavioral subtyping is defined in terms of the intrinsic ordering on specifications.

*Definition* 8.1. A specification table $ST$ has ***behavioral subtyping*** if and only if for all ref types $U$, method names $m \in Meths\ U$, and classes $K$, we have

$$K \leq U \ \Rightarrow\ ST(K,m) \sqsupseteq^K ST(U,m)$$

Note that quantification is over classes $K$ that are subtypes of $U$, ignoring interface subtypes of $U$.

The following variation is important in Sect. 9 where we study specification inheritance. The only difference is $\sqsupseteq^{*K}$ in place of $\sqsupseteq^K$.

*Definition* 8.2. A specification table $ST$ has ***robust behavioral subtyping*** if the following holds for all $U$, all $m \in Meths\ U$, and all $K$:

$$K \leq U \ \Rightarrow\ ST(K,m) \sqsupseteq^{*K} ST(U,m)$$

The reader may check that the example Fig. 1 in Sect. 2.2 has robust behavioral subtyping.

The following is a direct consequence of Lemma 5.12 in Sect. 5.3.

LEMMA 8.3. Robust behavioral subtyping implies behavioral subtyping.

The implication is strict. As observed following Lemma 5.12, $\sqsupseteq^{*K}$ is strictly stronger than $\sqsupseteq^K$. Here is a contrived example with behavioral subtyping that is not robust.

*Example* 8.4. Consider three types with $L < K < T$ where type $T$ has a Boolean field $f$ and a method $m$. Consider the specifications $ST(T,m) = (true, true)$ and $ST(K,m) = (P, true)$, where $P$ is the set denoted by "$\neg(\mathsf{self}\ \mathbf{is}\ L) \vee \mathsf{self}.f$". Note that $ST(K,m)$ is satisfied by $\varphi_K$ where $\varphi_K$ is the semantics of **if** self **is** $L \wedge \neg\mathsf{self}.f$ **then** *diverge* **else** *skip*. We have $ST(K,m) \sqsupseteq^K ST(T,m)$ but not $ST(K,m) \sqsupseteq^{*K} ST(T,m)$, for example $\varphi_K$ does not satisfy $ST(T,m)\lfloor^* K$.

Practical examples seem to be robust, even when exact type tests are used as in the following.

*Example* 8.5. The classes Cell and DCell in Fig. 7 are adapted from Parkinson and Bierman [2008]. They exemplify situations where inheritance is intended as a mechanism for code reuse, with no expectation that instances of the subtype are substitutable (or useful in contexts expecting the supertype). This is manifest in the specifications, which involve exact type tests like **selftype** = Cell.

The example also illustrates how data abstraction fits with our theory. The specifications use model fields, each defined using JML's **represents** syntax. What we intend Fig. 7 to denote is the

```
class Cell extends Object {
  public model Val: int;
  protected v: int;       protected represents Val := v;

  meth set(x: int)
    requires selftype = Cell;
    ensures exc = null ∧ Val = x;
  { v := x; }

  meth get(): int
    requires selftype = Cell;
    ensures exc = null ∧ res = Val;
  { res := v; }
}

class DCell extends Object {
  public model DVal: int;     protected represents DVal := v;

  meth set(x: int)
    requires true;
    ensures exc = null ∧ (selftype = DCell ⇒ DVal = 2*x) ∧ (selftype = Cell ⇒ Val = x);
  { v := 2*x; }

  meth get(): int
    requires true;
    ensures exc = null ∧ (selftype = DCell ⇒ res = DVal) ∧ (selftype = Cell ⇒ res = Val);
  { res := v; }
}
```

Fig. 7. The classes Cell and DCell, where the notation **selftype** means the runtime type of self.

specification table $ST_{CD}$ where $ST_{CD}(\mathsf{Cell}, \mathsf{set})$ and $ST_{CD}(\mathsf{DCell}, \mathsf{set})$ are these simple specifications

$$\begin{array}{rcl} pre_{CS} & = & \{(r,h,s) \mid (selftype(r,h,s) = \mathsf{Cell}\ \} \\ post_{CS} & = & \{(r,h,s) \mid s\ \mathsf{exc} = null \wedge h(s\ \mathsf{self})\mathsf{v} = s\ \mathsf{x})\} \\ pre_{DS} & = & \{(r,h,s) \mid true\} \\ post_{DS} & = & \{(r,h,s) \mid s\ \mathsf{exc} = null\ \wedge (selftype(r,h,s) = \mathsf{DCell} \Rightarrow h(s\ \mathsf{self})v = 2 * (s\ \mathsf{x})) \\ & & \qquad\qquad \wedge (selftype(r,h,s) = \mathsf{Cell} \Rightarrow h(s\ \mathsf{self})v = (s\ \mathsf{x})))\} \end{array}$$

The reader can check (using Corollary 5.21) that $ST_{CD}(\mathsf{DCell}, \mathsf{set}) \sqsupseteq^{*\mathsf{DCell}} ST_{CD}(\mathsf{Cell}, \mathsf{set})$ since the precondition from Cell's method becomes false when the type of self is restricted to DCell. Similarly for method get. Thus $ST_{CD}$ has robust behavioral subtyping. Specifications that require self to have a class's exact type, such as the specifications for Cell's methods in Fig. 7, do not impose any restrictions on the behavior of overriding subtype methods. Such specifications are not "subtype-constraining" in a sense similar to that discussed in Section 3.5.3 of Leavens [1990].

The specifications of Parkinson and Bierman [2008] look somewhat different from Fig. 7. They abstract from the explicit conditions involving specific types, using "abstract predicate families". The rough idea is to use a named predicate that has different interpretations in different subclasses.[17] A similar effect may be achieved in JML using pure methods, but this is beyond our scope.

### 8.2. Supertype abstraction

Recall that modular correctness of a command means it satisfies its specification when interpreted in any method environment that satisfies the assumed specification table $ST$ (see Def. 6.2). Modular

[17] Their paper also distinguishes between "dynamic specifications", used to reason about dynamically dispatched calls, and "static specifications" which are used for statically dispatched calls and to verify the method implementation. That feature is not used for this example.

verification means the command refines its specification, in predicate transformer semantics. (See Def. 6.4.) Supertype abstraction amounts to being able to establish modular correctness, either by modular verification (the first definition below) or by proving modular correctness under static dispatch (the second).

*Definition* 8.6. We say $ST$ ***allows weak supertype abstraction*** if and only if

$$\mathcal{S}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\}) \sqsupseteq \{\!|spec|\!\} \quad \text{implies} \quad ST, \mathcal{P} \models^{\mathcal{D}} spec \tag{34}$$

for every program phrase-in-context $\mathcal{P}$ and every *spec*.

*Definition* 8.7. We say $ST$ ***allows strong supertype abstraction*** if and only if, for all $\mathcal{P}$, *spec*

$$ST, \mathcal{P} \models^{\mathcal{S}} spec \quad \text{implies} \quad ST, \mathcal{P} \models^{\mathcal{D}} spec \tag{35}$$

Some readers might wonder about a variation on this definition in which single quantification over method environments is used: $\forall \dot{\eta} \in XMethEnv \cdot \dot{\eta} \models ST \Rightarrow (\mathcal{S}[\![\mathcal{P}]\!](\dot{\eta}) \models spec \Rightarrow \mathcal{D}[\![\mathcal{P}]\!](\dot{\eta}) \models spec)$. Of course this logically implies strong supertype abstraction. An example in Sect. A.11 confirms that it is strictly stronger, and does not follow from behavioral subtyping. There is yet another notion that is interesting, namely that $\mathcal{S}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\}) \sqsupseteq spec$ implies $\mathcal{D}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\}) \sqsupseteq spec$ for all *spec* —which amounts to $\mathcal{D}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\}) \sqsupseteq \mathcal{S}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\})$. This is considered near the end of this subsection.

Part (a) of the following Theorem says that modular verification is a sound way to establish modular correctness under static dispatch. Part (b) is the corresponding property for dynamic dispatch. A consequence of part (a) is that strong supertype abstraction implies weak supertype abstraction. (See the upper triangle in the diagram Eq. (25).)

THEOREM 8.8. For any $ST$, any phrase-in-context $\mathcal{P}$, and any suitably typed *spec*,

(a) $\mathcal{S}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\}) \sqsupseteq \{\!|spec|\!\}$ implies $ST, \mathcal{P} \models^{\mathcal{S}} spec$
(b) $\mathcal{D}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\}) \sqsupseteq \{\!|spec|\!\}$ implies $ST, \mathcal{P} \models^{\mathcal{D}} spec$

PROOF. For (a), observe that

$$\begin{array}{ll} & ST, \mathcal{P} \models^{\mathcal{S}} spec \\ \iff & \quad \text{Def. 6.3} \\ & \forall \eta \cdot \eta \models ST \Rightarrow \mathcal{S}[\![\mathcal{P}]\!](\eta) \models spec \\ \iff & \quad \text{Lemma 7.5} \\ & \forall \eta \cdot \eta \models ST \Rightarrow wp(\mathcal{S}[\![\mathcal{P}]\!](\eta)) \sqsupseteq \{\!|spec|\!\} \\ \iff & \quad \text{wp-equivalence Lemma 7.1} \\ & \forall \eta \cdot \eta \models ST \Rightarrow \mathcal{S}\{\!|\mathcal{P}|\!\}(wp(\eta)) \sqsupseteq \{\!|spec|\!\} \\ \iff & \quad \text{Lemma 7.11} \\ & \forall \eta \cdot wp(\eta) \sqsupseteq \{\!|ST|\!\} \Rightarrow \mathcal{S}\{\!|\mathcal{P}|\!\}(wp(\eta)) \sqsupseteq \{\!|spec|\!\} \end{array}$$

The last line follows from $\mathcal{S}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\}) \sqsupseteq \{\!|spec|\!\}$ using transitivity of $\sqsupseteq$. In detail: $wp(\eta) \sqsupseteq \{\!|ST|\!\}$ implies $\mathcal{S}\{\!|\mathcal{P}|\!\}(wp(\eta)) \sqsupseteq \mathcal{S}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\})$ by monotonicity of $\mathcal{S}\{\!|\mathcal{P}|\!\}(-)$ (Lemma 7.2). Then $\mathcal{S}\{\!|\mathcal{P}|\!\}(wp(\eta)) \sqsupseteq \mathcal{S}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\}) \sqsupseteq \{\!|spec|\!\}$.

For (b) the argument is essentially the same except for using Def. 6.2 in place of Def. 6.3 in the first step, and $\mathcal{D}$ for $\mathcal{S}$ throughout. □

The implications are strict. Indeed, modular verification is not complete with respect to modular correctness under static dispatch. That is, there are programs with specifications that are modularly correct under static dispatch but are not modularly verified.

*Example* 8.9. Let CT have one user-defined class $K$, with no fields and one method $m$ that has no parameters and return type int. Let $ST(K, m)$ be the simple specification $(pre, post)$, of type $(K, m)$, with precondition $pre(r, h, s) = (dom(h) = \{s(\mathsf{self})\})$, which means that the heap has a single object, and postcondition $post(r, h, s) = (s(\mathsf{res}) = 0 \vee s(\mathsf{res}) = 1)$, meaning the method returns either 0 or 1.

Let $C$ be the command $x := \mathsf{self}.m(); y := \mathsf{self}.m()$ in a state space with integer variables $x, y$. Let *spec* have pre true and post $x = y$. Then $ST, C \models^{\mathcal{S}} spec$ because this notion quantifies over deterministic state transformers and $m$ has no way to depend on state; so it is a constant function. However, we do not have $\mathcal{S}\{\!|C|\!\}(\{\!|ST|\!\}) \sqsupseteq \{\!|spec|\!\}$.

The point is that by considering semantic consequences in quantifying over all satisfying implementations, we may reason about properties like determinacy or computability that may not be

explicitly stated (or even expressible) in a specification. In the case of determinacy, the property is faithfully represented by the canonical method environment in predicate transformer semantics.

In this example, the precondition *pre* is a bit odd, but our specifications are semantic and allow us to say such things. Likewise, our method environments include exotic state transformers that aren't denoted by programs, e.g., because they depend on unreachable objects or non-visible fields. □

As an alternative to our definitions of supertype abstraction, some readers may be content with a formulation based entirely on predicate transformer semantics. The following Theorem says that this notion of supertype abstraction follows from behavioral subtyping (see lower triangle in the diagram Eq. (25)). In our view, the use of $\{\!|ST|\!\}$ ought to be justified in terms of actual program semantics, which is what is achieved by our main soundness Thm. 8.15.

THEOREM 8.10. If $ST$ is satisfiable and has behavioral subtyping then for any phrase-in-context $\mathcal{P}$ we have $\mathcal{D}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\}) \sqsupseteq \mathcal{S}\{\!|\mathcal{P}|\!\}(\{\!|ST|\!\})$.

PROOF. If $ST$ is satisfiable and has behavioral subtyping, then by Lemma 7.10 we have $\{\!|ST|\!\}(K, m) \sqsupseteq^K \{\!|ST|\!\}(T, m)$ for all $T$, all $m \in \mathit{Meths}\ T$, and all $K \leq T$. This lets us appeal to the following Lemma 8.11, which generalizes the Theorem to any method environment with this semantic form of behavioral subtyping. □

The refinement in the Theorem is strict in general. Dynamic dispatch with behavioral subtyping can impose strictly stronger specifications when the value of self is a subtype of its static type.

LEMMA 8.11. Let $\theta$ be a predicate transformer method environment such that

$$\theta(K, m) \sqsupseteq^K \theta(T, m) \tag{36}$$

for all $T, m, K$ such that $m \in \mathit{Meths}\ T$ and $K \leq T$. Then $\mathcal{D}\{\!|\mathcal{P}|\!\}(\theta) \sqsupseteq \mathcal{S}\{\!|\mathcal{P}|\!\}(\theta)$ for any phrase-in-context $\mathcal{P}$.

The proof is by structural induction on $\mathcal{P}$. The crux of the proof is that $\theta(K, m) \sqsupseteq^K \theta(T, m)$ implies the refinement in case $\mathcal{P}$ is a call to $m$. The other cases go through because $\mathcal{S}\{\!|-|\!\}$ and $\mathcal{D}\{\!|-|\!\}$ are the same for other constructs, and control structures are monotonic in their constituent parts, with respect to refinement. The details are in Appendix A.10.

### 8.3. Equivalence of behavioral subtyping and supertype abstraction

In this section we prove the main result of the paper: that behavioral subtyping is equivalent to supertype abstraction both strong and weak. We begin with some preliminary results and definitions.

Among the extended method environments are some which represent dynamic dispatch in the sense that they make the static dispatch semantics act as if dynamic —provided we have behavioral subtyping.

*Definition* 8.12 (*dynamic extension of a normal environment*). Let $\eta$ be a normal method environment. The ***dynamic extension*** of $\eta$ is the extended environment $\breve{\eta}$ defined by

$$\breve{\eta}(T, m)(\sigma) = \eta(\mathit{selftype}(\sigma), m)(\sigma) \qquad \text{for all } T,\ m,\ \sigma.$$

Note that $\breve{\eta}$ is not simply an extension of $\eta$ to apply to interface types; in general $\breve{\eta}(K, m)$ differs from $\eta(K, m)$. The effect is like having a single implementation that conditionally branches on the dynamic type of self.[18] One important property of dynamic extension is that $\breve{\eta}$ is correct if $\eta$ is. The proof depends on the characterization of refinement, Prop. 5.18, via Corollary 5.19.

LEMMA 8.13 (CORRECTNESS OF DYNAMIC EXTENSION). Suppose that $ST$ has behavioral subtyping and $\eta$ is a normal method environment with $\eta \models ST$. Let $\breve{\eta}$ be given by Def. 8.12. Then $\breve{\eta} \models ST$.

PROOF. Consider any $T$ and $m$. Suppose $ST(T, m)$ is a specification of type $\Gamma \rightsquigarrow \Gamma'$. To show that $\breve{\eta}(T, m) \models ST(T, m)$, recall first that behavioral subtyping says $ST(K, m) \sqsupseteq^K ST(T, m)$ for all classes $K \leq T$. We need to spell this out at the level of states. Consider any $K$ with $K \leq T$. Let $(I, \mathit{pre}, \mathit{post}) = ST(T, m)$ and $(J, \mathit{pre}', \mathit{post}') = ST(K, m)$. Note that $(J, \mathit{pre}', \mathit{post}')$ is satisfiable, as

[18] Such an implementation can be written in our programming language, given that the set of classes is fixed. It is used as a normal form by Borba et al. [2004]. However, our results do not depend on it being expressible as code.

$\eta \models ST$. So Corollary 5.19 is applicable: it says the behavioral subtyping condition $ST(K,m) \sqsupseteq^K ST(T,m)$ is equivalent to

$\forall i \in I, \sigma \in \mathit{State}(\Gamma) \cdot \sigma \in \mathit{pre}_i \lfloor T$
$\Rightarrow (\exists j \in J \cdot \sigma \in \mathit{pre}'_j) \wedge (\forall \tau \in \mathit{State}(\Gamma') \cdot (\forall j \in J \cdot \sigma \in \mathit{pre}'_j \Rightarrow \tau \in \mathit{post}'_j) \Rightarrow \tau \in \mathit{post}_i)$.

Now we prove $\breve{\eta}(T,m) \models ST(T,m)$. Observe for any $i \in I$ and any $\sigma$, writing $K$ for $\mathit{selftype}(\sigma)$.

$\sigma \in \mathit{pre}_i$
$\iff$ by $\mathit{selftype}(\sigma) = K$ and definition of $\lfloor K$
$\sigma \in \mathit{pre}_i \lfloor K$
$\Rightarrow$ by semantics, $K \leq T$; now use behavioral subtyping as spelled out above
$(\exists j \in J \cdot \sigma \in \mathit{pre}'_j) \wedge (\forall \tau \in \mathit{State}(\Gamma') \cdot (\forall j \in J \cdot \sigma \in \mathit{pre}'_j \Rightarrow \tau \in \mathit{post}'_j) \Rightarrow \tau \in \mathit{post}_i)$
$\Rightarrow$ logic: drop conjunct, instantiate $\tau$ as $\eta(K,m)(\sigma)$
$(\forall j \in J \cdot \sigma \in \mathit{pre}'_j \Rightarrow \eta(K,m)(\sigma) \in \mathit{post}'_j) \Rightarrow \eta(K,m)(\sigma) \in \mathit{post}_i$
$\Rightarrow$ antecedent follows from $\eta(K,m) \models (J, \mathit{pre}', \mathit{post}')$, from $\eta \models ST$
$\eta(K,m)(\sigma) \in \mathit{post}_i$
$\iff$ definition of $\breve{\eta}(T,m)$, $\mathit{selftype}(\sigma)$ is $K$
$\breve{\eta}(T,m)(\sigma) \in \mathit{post}_i$

□

The other important property is that dynamic extension encodes dynamic dispatch.

LEMMA 8.14 (COMPLETENESS OF DYNAMIC EXTENSION). Suppose that $ST$ has behavioral subtyping. Let $\eta$ be a normal method environment with $\eta \models ST$, and let $\breve{\eta}$ be given by Def. 8.12. Then for any phrase-in-context $\mathcal{P}$ we have $\mathcal{S}[\![\mathcal{P}]\!](\breve{\eta}) = \mathcal{D}[\![\mathcal{P}]\!](\eta)$.

PROOF. By induction on structure of $\mathcal{P}$. For any primitive expression or command, aside from method call, the equation holds by the definitions of $\mathcal{S}[\![-]\!]$ and $\mathcal{D}[\![-]\!]$, which coincide and are independent from the method environment in these cases.

For a method call, consider a context $\Gamma$ such that $x : T$ is in $\Gamma$ and observe that for each state $(r,h,s) \in \mathit{State}(\Gamma)$:

$\mathcal{D}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\eta)(r,h,s)$
$=$ semantics
**if** $s\,x = \mathit{null}$ **then** $\mathit{except}(r,h,U,\mathsf{NullDeref})$
**else let** $K = r(s\,x)$ **in let** $\overline{z} = \mathit{formals}(K,m)$ **in**
**let** $s_1 = [\mathsf{self} : s\,x,\ \overline{z : s\,y}]$ **in**
$\eta(K,m)(r,h,s_1)$
$=$ using $\eta(K,m)(r,h,s_1) = \breve{\eta}(T,m)(r,h,s_1)$ from definition of $\breve{\eta}$, see below
**if** $s\,x = \mathit{null}$ **then** $\mathit{except}(r,h,U,\mathsf{NullDeref})$
**else let** $K = r(s\,x)$ **in let** $\overline{z} = \mathit{formals}(K,m)$ **in**
**let** $s_1 = [\mathsf{self} : s\,x,\ \overline{z : s\,y}]$ **in**
$\breve{\eta}(T,m)(r,h,s_1)$
$=$ semantics, $x$ has type $T$, $\mathit{formals}(K,m) = \mathit{formals}(T,m)$ as class table well-formed
$\mathcal{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\breve{\eta})(r,h,s)$

In the step that appeals to the definition of $\breve{\eta}$, we use that $\mathit{selftype}(r,h,s_1) = r(s_1\,\mathsf{self}) = r(s\,x)$, and the local binding $K = r(s\,x)$.

For the remaining constructs, the argument is by direct use of the induction hypothesis and the fact that the static and dynamic dispatch semantics are the same for those constructs. □

The above manipulation of method call semantics would look more pleasant using the alternative formulation we mentioned in connection with the function *dispatch* defined by Eq. (14). The relevant property of $\breve{\eta}$ is then $\breve{\eta}(K,m)(r,h,s_1) = \mathit{dispatch}(T,m,\eta)(r,h,s_1)$ where $K = \mathit{selftype}(r,h,s_1)$.

A related property of *dispatch* is used to prove the following Theorem. For all class names $K$, and for all *spec*, $T$, $m$, and $\eta$:

$$\mathit{dispatch}(T,m,\eta) \models \mathit{spec} \iff \forall K \cdot K \leq T \Rightarrow \eta(K,m) \models \mathit{spec} \lfloor K. \tag{37}$$

To prove Eq. (37), we consider first the case of a simple specification $(pre, post)$. Observe for any $T$, $m$, and $\eta$, where self has static type $T$

$$\begin{array}{ll} & dispatch(T,m,\eta) \models (pre, post) \\ \iff & \quad \text{definitions of } \models \text{ and } dispatch \\ & \forall r,h,s \cdot (r,h,s) \models pre \Rightarrow \eta(r(s(\mathsf{self})),m)(r,h,s) \models post \\ \iff & \quad \mathsf{self} \text{ is non-null, definition of } Val(T,r) \\ & \forall K \cdot K \leq T \Rightarrow \forall r,h,s \cdot (r,h,s) \models pre \wedge r(s(\mathsf{self})) = K \Rightarrow \eta(K,m)(r,h,s) \models post \\ \iff & \quad \text{definition of } \lfloor \\ & \forall K \cdot K \leq T \Rightarrow \forall r,h,s \cdot (r,h,s) \models pre\lfloor K \Rightarrow \eta(K,m)(r,h,s) \models post \\ \iff & \quad \text{definition of } \models \\ & \forall K \cdot K \leq T \Rightarrow \eta(K,m) \models (pre, post)\lfloor K \end{array}$$

For a general specification $(J, pre, post)$, there is merely an extra quantification over elements of $J$.

Finally we can prove the main result of the paper.

THEOREM 8.15. For any satisfiable $ST$ the following are equivalent.

(a) $ST$ has behavioral subtyping (Def. 8.1).
(b) $ST$ allows strong supertype abstraction (Def. 8.7).
(c) $ST$ allows weak supertype abstraction (Def. 8.6).

PROOF. By cyclic implication.

**(a) implies (b):** Suppose $ST$ has behavioral subtyping. Consider any phrase in context $\mathcal{P}$ and any *spec*. Suppose $ST, \mathcal{P} \models^{\mathcal{S}} spec$. To show $ST, \mathcal{P} \models^{\mathcal{D}} spec$, consider any (normal) $\eta$ such that $\eta \models ST$. Let $\breve{\eta}$ be given by Def. 8.12. By Lemma 8.13 we have $\breve{\eta} \models ST$. So from $ST, \mathcal{P} \models^{\mathcal{S}} spec$ we have $\mathcal{S}[\![\mathcal{P}]\!](\breve{\eta}) \models spec$, hence by Lemma 8.14 we have $\mathcal{D}[\![\mathcal{P}]\!](\eta) \models spec$.

**(b) implies (c):** To prove (c) we observe for any $\mathcal{P}$ and any *spec*

$$\begin{array}{ll} & \mathcal{S}\{\![\mathcal{P}]\!\}(\{\![ST]\!\}) \sqsupseteq \{\![spec]\!\} \\ \Rightarrow & \quad \text{Thm. 8.8(a)} \\ & ST, \mathcal{P} \models^{\mathcal{S}} spec \\ \Rightarrow & \quad \text{assumption (b)} \\ & ST, \mathcal{P} \models^{\mathcal{D}} spec \end{array}$$

**(c) implies (a):** Assume (c). To prove (a), consider any $T$ and any $m$ in *Meths* $T$. We must show

$$\forall K \cdot K \leq T \Rightarrow ST(K,m) \sqsupseteq^{K} ST(T,m)$$

Take $\Gamma := \mathsf{self} : T, \overline{y : U}$ where $mtype(m,T) = \overline{y : U} \to U$. We instantiate (c) with the expression $\mathsf{self}.m(\overline{y})$ and specification $ST(T,m)$. That is, we assume this instance of weak supertype abstraction:

$$\mathcal{S}\{\![\Gamma \vdash \mathsf{self}.m(\overline{y}) : U]\!\}(\{\![ST]\!\}) \sqsupseteq \{\![ST(T,m)]\!\} \tag{38}$$

$$\Rightarrow$$

$$\forall \eta \cdot \eta \models ST \Rightarrow \mathcal{D}[\![\Gamma \vdash \mathsf{self}.m(\overline{y}) : U]\!](\eta) \models ST(T,m) \tag{39}$$

Next we show that (38) holds. Instantiating the semantic definition in Fig. 6 we get

$$\begin{array}{ll} (r,h,s) \in \mathcal{S}\{\![\Gamma \vdash \mathsf{self}.m(\overline{y}) : U]\!\}(\{\![ST]\!\})(Q) & \\ \quad \iff & \textbf{if } s(\mathsf{self}) = null \textbf{ then } except(r,h,U,\mathsf{NullDeref}) \in Q \\ & \textbf{else let } T = \Gamma(\mathsf{self}) \textbf{ in let } \overline{z} = formals(T,m) \textbf{ in} \\ & \textbf{let } s_1 = [\mathsf{self} : s(\mathsf{self}), \overline{z : s\ y}] \textbf{ in } (r,h,s_1) \in \{\![ST]\!\}(T,m)(Q) \end{array}$$

for all $\Gamma$-states $(r,h,s)$ and all predicates $Q$ on $[\mathsf{res} : U, \mathsf{exc} : \mathsf{Thr}]$. By definition of states, self is never null, see Eq. (8), so the conditional can be simplified. Moreover, $s_1$ is just $s$. So we have

$$\mathcal{S}\{\![\Gamma \vdash \mathsf{self}.m(\overline{y}) : U]\!\}(\{\![ST]\!\}) = \{\![ST]\!\}(T,m)$$

Similar simplification can be applied to the dynamic state transformer semantics to obtain, for any $\eta$,

$$\mathcal{D}[\![\Gamma \vdash \mathsf{self}.m(\overline{y}) : U]\!](\eta) = dispatch(T,m,\eta) \tag{40}$$

where *dispatch* is defined in Eq. (14). Now (38) simplifies to $\{\![ST]\!\}(T,m) \sqsupseteq \{\![ST(T,m)]\!\}$ which holds by definition of $\{\![ST]\!\}$ and reflexivity of $\sqsupseteq$. So by the assumed implication, we obtain (39). From this we can derive the required conclusion, i.e., behavioral subtyping at $T$:

$$
\begin{array}{ll}
 & (39) \\
\iff & \quad \text{equation (40)} \\
 & \forall \eta \cdot \eta \models ST \;\Rightarrow\; \mathit{dispatch}(T,m,\eta) \models ST(T,m) \\
\iff & \quad \text{property (37) of } \mathit{dispatch} \\
 & \forall \eta \cdot \eta \models ST \;\Rightarrow\; (\forall K \cdot K \leq T \;\Rightarrow\; \eta(K,m) \models ST(T,m)\lfloor K) \\
\iff & \quad ST \text{ satisfiable, predicate calculus (coordinate transformation, see below)} \\
 & \forall K \cdot K \leq T \;\Rightarrow\; (\forall \varphi \cdot \varphi \models ST(K,m) \;\Rightarrow\; \varphi \models ST(T,m)\lfloor K) \\
\iff & \quad \text{definition of } \sqsupseteq^K \\
 & \forall K \cdot K \leq T \;\Rightarrow\; ST(K,m) \sqsupseteq^K ST(T,m)
\end{array}
$$

For the third step, only the values of $\eta(K,m)$ for $K \leq T$ are relevant, so since $ST$ is satisfiable and the choice of $\eta$ is not dependent on $K$, we can restrict quantification to that part of $\eta$. □

The satisfiability hypothesis is necessary: If $ST$ is unsatisfiable then it allows both strong and weak supertype abstraction, but an unsatisfiable $ST$ need not have behavioral subtyping. Here is why both forms of supertype abstraction are allowed if $ST$ is unsatisfiable. For any $C$ and any *spec*, $C$ modularly satisfies *spec* w.r.t. an unsatisfiable $ST$ because the antecedent is false in Def. 6.2(20). Hence the consequent of (34) in Def. 8.6, and the consequent of (35) in Def. 8.7, both hold.

*Remark* 8.16. The implications (a) ⇒ (b) and (a) ⇒ (c) of Thm 8.15, i.e., soundness of behavioral subtyping, justify the diagonal and rightmost vertical in diagram (25) at the end of Sect. 6. The topmost horizontal depicts why (b) ⇒ (c), whence (a) ⇒ (c). In light of the diagram, let us point out an alternate proof of (a) ⇒ (c). To show (c) directly, i.e., modular verification implies modular correctness, let $\mathcal{P}$ be a phrase-in-context and *spec* be of the appropriate type for $\mathcal{P}$. Observe

$$
\begin{array}{ll}
 & \mathcal{S}\{\![\mathcal{P}]\!\}(\{\![ST]\!\}) \sqsupseteq \{\![\mathit{spec}]\!\} \\
\Rightarrow & \quad \text{Thm 8.10, using behavioral subtyping (a) and satisfiability of } ST\text{; } \sqsupseteq \text{ transitive} \\
 & \mathcal{D}\{\![\mathcal{P}]\!\}(\{\![ST]\!\}) \sqsupseteq \{\![\mathit{spec}]\!\} \\
\Rightarrow & \quad \text{Thm 8.8(b)} \\
 & ST, \mathcal{P} \models^{\mathcal{D}} \mathit{spec}
\end{array}
$$

We can get (b) ⇒ (a) by direct argument similar to our proof of (c) ⇒ (a). In place of (38) one has

$$\forall \eta \cdot \eta \models ST \;\Rightarrow\; \mathcal{S}[\![\Gamma \vdash \mathsf{self}.m(\overline{y}) : U]\!]\eta \models ST(T,m) \tag{41}$$

which by semantics of $\mathsf{self}.m(\overline{y})$ simplifies to $\forall \eta \cdot \eta \models ST \;\Rightarrow\; \eta(T,m) \models ST(T,m)$. This yields (39) and the rest is unchanged. It does not appear easy to prove (c) ⇒ (b) directly.

## 9. ENSURING BEHAVIORAL SUBTYPING BY SPECIFICATION INHERITANCE

In the preceding sections, behavioral subtyping is defined in terms of the intrinsic refinement relation, which is defined by quantifying over all state transformers. For practical purposes, we need means to check whether specifications have behavioral subtyping as well as means to construct such specifications. For checking, a characterization of refinement in terms of pre/post conditions is given in Sect. 5.4 (Corollary 5.19). The constructive approach takes an arbitrary collection of specifications and derives specifications that have behavioral subtyping; this technique, which is the topic of this section, is called specification inheritance. The key property of specification inheritance [Leavens 2006; Wills 1992] is that it forces behavioral subtyping [Dhara and Leavens 1996], as we show in Sect. 9.2. The technique is based on joins of specifications (Sect. 9.1). The concluding Sect. 9.3 connects specification inheritance with inheritance of implementations.

### 9.1. Joins of specifications

In this subsection we show that ***joins*** (i.e., least upper bounds) exist, by explicit constructions for the general case and for specifications in two-state form. Specification inheritance is defined in the sequel using joins of specifications of the same type, so we only need joins with respect to ordinary refinement $\sqsupseteq$, not refinement at a subtype. Moreover, we only need joins of finitely many specifications. It suffices to show that binary joins exist since these can be used to obtain finite joins.

We begin with a special case in which there is a simple construction.

LEMMA 9.1 (JOIN OF DISJOINT GENERAL SPECIFICATIONS). Suppose that $(I, pre, post)$ and $(J, pre', post')$ are specifications of type $\Gamma \rightsquigarrow \Gamma'$, and that $I$ and $J$ are disjoint (i.e., $I \cap J = \emptyset$). Then $(I \cup J,\ pre \cup pre',\ post \cup post')$ is a least upper bound of $(I, pre, post)$ and $(J, pre', post')$ with respect to refinement of specifications of type $\Gamma \rightsquigarrow \Gamma'$.

Here we treat functions *pre*, *post*, etc., as sets of ordered pairs. So, by disjointness of $I$ and $J$, the unions $pre \cup pre'$ and $post \cup post'$ are functions with domain $I \cup J$. For example, $(pre \cup pre')_i = pre_i$ for $i \in I$. Thus $(I \cup J,\ pre \cup pre',\ post \cup post')$ is indeed a specification.

PROOF. Recall that the least upper bound property of a specification $spec_0$ is

$$\forall spec \cdot spec \sqsupseteq spec_0 \iff spec \sqsupseteq (I, pre, post) \ \wedge \ spec \sqsupseteq (J, pre', post').$$

The least upper bound property holds because for any *spec* of type $\Gamma \rightsquigarrow \Gamma'$ we have:

$$\begin{array}{ll} & spec \sqsupseteq (I \cup J, pre \cup pre', post \cup post') \\ \iff & \quad \text{definition of } \sqsupseteq\text{, omit range } \varphi \in STrans(\Gamma, \Gamma') \\ & \forall \varphi \cdot \varphi \models spec \ \Rightarrow \ \varphi \models (I \cup J, pre \cup pre', post \cup post') \\ \iff & \quad \text{definition of } \models \\ & \forall \varphi \cdot \varphi \models spec \ \Rightarrow \ \forall j \in I \cup J \cdot \varphi \models ((pre \cup pre')_j, (post \cup post')_j) \\ \iff & \quad \text{disjointness of } I \text{ and } J\text{, union of disjoint functions} \\ & \forall \varphi \cdot \varphi \models spec \ \Rightarrow \ (\forall i \in I \cdot \varphi \models (pre_i, post_i)) \wedge (\forall j \in J \cdot \varphi \models (pre'_j, post'_j)) \\ \iff & \quad \text{predicate calculus, definition of } \models \\ & (\forall \varphi \cdot \varphi \models spec \ \Rightarrow \ \varphi \models (I, pre, post)) \wedge (\forall \varphi \cdot \varphi \models spec \ \Rightarrow \ \varphi \models (J, pre', post')) \\ \iff & \quad \text{definition of } \models\text{, definition of } \sqsupseteq \\ & spec \sqsupseteq (I, pre, post) \ \wedge \ spec \sqsupseteq (J, pre', post') \end{array}$$

□

LEMMA 9.2 (JOIN OF SPECIFICATIONS). Suppose that $(I, pre, post)$ and $(J, pre', post')$ are specifications of type $\Gamma \rightsquigarrow \Gamma'$. Then there exists a specification $spec_0$ that is a least upper bound of $(I, pre, post)$ and $(J, pre', post')$ with respect to refinement of specifications of type $\Gamma \rightsquigarrow \Gamma'$.

PROOF. Define $I + J$ to be the disjoint union $\{(i, 0) \mid i \in I\} \cup \{(j, 1) \mid j \in J\}$ with injection functions $inl : I \to I + J$ and $inr : J \to I + J$. Write $inl(I)$ for the image $\{(i, 0) \mid i \in I\}$ and observe that $(inl(I), pre \circ inl^{-1}, post \circ inl^{-1})$ is a specification. It is straightforward to show that $(inl(I), pre \circ inl^{-1}, post \circ inl^{-1})$ refines and is refined by $(I, pre, post)$. Mutatis mutandis for $(inr(J), pre' \circ inr^{-1}, post' \circ inr^{-1})$ and $(J, pre', post')$. Now the result follows from Lemma 9.1. □

The join given by this proof is $(I + J, p, q)$ where $p$ and $q$ satisfy $p_{(i,0)} = pre_i$, $p_{(j,1)} = pre'_j$, $q_{(i,0)} = post_i$, and $q_{(j,1)} = post'_j$ for all $i \in I$ and $j \in J$.

We sometimes use the symbol $\sqcup$ as a function which yields some chosen join, say, the one in the Lemma. But joins are only unique up to the equivalence $\simeq$ associated with the preorder relation $\sqsupseteq$.

The join of specifications is expressed directly by the **also** syntax in JML. For specifications in two-state form, desugaring to a single specification is straightforward as shown by Lemma 9.7 in the sequel. For general specifications, using **forall** in JML, application of Lemma 9.2 is less straightforward.

*Example* 9.3. Consider the method specifications in Fig. 8, which are part of a class BScale with fields x and b. Recall the interpretation of **forall** in Example 5.6.

If the language had disjoint sum types, the construction in Lemma 9.2 would correspond to a single specification as suggested in the top part of Fig. 9. Consider what it means for an implementation to satisfy this specification. For a given initial state there are only two values of oxb for which the precondition is satisfied: inl(x) and inr(b). The value inl(x) forces a correct implementation to increase x by d and the value inr(b) forces a correct implementation to invert b. That is, a correct implementation must satisfy both postconditions.

For a language like Java without disjoint sums, another formulation is needed. One is suggested in the bottom part of Fig. 9. It uses a boolean tag to encode the sum in some sense. To confirm that this is an equivalent specification, note that it is modeled by the following alternative construction to Lemma 9.2. Given specifications $(I, pre, post)$ and $(J, pre', post')$ of type $\Gamma \rightsquigarrow \Gamma'$, we use index set $\{0, 1\} \times I \times J$ in and $p$ and $q$ defined by

$$p_{(0,i,j)} = pre_i \qquad q_{(0,i,j)} = post_i \qquad p_{(1,i,j)} = pre'_i \qquad q_{(1,i,j)} = post'_j$$

```
class BScale extends Object {
  public x: int;
  public b: bool;

  meth change(d: int)
     forall ox: int;
     requires ox = x ∧ 0 < d ∧ d < 300;
     ensures exc = null ∧ x = ox + d;
   also
     forall ob: bool;
     requires ob = b;
     ensures exc = null ∧ b = ¬ ob;
```

Fig. 8. Two specifications (separated by **also**) of a change method for class BScale.

```
forall oxb: int+bool;
requires cases oxb of
         | inl(ox): ox = x ∧ 0 < d ∧ d < 300;
         | inr(ob): ob = b;
ensures cases oxb of
         | inl(ox): exc = null ∧ x = ox + d;
         | inr(ob): exc = null ∧ b = ¬ ob;

_____

forall ox: int, ob: bool, tag: bool;
requires (tag ∧ ox = x ∧ 0 < d ∧ d < 300) ∨ (not tag ∧ ob = b);
ensures (tag ∧ exc = null ∧ x = ox + d) ∨ (not tag ∧ exc = null ∧ b = ¬ ob);
```

Fig. 9. Two ways of expressing the join of the specifications of method change in Fig. 8.

The specification $(\{0,1\} \times I \times J, p, q)$ essentially joins $J$-many copies of $(I, pre, post)$ and $I$-many copies of $(J, pre', post')$. It can be shown to be equivalent to the specification in Lemma 9.2 by expanding the definition of satisfaction. □

In case of joining specifications obtained from several interface/class declarations, the example generalizes nicely by using a tag that ranges over type names.

The join of satisfiable specifications need not be satisfiable.

*Example* 9.4. The simple specifications $(true, \mathsf{res} = 0)$ and $(true, \mathsf{res} = 1)$ join to $(true, false)$.

As noted following Lemma 5.16, satisfiability of a specification given by formulas is equivalent to validity of a closed formula derived from the specification.

A straightforward use of Lemma 9.2 and definitions yields the following.

Corollary 9.5 (joins and subtypes). If *spec* and *spec'* have type $\Gamma \rightsquigarrow \Gamma'$, and $T \leq \Gamma\,\mathsf{self}$, then $(spec \sqcup spec')\lfloor^* T \simeq (spec\lfloor^* T) \sqcup (spec'\lfloor^* T)$ and $(spec \sqcup spec')\lfloor T \simeq (spec\lfloor T) \sqcup (spec'\lfloor T)$.

Using the specific construction in the Lemma, we actually get equality of specifications here. But for our purposes equivalence is enough. A further corollary is that both $\lfloor^* T$ and $\lfloor T$ distribute over the join of any finite set of specifications.

We conclude this subsection by considering the joins of specifications that are in two-state form (Def. 5.4). Line $(*)$ in the following proof shows how the definition arises as an extremal solution when

considering only two-state specifications. The result has a simple reading in terms of specifications given by formulas: the join is expressed by propositional combinations of the given specifications.

LEMMA 9.6 (JOIN AMONG SPECIFICATIONS WITH TWO-STATE POSTCONDITIONS).
Let $(pre, post)$ and $(pre', post')$ be specifications of type $\Gamma \rightsquigarrow \Gamma'$ in two-state form. Define $P, R$ by

$$P = pre \cup pre' \quad \text{and} \quad R = (\neg old(pre) \cup post) \cap (\neg old(pre') \cup post') \tag{42}$$

Then for any $Q, S$ we have $\langle\!\langle Q, S \rangle\!\rangle \sqsupseteq \langle\!\langle P, R \rangle\!\rangle$ iff $\langle\!\langle Q, S \rangle\!\rangle \sqsupseteq \langle\!\langle pre, post \rangle\!\rangle$ and $\langle\!\langle Q, S \rangle\!\rangle \sqsupseteq \langle\!\langle pre', post' \rangle\!\rangle$.

PROOF. Consider any $Q, S$. If $\langle\!\langle Q, S \rangle\!\rangle$ is unsatisfiable the refinements all hold trivially. Otherwise

$\langle\!\langle Q, S \rangle\!\rangle \sqsupseteq \langle\!\langle P, R \rangle\!\rangle$
$\iff$ Cor. 5.20, definition of $R$, satisfiability of $\langle\!\langle Q, S \rangle\!\rangle$
$P \subseteq Q \ \wedge \ old(P) \cap S \subseteq (\neg old(pre) \cup post) \cap (\neg old(pre') \cup post')$
$\iff$ set theory
$P \subseteq Q \ \wedge \ S \subseteq \neg old(P) \cup ((\neg old(pre) \cup post) \cap (\neg old(pre') \cup post'))$
$\iff$ definition of $P$; distribute $\cup$ over $\cap$
$P \subseteq Q \ \wedge \ S \subseteq (\neg old(pre \cup pre') \cup \neg old(pre) \cup post) \cap (\neg old(pre \cup pre') \cup \neg old(pre') \cup post'))$
$\iff$ def $P$; sets, using $\neg old(pre \cup pre') \subseteq \neg old(pre)$ from $old(pre) \subseteq old(pre \cup pre')$ and sym.
$pre \cup pre' \subseteq Q \ \wedge \ S \subseteq (\neg old(pre) \cup post) \cap (\neg old(pre') \cup post')$ $(*)$
$\iff$ set theory
$pre \subseteq Q \ \wedge \ pre' \subseteq Q \ \wedge \ old(pre) \cap S \subseteq post \ \wedge \ old(pre') \cap S \subseteq post'$
$\iff$ Cor. 5.20 twice, satisfiability of $\langle\!\langle Q, S \rangle\!\rangle$
$\langle\!\langle Q, S \rangle\!\rangle \sqsupseteq \langle\!\langle pre, post \rangle\!\rangle \ \wedge \ \langle\!\langle Q, S \rangle\!\rangle \sqsupseteq \langle\!\langle pre', post' \rangle\!\rangle$

□

For the join among general specifications, our focus in the sequel, Eq. (42) serves as well.

LEMMA 9.7 (JOIN OF TWO-STATE SPECIFICATIONS). Let $(pre, post)$ and $(pre', post')$ be specifications of type $\Gamma \rightsquigarrow \Gamma'$ in two-state form. Then $\langle\!\langle P, R \rangle\!\rangle$ from (42) is a join of $\langle\!\langle pre, post \rangle\!\rangle$ and $\langle\!\langle pre', post' \rangle\!\rangle$.

PROOF. First we note the following direct consequence of Lemma 5.17(b). Consider any specification $(J, r, s)$ and any specification $(Q, S)$ in two-state form. Then

$$(J, r, s) \sqsupseteq \langle\!\langle Q, S \rangle\!\rangle \quad \text{iff} \quad \langle\!\langle (J, r, s)\dagger \rangle\!\rangle \sqsupseteq \langle\!\langle Q, S \rangle\!\rangle \tag{43}$$

Now we prove that $\langle\!\langle P, R \rangle\!\rangle \simeq \langle\!\langle pre, post \rangle\!\rangle \sqcup \langle\!\langle pre', post' \rangle\!\rangle$ by observing for any $(J, r, s)$

$(J, r, s) \sqsupseteq \langle\!\langle pre, post \rangle\!\rangle \wedge (J, r, s) \sqsupseteq \langle\!\langle pre', post' \rangle\!\rangle$
$\iff$ use (43) twice
$\langle\!\langle (J, r, s)\dagger \rangle\!\rangle \sqsupseteq \langle\!\langle pre, post \rangle\!\rangle \wedge \langle\!\langle (J, r, s)\dagger \rangle\!\rangle \sqsupseteq \langle\!\langle pre', post' \rangle\!\rangle$
$\iff$ Lemma 9.6
$\langle\!\langle (J, r, s)\dagger \rangle\!\rangle \sqsupseteq \langle\!\langle P, R \rangle\!\rangle$
$\iff$ by (43)
$(J, r, s) \sqsupseteq \langle\!\langle P, R \rangle\!\rangle$

□

### 9.2. Specification inheritance

Earlier sections of this paper use a specification table to model the "effective specifications" used in reasoning about method invocations and method implementations. Here, we consider the problem of taking some specifications that have been declared by the programmer and obtaining suitable ones for reasoning. More specifically, we consider a technique to impose behavioral subtyping by *fiat*. We also consider techniques for completing a partial specification table. The techniques are all forms of specification inheritance.

There are other issues in going from declared specifications to effective ones, e.g., desugaring of modifies clause or other specification features that may have class-specific semantics. Here we continue to treat specifications semantically, in order to focus on the single issue of behavioral subtyping.

A related consideration is that in practice, method specifications are usually only declared together with method implementations and in interfaces, but not in subclasses where a method is inherited.

That provides a partial specification table $ST$, where $ST(K, m)$ is only defined for $K$ that declares an implementation of $m$. We may as well treat it as a specification table where, if $m$ is inherited in $K$ from $L$, $ST(K, m)$ is the bottom specification that imposes no constraint at all (see Remark 5.2). A natural alternative is to 'inherit' the specification by defining $ST(K, m)$ to be $ST(L, m)$, as discussed in Sect. 9.3. It turns out that using the bottom specification achieves the same effect, once one imposes behavioral subtyping by specification inheritance, because bottom is the identity of join.

*Specification inheritance.* There is a more complicated problem than filling in specifications where methods are inherited, namely, the problem that explicitly declared specifications may not have behavioral subtyping. The specifier may be focusing on the special features of the subclass and perhaps imagining that the implementation will be required to satisfy all supertype specifications. The problem is how to impose that requirement in a minimal way. In our formulation, it is the problem of deriving a specification table that has behavioral subtyping from one that might not. We proceed by defining three variations on specification inheritance, guided by three desiderata for the derived specification table:

(D1) It should refine the given one, to preserve the intent of the original specifications.
(D2) It should have behavioral subtyping.
(D3) It should be the least refined one with these properties, to avoid imposing unnecessary constraints on implementations.

Refinement of class tables is defined pointwise: $ST' \sqsupseteq ST$ iff $ST'(T, m) \sqsupseteq ST(T, m)$ for all $T, m$. The significance is that $ST' \sqsupseteq ST$ implies that $ST'$ proves more (client) programs correct, i.e., $ST', \mathcal{P} \models^{\mathcal{S}} \mathit{spec}$ implies $ST, \mathcal{P} \models^{\mathcal{S}} \mathit{spec}$ for all $\mathcal{P}, \mathit{spec}$ (as follows from the definition of $\models^{\mathcal{S}}$) and similarly for $\models^{\mathcal{D}}$ and for modular verification (as follows from monotonicity of predicate transformer semantics).

*Definition* 9.8 (*inheriting specifications,* $\widehat{ST}, \widetilde{ST}, \breve{ST}$). Let $ST$ be a specification table. Define the specification table $\widehat{ST}$ as follows. For each class $K$ or interface $I$, with method $m$, we define $\widehat{ST}$ as follows:

$$\begin{array}{lll} \widehat{ST}(K, m) & = & \sqcup\{ST(T, m)\lfloor^* K \mid m \in \mathit{Meths}\ T \wedge K \leq T\}, \text{ if } K \in \mathit{ClassName} \\ \widehat{ST}(I, m) & = & ST(I, m), \qquad\qquad\qquad\qquad\qquad\quad \text{if } I \in \mathit{InterfaceName} \end{array}$$

Similarly, define $\widetilde{ST}$ by taking joins at all ref types $U$, not just classes:

$$\widetilde{ST}(U, m) = \sqcup\{ST(T, m)\lfloor^* U \mid m \in \mathit{Meths}\ T \wedge U \leq T\} \tag{44}$$

Finally, define $\breve{ST}$ like $\widetilde{ST}$ but with $\lfloor U$ for $\lfloor^* U$:

$$\breve{ST}(U, m) = \sqcup\{ST(T, m)\lfloor U \mid m \in \mathit{Meths}\ T \wedge U \leq T\}$$

We proceed to show that $\widehat{ST}$ is the one that satisfies the three desiderata. A fourth desideratum introduced later favors $\widetilde{ST}$.

Theorem 9.9. For any $ST$ we have $\widehat{ST} \sqsupseteq ST$ and $\widetilde{ST} \sqsupseteq ST$.

Proof. We need $\widehat{ST}(T, m) \sqsupseteq ST(T, m)$ for all $T$ and $m$. This is immediate from the definition, in case $T$ is an interface (using reflexivity of $\sqsupseteq$). In the case that $T$ is a class, say $K$, we have $ST(K, m)\lfloor^* K = ST(K, m)$ by a remark following Def. 5.9. Thus $ST(K, m)$ is in $\{ST(T, m)\lfloor^* K \mid m \in \mathit{Meths} T \wedge K \leq T\}$, hence we have $\sqcup\{ST(T, m)\lfloor^* K \mid m \in \mathit{Meths} T \wedge K \leq T\} \sqsupseteq ST(K, m)$.

The proof of $\widetilde{ST} \sqsupseteq ST$ is very similar. □

By contrast, $\breve{ST}$ does not refine $ST$, because in general $\mathit{spec}\lfloor K$ is not a refinement of $\mathit{spec}$ (as the precondition is strengthened to states where self has exact type $K$).

Recall that robust behavioral subtyping (Def. 8.2) strengthens behavioral subtyping by using $\sqsupseteq^{*K}$ (based on $\lfloor^* K$) in place of $\sqsupseteq^K$ (based on $\lfloor K$).

Theorem 9.10 (specification inheritance and behavioral subtyping). Both $\widehat{ST}$ and $\widetilde{ST}$ have robust behavioral subtyping.

PROOF. For $\widehat{ST}$ to have robust behavioral subtyping means that

$$\forall U, K, m \cdot K \leq U \wedge m \in \mathit{Meths}\ U \Rightarrow \widehat{ST}(K, m) \sqsupseteq^{*K} \widehat{ST}(U, m)$$

To prove it, we make a case distinction on whether $U$ is a class or interface, in accord with the definition of $\widehat{ST}$. (Recall that $K$ ranges over classes only.)

For any $U, K, m$ such that $U$ is a class, $K \leq U$, and $m \in \mathit{Meths}\ U$, observe that

$$\begin{array}{ll} & \widehat{ST}(K, m) \sqsupseteq^{*K} \widehat{ST}(U, m)) \\ \iff & \quad \text{def. } \widehat{ST},\ U \text{ is class, omit range } T, V \text{ with } m \in \mathit{Meths}\ T,\ m \in \mathit{Meths}\ V \\ & \sqcup\{ST(T, m)\lfloor^* K \mid K \leq T\} \sqsupseteq^{*K} \sqcup\{ST(V, m)\lfloor^* U \mid U \leq V\} \\ \iff & \quad \text{definition of } \sqsupseteq^{*K} \\ & \sqcup\{ST(T, m)\lfloor^* K \mid K \leq T\} \sqsupseteq (\sqcup\{ST(V, m)\lfloor^* U \mid U \leq V\})\lfloor^* K \\ \iff & \quad \text{Corollary 9.5 } \lfloor^* K \text{ distributes } \sqcup, \text{ Lemma 5.13 with } K \leq U \\ & \sqcup\{ST(T, m)\lfloor^* K \mid K \leq T\} \sqsupseteq \sqcup\{ST(V, m)\lfloor^* K \mid U \leq V\} \\ \iff & \quad \text{the join property} \\ & \forall V \cdot U \leq V \Rightarrow \sqcup\{ST(T, m)\lfloor^* K \mid K \leq T\} \sqsupseteq ST(V, m)\lfloor^* K \end{array}$$

The last line is true, because for any $V$ such that $U \leq V$, we have $K \leq V$ because $K \leq U$. So $ST(K, m)\lfloor^* K$ is in $\{ST(K, m)\lfloor^* K \mid K \leq T\}$.

The other case is that $U$ is an interface. Assuming $K \leq U$, and $m \in \mathit{Meths}\ U$ we observe that

$$\begin{array}{ll} & \widehat{ST}(K, m) \sqsupseteq^{*K} \widehat{ST}(U, m)) \\ \iff & \quad \text{definition of } \widehat{ST},\ U \text{ is interface, omit that } T \text{ ranges over reftypes} \\ & \sqcup\{ST(T, m)\lfloor^* K \mid K \leq T\} \sqsupseteq^{*K} ST(U, m) \\ \iff & \quad \text{definition of } \sqsupseteq^{*K} \\ & \sqcup\{ST(T, m)\lfloor^* K \mid K \leq T\} \sqsupseteq ST(U, m)\lfloor^* K \end{array}$$

The last line is true because $ST(U, m)\lfloor^* K$ is in $\{ST(T, m)\lfloor^* K \mid K \leq T\}$.

For $\widetilde{ST}$ the proof is similar, but there is not a separate case for interface types. □

Although $\breve{ST}$ does not refine $ST$ (our desideratum (D1)), it does have behavioral subtyping. The proof proceeds similarly to the one above; but for the fourth line, after Corollary 9.5 was used, we get

$$\sqcup\{ST(T, m)\lfloor K \mid K \leq T\} \sqsupseteq \sqcup\{ST(V, m)\lfloor U\lfloor K \mid U \leq V\}$$

In case $U$ is not $K$, $ST(V, m)\lfloor U\lfloor K$ is equivalent to the bottom specification (identity of join) because the preconditions become empty. So the right side simplifies to the same as the left side.

As join does not preserve satisfiability (cf. Example 9.4), $\widehat{ST}$ (and $\widetilde{ST}$, $\breve{ST}$) need not be satisfiable even if $ST$ is. To address desideratum (D3) that the derived specification table be minimal, we state a result that applies when all specifications are satisfiable. The proof shows something slightly stronger, on a per-class basis, which we refrain from stating formally.

THEOREM 9.11. If $\widehat{ST}$ is satisfiable then it is the least refinement of $ST$ that is satisfiable and has robust behavioral subtyping.

PROOF. Suppose $\widehat{ST}$ is satisfiable. By Thm. 9.9 and Thm. 9.10, $\widehat{ST}$ refines $ST$ and has robust behavioral subtyping. Consider any satisfiable $ST'$ such that $ST' \sqsupseteq ST$ and $ST'$ has robust behavioral subtyping. To show $ST' \sqsupseteq \widehat{ST}$, observe for any $K, m$

$$\begin{array}{ll} & ST'(K, m) \sqsupseteq \widehat{ST}(K, m) \\ \iff & \quad \text{definition of } \widehat{ST} \\ & ST'(K, m) \sqsupseteq \sqcup\{ST(U, m)\lfloor^* K \mid K \leq U\} \\ \iff & \quad \text{the join property} \\ & \forall U \cdot K \leq U \Rightarrow ST'(K, m) \sqsupseteq ST(U, m)\lfloor^* K \\ \iff & \quad \text{definition} \\ & \forall U \cdot K \leq U \Rightarrow ST'(K, m) \sqsupseteq^{*K} ST(U, m) \\ \Leftarrow & \quad \text{transitivity Lemma 5.15 (and } \sqsupseteq^{*U} \text{ is } \sqsupseteq\text{), satisfiability of } ST'(U, m) \\ & \forall U \cdot K \leq U \Rightarrow ST'(K, m) \sqsupseteq^{*K} ST'(U, m) \wedge ST'(U, m) \sqsupseteq ST(U, m) \end{array}$$

We have $ST'(K,m) \sqsupseteq^{*K} ST'(U,m)$ because $ST'$ has robust behavioral subtyping and $ST'(U,m) \sqsupseteq ST(U,m)$ by $ST' \sqsupseteq ST$. $\square$

*Remark* 9.12. If we assume only that $ST'$ has behavioral subtyping and is satisfiable, we can't show $ST' \sqsupseteq \widehat{ST}$. By behavioral subtyping, we have for any reftype $U$, any $K \leq U$, and any $m$, that $ST'(K,m) \sqsupseteq^K ST'(U,m)$. By $ST' \sqsupseteq ST$ we have $ST'(U,m) \sqsupseteq ST(U,m)$. Thus by quasi-transitivity (Lemma 5.14) we get $ST'(K,m) \sqsupseteq^K ST(U,m)$. Hence by the join property we get $ST'(K,m) \sqsupseteq \sqcup\{ST(U,m)\lfloor K \mid K \leq U\}$. Here's the rub: $\widehat{ST}$ is the join of $ST(U,m)\lfloor^* K$ and in general the refinement $ST(U,m)\lfloor^* K \sqsupseteq ST(U,m)\lfloor K$ is a proper refinement. $\square$

By definition we have $\widetilde{ST} \sqsupseteq \widehat{ST}$ and in general the refinement is strict. In light of Thm. 9.11, $\widetilde{ST}$ is not the least refinement of $ST$ that has behavioral subtyping. So it is only $\widehat{ST}$ that fulfills the three desiderata. However, there is a fourth desideratum that favors $\widetilde{ST}$, as we now explain at some length.

LEMMA 9.13 (INTERFACE IRRELEVANCE). Let $ST$ and $ST'$ be such that $ST(K,m) = ST'(K,m)$ for all $K$. Then $ST, \mathcal{P} \models^{\mathcal{D}} \mathit{spec}$ iff $ST', \mathcal{P} \models^{\mathcal{D}} \mathit{spec}$, for any *spec* and phrase in context $\mathcal{P}$.

PROOF. By definition, $\eta \models ST$ iff $\eta \models ST'$ for any normal method environment $\eta$, as interface types are irrelevant. And $\mathcal{D}[\![\mathcal{P}]\!]$ is only applied to normal method environments, by definition of $\models^{\mathcal{D}}$. $\square$

Refining a specification table proves more client programs, in the sense of modular verification. That is, suppose $ST' \sqsupseteq ST$. Using Lemma 7.7 we have $ST' \sqsupseteq ST$ iff $\{\!|ST'|\!\} \sqsupseteq \{\!|ST|\!\}$. Then by monotonicity in the method environment, Lemma 7.2, we have $\mathcal{S}\{\!|\Gamma \vdash C|\!\}(\{\!|ST|\!\}) \sqsupseteq \{\!|\mathit{spec}|\!\} \Rightarrow \mathcal{S}\{\!|\Gamma \vdash C|\!\}(\{\!|ST'|\!\}) \sqsupseteq \{\!|\mathit{spec}|\!\}$. In short: if $ST' \sqsupseteq ST$ then modular correctness of $C$ under $ST$ implies its modular correctness under $ST'$. Refining a specification table also proves more client programs in the sense of modular correctness, simply because there are fewer satisfying method environments over which to quantify in Def. 6.2(20).

Of course we do not want to refine a given $ST$ arbitrarily: that makes it harder —even impossible— for the method environment to satisfy $ST$. Rather, we suggest a fourth desideratum for specification inheritance:

(D4) While imposing the least constraint on the class table —desideratum (D3)— the inherited specifications should provide the most complete modular verification.

In light of Lemma 9.13, let us consider refining $ST$ by changing only its interface specifications, while maintaining behavioral subtyping. For any $ST$, define $\mathit{Complete}(ST)$ by taking joins at interface types only. So $ST$ and $\mathit{Complete}(ST)$ are the same at any class type, and thus are satisfied by the same method environments. But for interface type $U$ we define $\mathit{Complete}(ST)(U,m)$ the same way as $\widetilde{ST}(U,m)$, i.e., Eq. (44) (for any $m \in \mathit{Meths}\ U$). This can be seen as a kind of completion with respect to modular verification, because $\mathit{Complete}(ST) \sqsupseteq ST$ which means $\mathit{Complete}(ST)$ verifies more programs.

LEMMA 9.14. For any $ST$, if $ST$ has behavioral subtyping, or robust behavioral subtyping, then so does $\mathit{Complete}(ST)$.

PROOF. Suppose $ST$ has behavioral subtyping. For any $K, m, U$ with $K \leq U$ we need $\mathit{Complete}(ST)(K,m) \sqsupseteq^K \mathit{Complete}(ST)(U,m)$ which is equivalent to $ST(K,m) \sqsupseteq^K \mathit{Complete}(ST)(U,m)$. In case $U$ is a class type, this is immediate by definition of *Complete*.

In case $U$ is an interface, behavioral subtyping is the first line of

$$\begin{array}{ll} & ST(K,m) \sqsupseteq^K \mathit{Complete}(ST)(U,m) \\ \iff & \quad \text{def } \mathit{Complete} \\ & ST(K,m) \sqsupseteq^K \sqcup\{ST(T,m)\lfloor^* U \mid m \in \mathit{Meths}\ U \wedge U \leq T\} \\ \iff & \quad \text{def } \sqsupseteq^K \text{ and } \lfloor K \text{ distributes over } \sqcup \\ & ST(K,m) \sqsupseteq \sqcup\{ST(T,m)\lfloor^* U \lfloor K \mid m \in \mathit{Meths}\ U \wedge U \leq T\} \end{array}$$

Consider the last line: For any $T$ such that $K \leq U \leq T$, by behavioral subtyping we have $ST(K,m) \sqsupseteq^K ST(T,m)$ that is, $ST(K,m) \sqsupseteq ST(T,m)\lfloor K$. But this is equivalent to $ST(K,m) \sqsupseteq$

$ST(T,m)\lfloor^* U \lfloor K$ by Lemma 5.13. Hence, as $ST(K,m)$ is above every element of the join, it is above the join.

This also works for robust behavioral subtyping, i.e., we get $\mathit{Complete}(ST)(K,m) \sqsupseteq^{*K} \mathit{Complete}(ST)(U,m)$ using that $ST(T,m)\lfloor^* U \lfloor^* K = ST(T,m)\lfloor^* K$ □

The following property of the *Complete* operator shows that in fact $\widetilde{ST}$ is the most satisfactory notion of specification inheritance. The property refers to the equivalence $\simeq$ associated with the preorder $\sqsupseteq$. We do not have literal equality, because joins are only defined up to equivalence.

PROPOSITION 9.15. For any $ST$, $\mathit{Complete}(\widehat{ST}) \simeq \widetilde{ST}$.

PROOF. For class types, all three of $\mathit{Complete}(\widehat{ST})$, $\widetilde{ST}$, and $\widehat{ST}$ are the same. So we consider any interface type $U$ and any $m \in \mathit{Meths}\, U$. By definition of *Complete* we have

$$\mathit{Complete}(\widehat{ST})(U,m) = \sqcup\{\widehat{ST}(T,m)\lfloor^* U \mid m \in \mathit{Meths}\, T \wedge U \leq T\} \tag{45}$$

Aiming for the defining condition of $\widetilde{ST}(U,m)$, Eq. (44), let us expand a typical term in the set displayed above. For any $T$ with $m \in \mathit{Meths}\, T$ and $U \leq T$ we have

$$\begin{array}{ll} & \widehat{ST}(T,m)\lfloor^* U \\ = & \quad \text{definition of } \widehat{ST} \\ & (\sqcup\{ST(V,m)\lfloor^* T \mid m \in \mathit{Meths}\, V \wedge T \leq V\})\lfloor^* U \\ \simeq & \quad \lfloor^* U \text{ distributes over joins} \\ & \sqcup\{ST(V,m)\lfloor^* T\lfloor^* U \mid m \in \mathit{Meths}\, V \wedge T \leq V\} \\ = & \quad \text{Lemma 5.13} \\ & \sqcup\{ST(V,m)\lfloor^* U \mid m \in \mathit{Meths}\, V \wedge T \leq V\} \end{array}$$

Now (45) joins these terms over all $T$ with $U \leq T$, and just above we expanded a term to a join over all $V$ with $T \leq V$. So the nested join amounts to a single join over all $V$ with $U \leq V$ —essentially Eq. (44). □

In summary, it is $\widetilde{ST}$, defined in Eq. (44), that satisfies desiderata (D1–D4), and it has robust behavioral subtyping.

### 9.3. Remarks on inheriting specifications

Let us consider a slightly different perspective, to clarify the significance of robust behavioral subtyping. Suppose we want to add a class $K$ to a classtable, with superclass $L$, and consider a method $m$ that is inherited from $L$. In this situation, the semantics manifests inheritance in the definition of the method environment:

$$\eta(K,m) = \mathit{restrict}(K, \eta(L,m))$$

where $\mathit{restrict}(K,-)$ restricts the domain of the state transformer $\eta(L,m)$ to states where self's type is $\leq K$ (see Sects. 4.6 and A.2). If for the moment we suppose that $ST(L,m)$ is the effective specification of $m$ in $L$, an obvious choice is to define $ST(K,m)$ by

$$ST(K,m) = ST(L,m)\lfloor^* K \tag{46}$$

This has the virtue that

$$\eta(L,m) \models ST(L,m) \quad \text{implies} \quad \mathit{restrict}(K,\eta(L,m)) \models ST(L,m)\lfloor^* K$$

(The restriction $\lfloor^* K$ is necessary; an implementation at subclass $K$ won't be able to cope with pre-states where self has type $L$ but not $K$.) That is, the inherited implementation automatically satisfies $ST(L,m)\lfloor^* K$ so we obtain a suitable specification for use by callers while not imposing any additional proof obligation.

Note that $\mathit{restrict}(K,\eta(L,m))$ will also satisfy $ST(L,m)\lfloor K$, simply because $ST(L,m)\lfloor^* K \sqsupseteq ST(L,m)\lfloor K$ (see Eq. (17) in Sect. 5.2). But we may as well choose the stronger, more informative $ST(L,m)\lfloor^* K$.

There is a second justification for (46). Consider any $T$ with $K < L < T$, and suppose behavioral subtyping holds at $L$ for $T$, i.e., $ST(L,m) \sqsupseteq^L ST(T,m)$ and that $ST(L,m)$ is satisfiable. We would like behavioral subtyping to hold at $K$ for $T$. With the preferred definition (46), we get

$ST(K,m) \sqsupseteq^{*K} ST(L,m)$ as mentioned earlier, and thus from $ST(L,m) \sqsupseteq^{*L} ST(T,m)$ we get $ST(K,m) \sqsupseteq^{*K} ST(T,m)$ by quasi-transitivity (Lemma 5.15) and thus behavioral subtyping at $K$ for $T$: $ST(K,m) \sqsupseteq^{K} ST(T,m)$ (by Lemma 5.12). So if $K$ inherits the implementation of $m$ and its specification from $L$ and robust behavioral subtyping holds for $m$ in $L$ then it holds for $m$ in $K$ —provided that $K$ does not implement additional interfaces. The proviso ensures that supertypes of $K$ are supertypes of $L$. However, behavioral subtyping at $K$ for $T$, i.e., $ST(K,m) \sqsupseteq^{K} ST(T,m)$, does not follow from $ST(K,m) \sqsupseteq^{*K} ST(L,m)$ and $ST(L,m) \sqsupseteq^{L} ST(T,m)$ (cf. discussion following Lemma 5.15).

Technically, this discussion adds nothing to the previous subsection. If $ST$ is the original specification table before specification inheritance, we can define $ST(K,m)$ as the bottom specification. Then $\widetilde{ST}(K,m)$ will be the same as $\widetilde{ST}(L,m)\lfloor^{*}K$ under the proviso that $K$ does not implement additional interfaces.

*Example* 9.16. As an example of the proviso, consider the class table for Fig. 2, and suppose we add a class $K$ that extends WeightLoss and inherits method lose. If we 'inherit' the specification too, in the sense of defining $ST(K,\mathsf{lose}) = ST(\mathsf{WeightLoss},\mathsf{lose})$, we automatically get behavioral subtyping for $ST(K,\mathsf{lose})$. However, suppose we change Fig. 2 so that WeightLoss only implements Tracker. Let $K$ inherit lose from WeightLoss and declare that $K$ implements Track, which does not satisfy the proviso above. Then we do not get behavioral subtyping at $K$ if we define $ST(K,\mathsf{lose}) = ST(\mathsf{WeightLoss},\mathsf{lose})$; for behavioral subtyping we need to join the specification from Track.

## 10. ADAPTING THE RESULTS TO PARTIAL CORRECTNESS

We have chosen to develop the theory for total correctness specifications, which brings to light the satisfiability requirements in the main results. All partial correctness specifications are satisfied by an always divergent program. Essentially the same results hold for partial correctness. We eschew the extra formalism that would be needed to generalize the results to encompass both total and partial correctness. Instead, in this section we adapt the theory to partial correctness, retaining the same program semantics as in Sect. 4 and Sect. 7. We retain the same form for specifications.

The interpretation of specifications, i.e., the definition of satisfaction (Def. 5.3), changes in the obvious way. In place of (15) we define the "liberal" satisfaction relation, $\models_l$, on simple specifications:

$$\varphi \models_l (pre, post) \qquad \text{iff} \qquad \forall \sigma \cdot \sigma \in pre \;\Rightarrow\; \varphi(\sigma) \in post \cup \{\bot\}.$$

This is lifted to general specifications by universal quantification over indices just as in Def. 5.3.. In place of the semantic *wp* function, Eq. (26), the weakest liberal precondition function *wlp* is used:

$$wlp(\varphi)(post) = \{\sigma \mid \varphi(\sigma) \in post \cup \{\bot\}\}. \tag{47}$$

Here *post* ranges over predicates, which as before do not contain $\bot$.

Because the form of specifications is not changed, several definitions in Sect. 5 are still applicable, e.g., Def. 5.4 for specifications in two-state form and Def. 5.9 of subtype restriction. For partial correctness specifications with two-state postconditions, several variations can be found in the literature, including the use of a single relation. We shall interpret a specification $(P,R)$ via $\langle\!\langle P,R\rangle\!\rangle$ (see Def. 5.4). Expanding definitions we have that $\varphi \models_l \langle\!\langle P,R\rangle\!\rangle$ iff $\sigma \in P \Rightarrow \varphi(\sigma) = \bot \vee (\sigma,\varphi(\sigma)) \in R$ for all $\sigma$. Thus $\varphi$ satisfies $\langle\!\langle P,R\rangle\!\rangle$ iff it satisfies $\langle\!\langle P, old(P) \cap R\rangle\!\rangle$. Such considerations suggest restricting specifications to one or another "normal form", as is done in various systems. We define one for later use.

A specification $(I, pre, post)$ is ***broad*** iff for all $\sigma$ there is some $i$ with $\sigma \in pre_i$, i.e., the preconditions cover the state space. Define $broaden(I, pre, post)$ by choosing some value $\bullet \notin I$ and defining $broaden(I, pre, post) = (I \cup \{\bullet\}, pre', post')$ where $pre'_\bullet = \{\sigma \mid \neg(\exists i \cdot \sigma \in pre_i)\}$, $post'_\bullet = True$, and for $i \in I$, $pre'_i = pre_i$ and $post'_i = post_i$. Note that the simple specification $(\{\sigma \mid \neg(\exists i \cdot \sigma \in pre_i)\}, True)$ is satisfied, in the sense of partial correctness, by all state transformers. Thus $\varphi \models_l spec$ iff $\varphi \models_l broaden(spec)$ for each $\varphi$ and *spec*.

Refinement of predicate transformers is unchanged from Eq. (27). The intrinsic refinement order on specifications, $\sqsupseteq_l$, is defined just as in Def. 5.10 except using $\models_l$ in place of $\models$. Writing $\simeq_l$ for the equivalence associated with the preorder $\sqsupseteq_l$, we have

$$broaden(spec) \simeq_l spec \quad \text{for any } spec \tag{48}$$

owing to the observation above about *broaden* and $\models_l$. By contrast, for total correctness there is strict refinement $broaden(spec) \sqsupseteq spec$ in general, because $broaden(spec)$ requires termination from every initial state.

Most results and definitions carry over straightforwardly, with $\models_l$ and $\sqsupseteq_l$ in place of $\models$ and $\sqsupseteq$. Of course the characterization of satisfiable specifications, Lemma 5.16, is no longer meaningful. Results in Sect. 5 that hold for satisfiable specifications now hold for all specifications; an example is the quasi-transitivity Lemma 5.14. To adapt the characterization of refinement, Prop. 5.18, we need a slightly different characteristic formula which does not seem widely known but appears in Pierik [2006].

PROPOSITION 10.1 (CHARACTERIZATION OF REFINEMENT FOR PARTIAL CORRECTNESS). Suppose that $(J, pre', post')$ and $(I, pre, post)$ are specifications of type $\Gamma \rightsquigarrow \Gamma'$. The following are equivalent:

(a) $(J, pre', post') \sqsupseteq_l (I, pre, post)$
(b) $\forall i \in I, \sigma \in State(\Gamma) \cdot \sigma \in pre_i$
$\Rightarrow (\forall \tau \in State(\Gamma') \cdot (\forall j \in J \cdot \sigma \in pre'_j \Rightarrow \tau \in post'_j) \Rightarrow \tau \in post_i)$

The proof is in Appendix A.12. Note that (b) differs from Prop. 5.18(b) by omitting conjunct $\exists j \in J \cdot \sigma \in pre'_j$, in such a way that (b) here follows from the other. A broad specification is exactly one that satisfies $\exists j \in J \cdot \sigma \in pre'_j$, so for such specifications the characteristic formulas are equivalent.

A pleasing consequence is that, for specifications that are satisfiable in the sense of total correctness, $spec' \sqsupseteq_l spec$ follows from $spec' \sqsupseteq spec$. Because: If $spec'$ is satisfiable (for total corr.), $spec' \sqsupseteq spec$ iff the characteristic formula, Prop. 5.18(b), holds; that in turn implies Prop. 10.1(b) which is equivalent to $spec' \sqsupseteq_l spec$. This means that a single specification table can be used for both total and partial correctness, and it has behavioral subtyping for partial correctness if it has it for total. Even if the relations $\sqsupseteq_l$ and $\sqsupseteq$ coincided, however, we would not immediately get the partial-correctness versions of the main theorems for free: We want those results for all specifications, not just those satisfiable in the sense of total correctness; also, the meaning of supertype abstraction is different for $\models_l$.

Adapting the proof of Corollary 5.20 leads us to find that

$$\langle\!\langle P', R' \rangle\!\rangle \sqsupseteq_l \langle\!\langle P, R \rangle\!\rangle \quad \text{iff} \quad old(P) \cap R' \subseteq R \ \wedge \ old(P) \cap old(\neg P') \subseteq R \tag{49}$$

which drops $P \subseteq P'$ from Eq. (19) and adds a condition involving the complement, $\neg P'$, of $P'$. In case $P \subseteq P'$ holds, $old(P) \cap old(\neg P')$ is empty and the condition holds, so (19) is a sufficient but not necessary condition for the refinement.

Moving on to Sect. 6, the definitions of supertype abstraction and behavioral subtyping are unchanged except for using $\models_l$ and $\sqsupseteq_l$. Moving on to Sect. 7, Lemma 7.1 holds using *wlp* in place of *wp* —and, as noted earlier, without changing the semantics $\mathcal{S}\{\![-]\!\}$ or $\mathcal{D}\{\![-]\!\}$ of commands and expressions. Most of the proof goes through essentially unchanged. It is only the method call cases that involve the environment, and thus *wlp*. We consider method call in static dispatch semantics:

$$\begin{array}{ll}
& (r, h, s) \in wlp(\mathcal{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\dot{\eta}))(Q) \\
\iff & \quad \text{definition (47) of } wlp \\
& \mathcal{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\dot{\eta})(r, h, s) \in Q \cup \{\bot\} \\
\iff & \quad \text{definition of } \mathcal{S}[\![-]\!]\text{; simplify using } except(r, h, U, \mathsf{NullDeref}) \neq \bot \text{ by def. of } except \\
& \quad \textbf{if } s\,x = null \textbf{ then } except(r, h, U, \mathsf{NullDeref}) \in Q \\
& \quad \textbf{else let } T = \Gamma\,x \textbf{ in let } \overline{z} = formals(T, m) \textbf{ in} \\
& \quad \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } \dot{\eta}(T, m)(r, h, s_1) \in Q \cup \{\bot\} \\
\iff & \quad \text{definition of } wlp \\
& \quad \textbf{if } s\,x = null \textbf{ then } except(r, h, U, \mathsf{NullDeref}) \in Q \\
& \quad \textbf{else let } T = \Gamma\,x \textbf{ in let } \overline{z} = formals(T, m) \textbf{ in} \\
& \quad \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } (r, h, s_1) \in wlp(\dot{\eta}(T, m))(Q) \\
\iff & \quad \text{definition of } wlp(\dot{\eta}) \\
& \quad \textbf{if } s\,x = null \textbf{ then } except(r, h, U, \mathsf{NullDeref}) \in Q \\
& \quad \textbf{else let } T = \Gamma\,x \textbf{ in let } \overline{z} = formals(T, m) \textbf{ in} \\
& \quad \textbf{let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in } (r, h, s_1) \in wlp(\dot{\eta})(T, m)(Q) \\
\iff & \quad \text{definition of } \mathcal{S}\{\![-]\!\} \\
& (r, h, s) \in \mathcal{S}\{\![\Gamma \vdash x.m(\overline{y}) : U]\!\}(wlp(\dot{\eta}))(Q)
\end{array}$$

The predicate transformer semantics of specifications, Def. 7.3, does need to be changed, so that initial states that falsify the precondition are miraculous rather than abortive. We define $\{\![J, pre, post]\!\}_l$, for any $\sigma$ and $Q$, by

$$\sigma \in \{\![J, pre, post]\!\}_l(Q) \iff (\forall \tau \cdot (\forall i \cdot \sigma \in pre_i \Rightarrow \tau \in post_i) \Rightarrow \tau \in Q) \tag{50}$$

This differs from Def. 7.3 only by dropping the conjunct $\exists j \cdot \sigma \in pre_j$. The key Lemma 7.5 works for partial correctness, i.e., we have $\varphi \models_l spec$ iff $wlp(\varphi) \sqsupseteq \{\![spec]\!\}_l$. Its proof, in Appendix A.13, uses (48).

As mentioned earlier, Def. 7.9 of refinement for predicate transformers is unchanged, as is Eq. (32) that defines pointwise meet. A new proof is needed for the adaptation of Lemma 7.6, which becomes $\{\![J, pre, post]\!\}_l = (\dot{\sqcap}\varphi \mid \varphi \models (J, pre, post) \cdot wlp(\varphi))$. This is in Appendix A.14. For the remaining results in Sect. 7, the proofs adapt straightforwardly.

Moving on to Sect. 8, we already noted that the definitions are adapted by using $\models_l$ and $\sqsupseteq_l$ in place of their counterparts, retaining $\sqsupseteq$ for predicate transformers. In particular, weak supertype abstraction means $\mathcal{S}\{\![\mathcal{P}]\!\}(\{\![ST]\!\}_l) \sqsupseteq \{\![spec]\!\}_l$ implies $ST, \mathcal{P} \models_l^{\mathcal{D}} spec$ and strong supertype abstraction means $ST, \mathcal{P} \models_l^{\mathcal{S}} spec$ implies $ST, \mathcal{P} \models_l^{\mathcal{D}} spec$. Thm. 8.8 now says that $\mathcal{S}\{\![\mathcal{P}]\!\}(\{\![ST]\!\}_l) \sqsupseteq \{\![spec]\!\}_l$ implies $ST, \mathcal{P} \models_l^{\mathcal{S}} spec$ (and the same for $\mathcal{D}$). Thm. 8.10 now uses $\{\![ST]\!\}_l$ in place of $\{\![ST]\!\}$; its proof appeals to Lemma 7.10 which needs no change because it refers to refinement of predicate transformers. The remaining results in Sect. 8 require little or no change. For example, for the key property Eq. (37) of dynamic dispatch the proof unfolds definitions to the point of applying $\models_l$ but not expanding its definition, so the argument goes through unchanged. The key Lemma 8.13 holds for partial correctness. (The proof relies on characteristic formula given in Corollary 5.19, but it does not use the conjunct that is dropped in the adaptation to partial correctness.) In summary, in the setting of partial correctness, behavioral subtyping is equivalent to supertype abstraction.

Finally we consider specification inheritance. The constructions for joins for specifications, in Lemmas 9.1, 9.2, and Corollary 9.5 are unchanged except for using $\sqsupseteq_l$ and $\simeq_l$. To show that they are in fact joins, the proofs need to be revised to use $\sqsupseteq_l$ and $\models_l$, but there are no surprises. For join of specifications in two-state form, the definition Eq. (42) still works, although the proof of the join property is different in detail from the proof of Lemma 9.6. Adaptation of Lemma 9.7 is then straightforward. For the analysis of specification inheritance in Sect. 9.2, the proofs need essentially no change, as they rely on join properties.

In summary, the construction of joins, both for general and two-state specifications, is the same for both total and partial correctness. Programmer provided specifications that are suitable for both total and partial correctness may thus be inherited and ensure behavioral subtyping for both forms of correctness.

## 11. CONCLUSIONS

In summary, we have identified a notion of behavioral subtyping that is equivalent to supertype abstraction —which tells us what specifications are good for reasoning about pre-post properties of method calls. Separately, we identified a slightly stronger notion of behavioral subtyping, dubbed 'robust', that is better suited to inheritance of implementations, and which can be obtained from arbitrary declared specifications by specification inheritance. Both notions of behavioral subtyping are based on the intrinsic refinement order on specifications. For $S$ declared to be a subtype of $T$, and $m$ a method of $T$, behavioral subtyping requires the specification of $S$'s implementation of $m$ to refine that of $T$'s, for objects of exactly type $S$. By contrast, robust behavioral subtyping requires refinement for objects "of type $S$" in the usual sense of having dynamic type $S$ or a subtype. The robust form leads to more informative specifications, though it is slightly stronger than what is necessary for supertype abstraction. We also uncovered that, if $S$ is not an instantiable class then for soundness of supertype abstraction its specification of $m$ need not refine that of $m$ in $T$. Like robust subtyping, however, the refinement does provide the most complete information for reasoning about calls of $m$ at static type $S$, and it can be achieved by *fiat* using specification inheritance.

The idea of specification inheritance has been criticized by Findler and Felleisen as covering up design errors [Findler and Felleisen 2001; Findler et al. 2001]. The argument is that refinement of supertype specifications is the responsibility of the subtype's designer, and that the use of specification inheritance can turn design errors into assertion violations (especially when postconditions conflict), leading to confusing error messages. So in their methodology, specifications for methods must be

written explicitly and not synthesized by specification inheritance. Writing complete specifications for each method makes it possible to detect non-refinement as a design error, instead of turning such problems into unsatisfiable specifications as with specification inheritance. Part of this argument is the result of a technical difference in definitions of behavioral subtyping, which our work should clarify; Findler and Felleisen use the definition of Liskov and Wing, which is unnecessarily strong. But the other part of this argument is a methodological difference. Findler and Felleisen want to have complete specifications written explicitly by a person for each method, instead of having the specifier work (indirectly) on method specifications by writing deltas to inherited specifications. Further investigation would be worthwhile to better understand the software engineering issues pertaining to information hiding and reuse with the need to detect design flaws, which might lead to better tools and specification notations.

We have explored two notions of supertype abstraction, the strong form expressed in terms of correctness under hypotheses, and the weak form expressed in terms of axiomatic semantics. When quantified over all programs and specifications the two forms are equivalent. However, for a fixed program and specification, the 'strong' form is a strictly stronger property. This suggests the problem of giving a direct proof that (weak supertype abstr.) $\Rightarrow$ (strong supertype abstr.) without recourse to behavioral subtyping. Our proof of Thm. 8.15 uses an equivalence with behavioral subtyping, i.e., the implications (behavioral subtyping) $\Rightarrow$ (strong supertype abstr.) $\Rightarrow$ (weak supertype abstr.) $\Rightarrow$ (behavioral subtyping). We also give direct proofs of (strong supertype abstr.) $\Rightarrow$ (behavioral subtyping) and (behavioral subtyping) $\Rightarrow$ (weak supertype abstr.).

The characterization of refinement in terms of pre- and post-conditions depends on the form of specification, i.e., whether two-state postconditions are used or auxiliary variables. It also depends on whether partial or total correctness is considered. Our characterization for partial correctness seems to be novel. In the introduction we point out that the Liskov-Wing postcondition rule is stronger than necessary, and note that some but not all recent work adopts the better version. For partial correctness, the Liskov-Wing precondition rule is also unnecessarily strong. However, this may have little practical significance, and their precondition rule is used in most of the literature and in implemented systems.

We chose to develop the theory using a fairly elaborate programming language, in part to show that features like type cast and exceptions do not subvert the account of behavioral subtyping and supertype abstraction based on refinement. We leave it as an open problem to formulate an account that abstracts from the particulars of the programming language and also yields our results when instantiated for our language. Such an account might also encompass constructs from refinement calculus (e.g., unbounded nondeterminacy, angelic nondeterminacy, and specification statements).

It is common wisdom that subclasses should be behavioral subtypes, and also that behavioral subtyping sometimes needs to be violated in order to use inheritance merely as a way to reuse code. Parkinson and Bierman [2008] give such an example (adapted as our Example 8.5), which they describe as being at odds with behavioral subtyping. We have shown that by using a precondition with an exact test on the type of self, as in their specification, one in fact retains behavioral subtyping while not imposing undesired restrictions on subtypes. Carrying this to the extreme, for a closed collection of classes one may give specifications that have behavioral subtyping but impose quite arbitrary requirements on a method's implementation in different classes. (This phenomenon is related to our critical Lemma 8.13 on encoding dynamic dispatch using static dispatch.) This effectively results in reasoning by cases on possible dynamic types, the opposite of what behavioral subtyping and supertype abstraction are meant to provide.

Another direction for future work is to extend the treatment of behavioral subtyping to programming languages involving observations beyond simply initial and final states —most obviously, concurrent programs and trace properties. A good starting point may be the work of Alagic and Kouznetsova [2002].

## 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 the Dagstuhl conference center for hosting two other weeks of our work together. Thanks to participants of the 2008 PhD Fall School on Logics and Semantics of State, IT University of Copenhagen. Thanks to Barbara Liskov, Jeannette Wing, Matthew Parkinson, Shengchao Qin, Joseph Kiniry, Peter Müller, Andreas

Podelski, and Erik Poll for comments on earlier drafts. Thanks to the following for helpful discussions: Ian Stark, Matt Parkinson, and Bernhard Reus.

## A. APPENDIX

This appendix presents various proofs and ancillary definitions that were postponed from the main text.

The typing rules for expressions and commands are in Fig. 10 and Fig. 11. The rules for type cast and test are slightly simpler and more general than Java, allowing casts that always fail and tests that are always false.

### A.1. Typing rules, and semantics of expressions and commands not given in Sect. 4.5

$$\Gamma \vdash 0 : \mathsf{int} \qquad \Gamma \vdash \textbf{true} : \mathsf{bool} \qquad \frac{T \in \mathit{RefType}}{\Gamma \vdash \mathsf{null} : T} \qquad \Gamma \vdash x : \Gamma x$$

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

$$\frac{\Gamma \vdash x : U \qquad (f : T) \in \mathit{fields}\ V \qquad U \leq V}{\Gamma \vdash x.f : T} \qquad \Gamma \vdash \textbf{new}\ K() : K$$

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

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

Fig. 10. Typing rules for expressions. (For $(U)\ x$ and $x$ **is** $U$ one might prefer to add $U \leq T$, or the weaker condition that $U$ is a ref type, but there is no need.)

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

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

$$\frac{\Gamma \vdash x : \mathsf{bool} \qquad \Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash \textbf{if}\ x\ \textbf{then}\ C_1\ \textbf{else}\ C_2} \qquad \frac{[\Gamma,\ x : T] \vdash C}{\Gamma \vdash \textbf{var}\ x : T\ \textbf{in}\ C} \qquad \frac{\Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash C_1;\ C_2}$$

$$\frac{\Gamma \vdash x : T \qquad T \leq \mathsf{Thr}}{\Gamma \vdash \textbf{throw}\ x} \qquad \frac{\Gamma \vdash C_1 \qquad [\Gamma,\ x : T] \vdash C_2 \qquad T \leq \mathsf{Thr}}{\Gamma \vdash \textbf{try}\ C_1\ \textbf{catch}(x : T)\ C_2}$$

Fig. 11. Typing rules for commands.

The semantic clauses in Figures 12 and 13 use a few auxiliary definitions to follow.

The semantics is defined with respect to an arbitrary allocator. An ***allocator*** is a choice function for unused references, i.e., a function *fresh* that maps a pair $(r, h)$, with $h \in \mathit{Heap}(r)$, to a reference such that $\mathit{fresh}(r, h) \notin \mathit{dom}\, r$.[19]

If $s$ is a store then $s - \mathsf{exc}$ is the same store but with exc removed from its domain; similarly, $s - \overline{x}$ removes every $x$ in $\overline{x}$. We write $\sigma - x$ to abbreviate the state $\sigma$ but with $x$ removed from its store.

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

$$
\begin{array}{l}
[\![\Gamma \vdash x : T]\!](\eta)(r,h,s) \;=\; (r,h,[\mathsf{res} : s\,x, \mathsf{exc} : \mathit{null}]) \\
[\![\Gamma \vdash \mathbf{true} : \mathsf{bool}]\!](\eta)(r,h,s) \;=\; (r,h,[\mathsf{res} : \mathit{true}, \mathsf{exc} : \mathit{null}]) \\
[\![\Gamma \vdash 0 : \mathsf{int}]\!](\eta)(r,h,s) \;=\; (r,h,[\mathsf{res} : 0, \mathsf{exc} : \mathit{null}]) \\
[\![\Gamma \vdash \mathsf{null} : T]\!](\eta)(r,h,s) \;=\; (r,h,[\mathsf{res} : \mathit{null}, \mathsf{exc} : \mathit{null}]) \\
[\![\Gamma \vdash x = y : \mathsf{bool}]\!](\eta)(r,h,s) \;= \\
\quad \mathbf{let}\ v = (\mathbf{if}\ s\,x = s\,y\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ \mathit{false})\ \mathbf{in}\ (r,h,[\mathsf{res} : v, \mathsf{exc} : \mathit{null}]) \\
[\![\Gamma \vdash \mathbf{new}\ K() : K]\!](\eta)(r,h,s) \;= \\
\quad \mathbf{let}\ o = \mathit{fresh}(r,h)\ \mathbf{in\ let}\ r_0 = [r\,,\ o : K]\ \mathbf{in\ let}\ h_0 = [h\,,\ o : \mathit{defaultObrecord}\ K]\ \mathbf{in} \\
\quad (r_0, h_0, [\mathsf{res} : o, \mathsf{exc} : \mathit{null}]) \\
[\![\Gamma \vdash x.f : T]\!](\eta)(r,h,s) \;= \\
\quad \mathbf{if}\ s\,x \neq \mathit{null}\ \mathbf{then}\ (r,h,[\mathsf{res} : h(s\,x)\,f, \mathsf{exc} : \mathit{null}])\ \mathbf{else}\ \mathit{except}(r,h,T,\mathsf{NullDeref}) \\
[\![\Gamma \vdash (U)\ x : U]\!](\eta)(r,h,s) \;= \\
\quad \mathbf{if}\ s\,x = \mathit{null} \vee r(s\,x) \leq U\ \mathbf{then}\ (r,h,[\mathsf{res} : s\,x, \mathsf{exc} : \mathit{null}])\ \mathbf{else}\ \mathit{except}(r,h,U,\mathsf{ClassCast}) \\
[\![\Gamma \vdash x\ \mathbf{is}\ U : \mathsf{bool}]\!](\eta)(r,h,s) \;= \\
\quad \mathbf{let}\ v = (\mathbf{if}\ s\,x \neq \mathit{null} \wedge \rho(s\,x) \leq U\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ \mathit{false})\ \mathbf{in}\ (r,h,[\mathsf{res} : v, \mathsf{exc} : \mathit{null}])
\end{array}
$$

Fig. 12. Semantics of expressions other than **let** and method call, for which see equations (10), (12) and (13). Read $[\![-]\!]$ as either $\mathcal{D}[\![-]\!]$ or $\mathcal{S}[\![-]\!]$ throughout, and $\eta$ is either normal or extended method environment accordingly.

$$
\begin{array}{l}
[\![\Gamma \vdash x.f := y]\!](\eta)(r,h,s) \;= \\
\quad \mathbf{if}\ s\,x \neq \mathit{null}\ \mathbf{then}\ (r,[h \mid s\,x.f : s\,y],s)\ \mathbf{else}\ \mathit{except}(r,h,s,\mathsf{NullDeref}) \\
[\![\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!](\eta)(r,h,s) \;= \\
\quad \mathbf{if}\ s\,x = \mathit{true}\ \mathbf{then}\ [\![\Gamma \vdash C_1]\!](\eta)(r,h,s)\ \mathbf{else}\ [\![\Gamma \vdash C_2]\!](\eta)(r,h,s) \\
[\![\Gamma \vdash \mathbf{var}\ x : T\ \mathbf{in}\ C]\!](\eta)(r,h,s) \;= \\
\quad \mathbf{lets}\ (r_1,h_1,s_1) = [\![\Gamma, x : T \vdash C]\!](\eta)(r,h,[s\,,\ x : \mathit{default}\ T])\ \mathbf{in}\ (r_1,h_1,s_1 - x) \\
[\![\Gamma \vdash C_1;\ C_2]\!](\eta)(r,h,s) \;= \\
\quad \mathbf{lets}\ (r_1,h_1,s_1) = [\![\Gamma \vdash C_1]\!](\eta)(r,h,s)\ \mathbf{in} \\
\quad \mathbf{if}\ s_1\,\mathsf{exc} = \mathit{null}\ \mathbf{then}\ [\![\Gamma \vdash C_2]\!](\eta)(r_1,h_1,s_1 - \mathsf{exc})\ \mathbf{else}\ (r_1,h_1,s_1) \\
[\![\Gamma \vdash \mathbf{throw}\ x]\!](\eta)(r,h,s) \;= \\
\quad \mathbf{if}\ s\,x \neq \mathit{null}\ \mathbf{then}\ (r,h,[s\,,\ \mathsf{exc} : s\,x])\ \mathbf{else}\ \mathit{except}(r,h,s,\mathsf{NullDeref}) \\
[\![\Gamma \vdash \mathbf{try}\ C_1\ \mathbf{catch}(x : T)\ C_2]\!](\eta)(r,h,s) \;= \\
\quad \mathbf{lets}\ (r_1,h_1,s_1) = [\![\Gamma \vdash C_1]\!](\eta)(r,h,s)\ \mathbf{in} \\
\quad \mathbf{if}\ s_1\,\mathsf{exc} = \mathit{null} \vee r(s_1\,\mathsf{exc}) \not\leq T\ \mathbf{then}\ (r_1,h_1,s_1) \\
\quad \mathbf{else\ let}\ s_3 = [s_1 \mid x : s_1\,\mathsf{res}] - \mathsf{exc}\ \mathbf{in} \\
\qquad \mathbf{lets}\ (r_2,h_2,s_2) = [\![\Gamma, x : T \vdash C_2]\!](\eta)(r_1,h_1,s_3)\ \mathbf{in}\ (r_2,h_2,s_2 - x)
\end{array}
$$

Fig. 13. Semantics of commands except for assignment, for which see equation (11). Read $[\![-]\!]$ as either $\mathcal{D}[\![-]\!]$ or $\mathcal{S}[\![-]\!]$ throughout, and $\eta$ is either normal or extended method environment accordingly.

We write $\mathit{default}\,T$ for the default value for type $T$. We arbitrarily choose $\mathit{default}\,\mathsf{bool}$ to be *false* and $\mathit{default}\,\mathsf{int}$ to be 0. We need to define $\mathit{default}\,T = \mathit{null}$ if $T$ is a ref type, so that it is an element of $\mathit{Val}(T,r)$ for all $r$. Also, $\mathit{defaultObrecord}\,K$ is defined as the element of $\mathit{Obrecord}(K,r)$ that gives default values to all fields.

We use a helping function to create exceptional result states. Given a ref context $r$, heap $h \in \mathit{Heap}(r)$, classname $K \leq \mathsf{Thr}$, and any type $T$ we define $\mathit{except}(r,h,T,K)$ to be an element of $\mathit{State}([\mathsf{res} : T, \mathsf{exc} : \mathsf{Thr}])$ as follows:

$$
\begin{array}{rl}
\mathit{except}(r,h,T,K) \;=\; & \mathbf{let}\ o = \mathit{fresh}(r,h)\ \mathbf{in} \\
& \mathbf{let}\ r_0 = [r\,,\ o : K]\ \mathbf{in} \\
& \mathbf{let}\ h_0 = [h\,,\ o : \mathit{defaultObrecord}\ K]\ \mathbf{in}\ (r_0,h_0,[\mathsf{res} : \mathit{default}\ T, \mathsf{exc} : o])
\end{array}
$$

This is similar to the semantics of **new** $K()$, but the new object is assigned to exc rather than to res. The following similar function is used in the semantics of commands. Given $(r,h,s)$ in $\mathit{State}(\Gamma)$ and

classname $K \leq \mathsf{Thr}$ we define $except(r,h,s,K)$ to be an element of $State([\Gamma\,,\ \mathsf{exc}:\mathsf{Thr}])$ as follows:

$$\begin{array}{rl} except(r,h,s,K) \ = & \textbf{let } o = fresh(r,h) \textbf{ in} \\ & \textbf{let } r_0 = [r\,,\ o:K] \textbf{ in} \\ & \textbf{let } h_0 = [h\,,\ o:defaultObrecord\ K] \textbf{ in } (r_0, h_0, [s\,,\ \mathsf{exc}:o]) \end{array}$$

### A.2. Semantics of class table

Method environments are ordered by $\eta \leq \eta'$ iff $\eta(K,m) \leq \eta'(K,m)$ for all $K, m$. This second $\leq$ refers to the usual ordering on state transformers: that is, for $\varphi$ and $\psi$ in $STrans(\Gamma,\Gamma')$, define $\varphi \leq \psi$ iff for all $\sigma$ in $State(\Gamma)$ we have either $\varphi\,\sigma = \psi\,\sigma$ or $\varphi\,\sigma = \bot$.

The first step in defining the semantics of the class table is to define the semantics of the declaration of a method $m$ in some class $K$, written $\mathcal{D}[\![K;\ \textbf{meth}\ m(\overline{x:T}):T\ \{\ C\ \}]\!]$, as a function from method environments to $SemMeth(K,m)$. That is, the declaration denotes an element of

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

Let $\Gamma = [\mathsf{self}:K, \mathsf{res}:T, \overline{x:T}]$ so that $\Gamma \vdash C$ due to condition (W5) in the definition of a well-formed class table. For each method environment $\eta$ and for each state $(r,h,s)$ in $State([\mathsf{self}:K, \overline{x:T}])$, define

$$\begin{array}{l} \mathcal{D}[\![K;\ \textbf{meth}\ m(\overline{x:T}):T\ \{\ C\ \}]\!](\eta)(r,h,s) \ = \\ \quad \textbf{let } s_0 = [s\,,\ \mathsf{res}:default\ T] \textbf{ in} \\ \quad \textbf{lets } (r_1,h_1,s_1) = \mathcal{D}[\![\Gamma \vdash C]\!](\eta)(r,h,s_0) \textbf{ in } (r_1,h_1,s_1 - (\mathsf{self},\overline{x})). \end{array}$$

The next step is to define a chain $\eta \in \mathbb{N} \to MethEnv$ of method environments as follows.

$$\eta_0(K,m) = \lambda\sigma \cdot \bot, \quad \text{for any } m \text{ declared or inherited in } K.$$

$$\begin{array}{ll} \eta_{j+1}(K,m) = \mathcal{D}[\![K;\ mdec]\!](\eta_j), & \text{if } m \text{ is declared as } mdec \text{ in } K. \\ \eta_{j+1}(K,m) = restr(\eta_{j+1}(L,m),\ K), & \text{if } m \text{ is inherited in } K \text{ from } L. \end{array}$$

Here *restr* restricts the function $\eta_{j+1}(L,m)$, which is defined on states with $\mathsf{self}:L$, to states with $\mathsf{self}:K$. This works because $K \leq L$ implies that for all $r \in RefCtx$, $Val(K,r) \subseteq Val(L,r)$ which in turn induces an inclusion $State([\mathsf{self}:K, \overline{x:T}]) \subseteq State([\mathsf{self}:L, \overline{x:T}])$ on states.

We make such conversions explicit throughout the paper, since behavioral subtyping is all about refinement or satisfaction relations between programs and specifications at different types.

The everywhere-$\bot$ function is the least element in the set of state transformers of a given type, and this induces the least method environment, $\eta_0$. For any $\Gamma \vdash C$, the semantics $\mathcal{D}[\![\Gamma \vdash C]\!]$ 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 $i \leq j \Rightarrow \eta_i \leq \eta_j$, that is, the approximation chain is ascending.

The third and last step is to define the semantics, $\widehat{\eta}$, of the class table, also written $\mathcal{D}[\![CT]\!]$. It is defined to be the least upper bound of the approximation chain. Characterization of the least upper bound is straightforward. Machine-checked proofs of the continuity properties, as well as type soundness etc., appear in Leavens et al. [2006], building on Naumann [2005].

### A.3. Proof of Proposition 5.18 in Sect. 5.4

First we rewrite item (a) as follows:

$$\begin{array}{ll} & (J, pre', post') \sqsupseteq (I, pre, post) \\ \iff & \quad \text{definition of } \sqsupseteq, \text{ omit range } \varphi \in STrans(\Gamma,\Gamma') \\ & \forall \varphi \cdot \varphi \models (J, pre', post') \Rightarrow \varphi \models (I, pre, post) \\ \iff & \quad \text{definition of } \models \text{ for general specs, omit ranges } i \in I, j \in J \\ & \forall \varphi \cdot (\forall j \cdot \varphi \models (pre'_j, post'_j)) \Rightarrow (\forall i \cdot \varphi \models (pre_i, post_i)) \\ \iff & \quad \text{definition of } \models \text{ for simple specs} \\ & \forall \varphi \cdot (\forall j, \tau \cdot \tau \in pre'_j \Rightarrow \varphi(\tau) \in post'_j) \Rightarrow (\forall i, \sigma \cdot \sigma \in pre_i \Rightarrow \varphi(\sigma) \in post_i) \\ \iff & \quad \text{predicate calculus} \\ & \forall \varphi, i, \sigma \cdot \sigma \in pre_i \Rightarrow ((\forall j, \tau \cdot \tau \in pre'_j \Rightarrow \varphi(\tau) \in post'_j) \Rightarrow \varphi(\sigma) \in post_i) \\ \iff & \quad \text{predicate calculus} \\ & \forall i, \sigma \cdot \sigma \in pre_i \Rightarrow (\forall \varphi \cdot (\forall j, \tau \cdot \tau \in pre'_j \Rightarrow \varphi(\tau) \in post'_j) \Rightarrow \varphi(\sigma) \in post_i) \end{array}$$

Recall that item (b) is

(b) $\forall i \in I, \sigma \in \mathit{State}(\Gamma) \cdot \sigma \in \mathit{pre}_i$
$\Rightarrow (\exists j \in J \cdot \sigma \in \mathit{pre}'_j) \wedge (\forall \tau \in \mathit{State}(\Gamma') \cdot (\forall j \in J \cdot \sigma \in \mathit{pre}'_j \Rightarrow \tau \in \mathit{post}'_j) \Rightarrow \tau \in \mathit{post}_i)$

In accord with the rewritten (a), items (a) and (b) are equivalent if the following are equivalent for all $i \in I$ and all $\sigma \in \mathit{pre}_i$:

(a') $\forall \varphi \cdot (\forall j, \tau \cdot \tau \in \mathit{pre}'_j \Rightarrow \varphi(\tau) \in \mathit{post}'_j) \Rightarrow \varphi(\sigma) \in \mathit{post}_i$
(b') $(\exists j \cdot \sigma \in \mathit{pre}'_j) \wedge (\forall \tau \cdot (\forall j \cdot \sigma \in \mathit{pre}'_j \Rightarrow \tau \in \mathit{post}'_j) \Rightarrow \tau \in \mathit{post}_i x)$

For arbitrary $i$ and $\sigma$ with $\sigma \in \mathit{pre}_i$ we argue by mutual implication.

To show (a') follows from (b'), consider any $\varphi$. Assume the antecedent $\forall j, \tau \cdot \tau \in \mathit{pre}'_j \Rightarrow \varphi(\tau) \in \mathit{post}'_j$ in (a'), to show $\varphi(\sigma) \in \mathit{post}_i$. Instantiate $\tau := \sigma$ to get $\forall j \cdot \sigma \in \mathit{pre}'_j \Rightarrow \varphi(\sigma) \in \mathit{post}'_j$. By the first conjunct in (b') there is some $j$ with $\sigma \in \mathit{pre}'_j$ and so $\varphi(\sigma) \in \mathit{post}'_j$, whence $\varphi(\sigma) \neq \bot$. Because $\varphi(\sigma)$ is a state, we can instantiate the second conjunct in (b') by $\tau := \varphi(\sigma)$ which yields $(\forall j \cdot \sigma \in \mathit{pre}'_j \Rightarrow \varphi(\sigma) \in \mathit{post}'_j) \Rightarrow \varphi(\sigma) \in \mathit{post}_i$ and thus $\varphi(\sigma) \in \mathit{post}_i$.

To show that (b') follows from (a'), we need that $(J, \mathit{pre}', \mathit{post}')$ is satisfiable. Choose some state transformer $\psi$ that satisfies $(J, \mathit{pre}', \mathit{post}')$ and yields $\bot$ on any initial state where that is allowed. That is, for any state $\rho$,

$\psi(\rho) = \bot$ if $\neg(\exists j \cdot \rho \in \mathit{pre}'_j)$, and otherwise:
$\psi(\rho) = \tau$ where $\tau$ is chosen to be any state such that $\forall j \cdot \rho \in \mathit{pre}'_j \Rightarrow \tau \in \mathit{post}'_j$.

There must be some such $\tau$ in the second case, by Lemma 5.16. By construction we have

$$\forall j, \rho \cdot \rho \in \mathit{pre}'_j \Rightarrow \psi(\rho) \in \mathit{post}'_j \tag{51}$$

We prove the first conjunct of (b') by contradiction. Suppose to the contrary that $\forall j \cdot \sigma \notin \mathit{pre}'_j$. Instantiate (a') with $\varphi := \psi$ and use (51) which yields $\psi(\sigma) \in \mathit{post}_i$ —but this contradicts $\psi(\sigma) = \bot$ from the definition of $\psi$, because predicates do not contain $\bot$.

Now we show the second conjunct in (b'). Consider any $\tau$ and assume the antecedent $(\forall j \cdot \sigma \in \mathit{pre}'_j \Rightarrow \tau \in \mathit{post}'_j)$, to prove $\tau \in \mathit{post}_i$. Let $\psi$ be defined as above; the assumption lets us make the specific choice $\psi(\sigma) = \tau$. Taking $\varphi := \psi$ in (a') we get $\psi(\sigma) \in \mathit{post}_i$, thus $\tau \in \mathit{post}_i$.

### A.4. Refactoring the denotational semantics using an algebra state transformers.

For each expression and command form, we reformulate the semantic definition in Sect. 4.5 using some primitive state transformers together with three operations on state transformers —sequence, alternatives, and a form of pairing. The three operations on state transformers are as follows:

*Kleisli composition.* Given $\varphi_0$ of type $\Gamma_0 \rightsquigarrow \Gamma_1$ and $\varphi_1$ of type $\Gamma_1 \rightsquigarrow \Gamma_2$, define $\varphi_0; \varphi_1$ of type $\Gamma_0 \rightsquigarrow \Gamma_2$ by

$$(\varphi_0; \varphi_1)\,\sigma \;=\; (\textbf{if } \varphi_0\,\sigma = \bot \textbf{ then } \bot \textbf{ else } \varphi_1(\varphi_0\,\sigma)).$$

*Alternatives.* For $\varphi$ and $\psi$ of type $\Gamma \rightsquigarrow \Gamma'$ and $P \subseteq \mathit{State}(\Gamma)$, define IF $P$ THEN $\varphi$ ELSE $\psi$ of type $\Gamma \rightsquigarrow \Gamma'$ by

$$(\text{IF } P \text{ THEN } \varphi \text{ ELSE } \psi)\,\sigma \;=\; \textbf{if } \sigma \in P \textbf{ then } \varphi\,\sigma \textbf{ else } \psi\,\sigma.$$

*Store pairing.* For $\Gamma$ and $\Gamma'$ with disjoint domains and $\varphi$ of type $\Gamma \rightsquigarrow \Gamma'$, define $\langle \varphi\ ,\ id \rangle$ of type $\Gamma \rightsquigarrow [\Gamma', \Gamma]$ by

$$\langle \varphi\ ,\ id \rangle(r, h, s) \quad = \quad \textbf{if } \varphi(r, h, s) = \bot \textbf{ then } \bot \textbf{ else } \varphi(r, h, s){+}s,$$

where we write $+s$ to add store $s$ to the state $\varphi(r, h, s)$. That is, for any $(t, k, s') \in \mathit{State}(\Gamma')$ and $s \in \mathit{Store}(\Gamma)$ with $\Gamma$ disjoint from $\Gamma'$ we define

$$(t, k, s'){+}s \quad = \quad (t, k, [s', s]), \tag{52}$$

where the state $(t, k, [s', s])$ is in $\mathit{State}([\Gamma', \Gamma])$.

Note that Kleisli composition takes divergence into account but does nothing special with the exc variable.[20]

[20]Store pairing seems ad hoc. A proper "algebra" of state transformers might include products in a general form, with pairing $\Gamma \rightsquigarrow \Gamma' \times \Gamma''$ etc. But that would require us to introduce specifications of type $\Gamma \rightsquigarrow \Gamma' \times \Gamma''$.

Refactoring the semantics of conditionals is straightforward:

$$[\![\Gamma \vdash \textbf{if } x \textbf{ then } C_1 \textbf{ else } C_2]\!](\eta) \;=\; \textsc{if } \mathit{true}(x) \textsc{ then } [\![\Gamma \vdash C_1]\!](\eta) \textsc{ else } [\![\Gamma \vdash C_2]\!](\eta),$$

where $\mathit{true}(x)$ is the set of $\Gamma$-states where $x$ is true.

The semantics of sequential composition refactors as follows:

$$[\![\Gamma \vdash C_1;\; C_2]\!](\eta) = [\![\Gamma \vdash C_1]\!](\eta); \textsc{if } \mathit{null}(\mathsf{exc}) \textsc{ then } \mathit{dropExc}; [\![\Gamma \vdash C_2]\!](\eta) \textsc{ else } \mathit{ident},$$

where

— $\mathit{null}(\mathsf{exc})$ is the set of $[\Gamma\,,\ \mathsf{exc} : \mathsf{Thr}]$-states in which $\mathsf{exc}$ is null
— $\mathit{dropExc}$ drops $\mathsf{exc}$ from the state space (it can be written $\lambda\sigma \cdot \sigma - \mathsf{exc}$),
— $\mathit{ident}$ is the identity state transformer (here used to retain the exception from $C_1$).

To see the need for store pairing, consider the semantics of assignment, Eq. (11). Using $[\![-]\!]$ for either $\mathcal{S}[\![-]\!]$ or $\mathcal{D}[\![-]\!]$, we can write $[\![\Gamma \vdash x := E]\!](\eta)$ in the following way, using store pairing and sequence:

$$[\![\Gamma \vdash x := E]\!](\eta) = \langle([\![\Gamma \vdash E : T]\!](\eta)\,;\ \mathit{rename})\,,\ \mathit{id}\rangle; \mathit{assg}, \tag{53}$$

where

— $\mathit{rename} : [\mathsf{res} : T, \mathsf{exc} : \mathsf{Thr}] \rightsquigarrow [\mathsf{res}' : T, \mathsf{exc} : \mathsf{Thr}]$ is the state transformer that just renames $\mathsf{res}$ to give a context[21] disjoint from $\Gamma$: $\mathit{rename}(r, h, s) = (r, h, [s \mid \mathsf{res}' : s\,\mathsf{res}] - \mathsf{res})$.
— $\mathit{assg} : [\Gamma, \mathsf{res}' : T, \mathsf{exc} : \mathsf{Thr}] \rightsquigarrow [\Gamma, \mathsf{exc} : \mathsf{Thr}]$ is the state transformer that updates $x$ with the value of $\mathsf{res}'$ if $\mathsf{exc}$ is null and in either case drops $\mathsf{res}'$ from the store: $\mathit{assg}(r, h, s) = (r, h, s')$ where $s'$ is $\textbf{if } \mathsf{exc} = \mathit{null} \textbf{ then } [s \mid x : s\,\mathsf{res}'] - \mathsf{res}' \textbf{ else } s - \mathsf{res}'$.

The proof of Eq. (53) is straightforward using the definitions.

The **var** construct refactors as

$$[\![\Gamma \vdash \textbf{var } x : T \textbf{ in } C]\!](\eta) = \mathit{extndef}; [\![\Gamma, x : T \vdash C]\!](\eta); \mathit{dropx}$$

where

— the primitive $\mathit{extndef} : \Gamma \rightsquigarrow [\Gamma\,,\ x : T]$ adds variable $x$ with default value
— $\mathit{dropx} : [\Gamma, x : T, \mathsf{exc} : \mathsf{Thr}] \rightsquigarrow [\Gamma\,,\ \mathsf{exc} : \mathsf{Thr}]$ discards $x$.

To refactor the semantics of **let** expressions, recall the semantics in Eq. (10), written here using $[\![-]\!]$ and $\eta$ to stand for either $\mathcal{D}[\![-]\!]$ and a normal method environment or $\mathcal{S}[\![-]\!]$ and an extended method environment:

$$\begin{array}{l}[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!](\eta)(r, h, s) \;= \\ \quad \textbf{lets } (r_0, h_0, s_0) = [\![\Gamma \vdash E : T]\!](\eta)(r, h, s) \textbf{ in} \\ \quad \textbf{if } s_0\,\mathsf{exc} \neq \mathit{null} \textbf{ then } (r_0, h_0, [\mathsf{res} : \mathit{default}\,U, \mathsf{exc} : s_0\,\mathsf{exc}]) \\ \quad \textbf{else let } s_1 = [s\,,\ x : s_0\,\mathsf{res}] \textbf{ in } [\![\Gamma, x : T \vdash E1 : U]\!](\eta)(r_0, h_0, s_1)\end{array}$$

It is straightforward to show that for either the static or dynamic semantics we have the refactoring

$$[\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1 : U]\!](\eta) = \langle([\![\Gamma \vdash E : T]\!](\eta)\,;\ \mathit{rename})\,,\ \mathit{id}\rangle; \mathit{doE1}(\eta) \tag{54}$$

for any $\eta$, where

— $\mathit{rename} : [\mathsf{res} : T, \mathsf{exc} : \mathsf{Thr}] \rightsquigarrow [\mathsf{res}' : T, \mathsf{exc} : \mathsf{Thr}]$ just renames $\mathsf{res}$ to $\mathsf{res}'$, the same as in (53) of Sect. A.4
— $\mathit{doE1}(\eta) : [\Gamma, \mathsf{res}' : T, \mathsf{exc} : \mathsf{Thr}] \rightsquigarrow [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thr}]$ is the state transformer

$$\textsc{if } \mathit{null}(\mathsf{exc}) \textsc{ then } \mathit{init}\,;\ [\![\Gamma, x : T \vdash E1 : U]\!](\eta) \textsc{ else } \mathit{propag}$$

— $\mathit{null}(\mathsf{exc})$ is the set of $[\Gamma, \mathsf{res}' : T, \mathsf{exc} : \mathsf{Thr}]$-states where $\mathsf{exc}$ is null
— $\mathit{init} : [\Gamma, \mathsf{res}' : T, \mathsf{exc} : \mathsf{Thr}] \rightsquigarrow [\Gamma, x : T]$ sets $x$ to the value of $\mathsf{res}'$ and discards $\mathsf{exc}$
— $\mathit{propag} : [\Gamma, \mathsf{res}' : T, \mathsf{exc} : \mathsf{Thr}] \rightsquigarrow [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thr}]$ is the (primitive) transformer that progopates the exception, i.e., it sets $\mathsf{res}$ to $\mathit{default}\,U$ and copies $\mathsf{exc}$ (the possible exception from $E$).

[21] We are only interested in $\Gamma$ that has $\mathsf{res}$ in its domain since it is present for any expression and command in a method body; cf. the typing rule for method body.

We omit the refactoring of $[\![\Gamma \vdash \textbf{try}\ C_1\ \textbf{catch}(x:T)\ C_2]\!]$. It is similar to the refactoring of sequential composition, together with **var**, using a primitive state transformer to rename variable exc to $x$ under the condition that exc is not null.

We did not need to refactor the semantics of method call; we directly derive its predicate transformer semantics in the proof of Lemma 7.1 in Sect. 7.2. For the sake of elegance, we refactor the semantics of method call $\Gamma \vdash x.m(\overline{y}) : U$, with $\Gamma\, x = T$ and $mtype(T, m) = \overline{z : U} \to U$ as in the typing rule for method call. We have

$$\mathcal{S}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\dot\eta) \;=\; \text{IF } null(x) \text{ THEN } excpt \text{ ELSE } args; \dot\eta(T, m) \tag{55}$$

$$\mathcal{D}[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\eta) \;=\; \text{IF } null(x) \text{ THEN } excpt \text{ ELSE } args; dispatch(T, m, \eta) \tag{56}$$

where

— $null(x)$ is the set of $\Gamma$-states in which $x$ is null
— $args : \Gamma \rightsquigarrow [\mathsf{self} : T, \overline{z : T}]$ is the semantics of the assignment "$\mathsf{self}, \overline{z} := x, \overline{y}$"
— $excpt$ is $\lambda(r, h, s) \cdot excpt(r, h, U, \mathsf{NullDeref})$, using $excpt$ from Sect. 4.5.
— $dispatch(T, m, \eta)$ is the state transformer of type $[\mathsf{self} : T, \overline{z : T}] \to [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thr}]$, defined by Eq. (14) in Sect. 4.

### A.5. Weakest preconditions for state transformer algebra

The following Lemma shows how $wp$ itself distributes over constructs of the algebra of state transformers. For sequence and alternatives the result is standard. For store pairing, the result is expressed using a kind of partial application for predicates, in terms of the variables in the store. It is an asymmetric operator notated $\ominus$ and defined as follows. For $\Gamma$ and $\Gamma'$ with disjoint domains, any predicate $Q$ on $[\Gamma', \Gamma]$, and $\Gamma$-store $s$, let $Q \ominus s$ be the $\Gamma'$-predicate defined by

$$\tau \;\in\; Q \ominus s \;\iff\; \tau + s \;\in\; Q.$$

using $+$ defined in Eq. (52). This lifts to an operation on predicate transformers: If $f$ models a program from $\Gamma$-states to $\Gamma'$-states, with the variables of $\Gamma'$ distinct from those of $\Gamma$, then $f \oplus id$ models the program that acts the same but also saves a copy of the initial store for $\Gamma$, yielding a state for $[\Gamma', \Gamma]$. For predicate transformer $f$ of type $\Gamma \rightsquigarrow \Gamma'$, define $f \oplus id$ of type $\Gamma \rightsquigarrow [\Gamma', \Gamma]$ by

$$(r, h, s) \in (f \oplus id)(Q) \iff (r, h, s) \in f(Q \ominus s).$$

LEMMA A.1 (WP DISTRIBUTION). For any state transformers $\varphi_0$ and $\varphi_1$ of suitable types, we have for any state $\sigma$ and predicate $Q$,

$$\sigma \in wp(\varphi_0; \varphi_1)(Q) \iff \sigma \in wp(\varphi_0)(wp(\varphi_1)(Q)).$$

For any state transformers $\varphi_0$ and $\varphi_1$ of suitable type, we have

$$\sigma \in wp(\text{IF } P \text{ THEN } \varphi \text{ ELSE } \psi)(Q) \iff \textbf{if } \sigma \in P \textbf{ then } \sigma \in wp(\varphi)(Q) \textbf{ else } \sigma \in wp(\psi)(Q),$$

or equivalently

$$wp(\text{IF } P \text{ THEN } \varphi \text{ ELSE } \psi)(Q) = (P \cap wp(\varphi)(Q)) \cup (\neg P \cap wp(\psi)(Q)), \tag{57}$$

where we write $\neg P$ for the set of states not in $P$.

For the case of store pairing we spell out the types. Let $\Gamma$ and $\Gamma'$ have disjoint domains and let $\varphi$ be of type $\Gamma \rightsquigarrow \Gamma'$. For any $\Gamma$-state $(r, h, s)$ and any predicate $Q$ on $[\Gamma', \Gamma]$, we have

$$(r, h, s) \in wp(\langle \varphi\, , id \rangle)(Q) \iff (r, h, s) \in wp(\varphi)(Q \ominus s) \tag{58}$$

or equivalently $wp(\langle \varphi\, , id \rangle) = wp(\varphi) \oplus id$.

PROOF. For sequence, observe for any state $\sigma$

$$\begin{array}{ll} & \sigma \in wp(\varphi_0; \varphi_1)(Q) \\ \iff & \quad \text{definitions of } wp \text{ and of ;} \\ & \varphi_0(\sigma) \neq \bot \wedge \varphi_1(\varphi_0(\sigma)) \in Q, \\ \iff & \quad \text{definition of } wp \text{ thrice} \\ & \sigma \in wp(\varphi_0)(true) \wedge \sigma \in wp(\varphi_0)(wp(\varphi_1)(Q)) \\ \iff & \quad wp(\varphi) \text{ is monotonic and } wp(\varphi_1)(Q) \text{ is a subset of "}true\text{"} \\ & \sigma \in wp(\varphi_0)(wp(\varphi_1)(Q)) \end{array}$$

$$
\begin{array}{l}
(r,h,s) \in \{\!|\Gamma \vdash x := E|\!\}(\theta)(Q) \iff (r,h,s) \in \{\!|\Gamma \vdash E : T|\!\}(\theta)(wp(rename)(wp(assg)(Q) \oplus s)) \\
(r,h,s) \in \{\!|\Gamma \vdash C_1;\ C_2|\!\}(\theta)(Q) \iff \\
\quad (r,h,s) \in \{\!|\Gamma \vdash C_1|\!\}(\theta)((null(\mathsf{exc}) \cap wp(dropExc)(\{\!|\Gamma \vdash C_2|\!\}(\theta)(Q))) \cup (\neg null(\mathsf{exc}) \cap Q)) \\
(r,h,s) \in \{\!|\Gamma \vdash \textbf{true} : \mathsf{bool}|\!\}(\theta))(Q) \iff (r,h,[\mathsf{res} : true, \mathsf{exc} : null]) \in Q \\
(r,h,s) \in \{\!|\Gamma \vdash \mathsf{null} : T|\!\}(\theta)(Q) \iff (r,h,[\mathsf{res} : null, \mathsf{exc} : null]) \in Q \\
(r,h,s) \in \{\!|\Gamma \vdash x = y : \mathsf{bool}|\!\}(\theta))(Q) \iff \\
\quad \textbf{let}\ v = (\textbf{if}\ s\,x = s\,y\ \textbf{then}\ true\ \textbf{else}\ false)\ \textbf{in}\ (r,h,[\mathsf{res} : v, \mathsf{exc} : null]) \in Q \\
(r,h,s) \in \{\!|\Gamma \vdash \textbf{new}\ K() : K|\!\}(\theta)(Q) \iff \\
\quad \textbf{let}\ o = fresh(r,h)\ \textbf{in let}\ r_0 = [r\ ,\ o : K]\ \textbf{in let}\ h_0 = [h\ ,\ o : defaultObrecord\ K]\ \textbf{in} \\
\quad (r_0,h_0,[\mathsf{res} : o, \mathsf{exc} : null]) \in Q \\
(r,h,s) \in \{\!|\Gamma \vdash x.f : T|\!\}(\theta)(Q) \iff \\
\quad \textbf{if}\ s\,x \neq null\ \textbf{then}\ (r,h,[\mathsf{res} : h(s\,x)\,f, \mathsf{exc} : null]) \in Q\ \textbf{else}\ except(r,h,T,\mathsf{NullDeref}) \in Q \\
(r,h,s) \in \{\!|\Gamma \vdash (U)\ x : U|\!\}(\theta)(Q) \iff \\
\quad \textbf{if}\ s\,x = null \vee r(s\,x) \leq U\ \textbf{then}\ (r,h,[\mathsf{res} : s\,x, \mathsf{exc} : null]) \in Q\ \textbf{else}\ except(r,h,U,\mathsf{ClassCast}) \in Q \\
(r,h,s) \in \{\!|\Gamma \vdash x\ \textbf{is}\ U : \mathsf{bool}|\!\}(wp(\eta))(Q) \iff \\
\quad \textbf{let}\ v = (\textbf{if}\ s\,x \neq null \wedge \rho(s\,x) \leq U\ \textbf{then}\ true\ \textbf{else}\ false)\ \textbf{in}\ (r,h,[\mathsf{res} : v, \mathsf{exc} : null]) \in Q \\
(r,h,s) \in \{\!|\Gamma \vdash \textbf{let}\ x\ \textbf{be}\ E\ \textbf{in}\ E1 : U|\!\}(\theta)(Q) \iff \\
\quad (r,h,s) \in \{\!|\Gamma \vdash E : T|\!\}(\theta))(wp(rename)(ptDoE1(Q) \oplus s)) \qquad \text{where } ptDoE1 \text{ is from (59).} \\
(r,h,s) \in \{\!|\Gamma \vdash \textbf{throw}\ \ x|\!\}(\theta)(Q) \iff \\
\quad \textbf{if}\ s\,x \neq null\ \textbf{then}\ (r,h,[s\ ,\ \mathsf{exc} : s\,x]) \in Q\ \textbf{else}\ except(r,h,s,\mathsf{NullDeref}) \in Q \\
(r,h,s) \in \{\!|\Gamma \vdash x.f := y|\!\}(\theta)(Q) \iff \\
\quad \textbf{if}\ s\,x \neq null\ \textbf{then}\ (r,[h \mid s\,x.f : s\,y],s) \in Q\ \textbf{else}\ except(r,h,s,\mathsf{NullDeref}) \in Q \\
(r,h,s) \in \{\!|\Gamma \vdash \textbf{var}\ x : T\ \textbf{in}\ C|\!\}(\theta)(Q) \iff \\
\quad (r,h,s) \in wp(extndef)(\{\!|\Gamma, x : T \vdash C|\!\}(\theta)(wp(dropx)(Q)))
\end{array}
$$

Fig. 14. Predicate transformer semantics, cf. Fig. 6. See the text for abbreviations like *ptRename* and *ptDoE*1.

The case for alternatives is even simpler and is left to the reader. For store pairing, observe for any state $(r,h,s)$

$$
\begin{array}{ll}
 & (r,h,s) \in wp(\langle \varphi\ ,\ id \rangle)(Q) \\
\iff & \quad \text{definitions} \\
 & \varphi(r,h,s) \neq \bot \wedge (\varphi(r,h,s) + s) \in Q \\
\iff & \quad \text{definitions} \\
 & (r,h,s) \in wp(\varphi)(true) \wedge \varphi(r,h,s) \in (Q \ominus s) \\
\iff & \quad \text{definition of } wp \\
 & (r,h,s) \in wp(\varphi)(true) \wedge (r,h,s) \in wp(\varphi)(Q \ominus s) \\
\iff & \quad wp(\varphi) \text{ monotonic and } Q \ominus s \text{ is a subset of ``}true\text{''} \\
 & (r,h,s) \in wp(\varphi)(Q \ominus s)
\end{array}
$$

□

### A.6. Predicate transformer semantics derived for Lemma 7.1

Fig. 14 gives the clauses of predicate transformer semantics omitted from Fig. 6. This subsection derives the definitions, completing the proof of Lemma 7.1. Omitted from the figure and proof is the case of try/catch, because we omitted that from the refactorings in Sect. A.4; it is essentially a combination of sequence and **var**.

We begin with assignment. For brevity we omit the context $\Gamma$.

$$\begin{array}{ll} & (r,h,s) \in wp([\![x := E]\!](\eta))(Q) \\ \iff & \quad \text{refactoring Eq. (53) in Sect. A.4} \\ & (r,h,s) \in wp(\langle([\![E : T]\!](\eta); rename)\ ,\ id\rangle; assg)(Q) \\ \iff & \quad \text{Lemma A.1 for sequence} \\ & (r,h,s) \in wp(\langle([\![E : T]\!](\eta); rename)\ ,\ id\rangle)(wp(assg)(Q)) \\ \iff & \quad \text{Lemma A.1 for store pairing} \\ & (r,h,s) \in wp(([\![E : T]\!](\eta); rename))(wp(assg)(Q) \oplus s) \\ \iff & \quad \text{Lemma A.1 for sequence} \\ & (r,h,s) \in wp([\![E : T]\!](\eta))(wp(rename)(wp(assg)(Q) \oplus s)) \\ \iff & \quad \text{induction hypothesis for } E \\ & (r,h,s) \in \{\![E : T]\!\}(wp(\eta))(wp(rename)(wp(assg)(Q) \oplus s)) \\ \iff & \quad \text{definition of } \{\![x := E]\!\} \text{ (see Fig. 6)} \\ & (r,h,s) \in \{\![x := E]\!\}(wp(\eta))(Q) \end{array}$$

Since *assg* and *rename* are fixed state transformers, the penultimate formula can serve as the definition. In this calculation and in the ones to follow, the last step tells us the general definition for arbitrary predicate transformer environment $\theta$ (as it is given in Fig. 14).

For sequence, we derive an appropriate predicate transformer semantics as follows, where we write $\neg null(\mathsf{exc})$ for the complement of the set of states $null(\mathsf{exc})$.

$$\begin{array}{ll} & wp([\![C_1;\ C_2]\!](\eta))(Q) \\ = & \quad \text{refactoring of sequence} \\ & wp([\![C_1]\!](\eta); \textsc{if}\ null(\mathsf{exc})\ \textsc{then}\ dropExc; [\![C_2]\!](\eta)\ \textsc{else}\ ident)(Q) \\ = & \quad wp \text{ distribute sequence, Lemma A.1} \\ & wp([\![C_1]\!](\eta))(wp(\textsc{if}\ null(\mathsf{exc})\ \textsc{then}\ dropExc; [\![C_2]\!](\eta)\ \textsc{else}\ ident)(Q)) \\ = & \quad wp \text{ distribute alternative Eq. (57)} \\ & wp([\![C_1]\!](\eta))(null(\mathsf{exc}) \cap wp(dropExc; [\![C_2]\!](\eta))(Q)) \cup (\neg null(\mathsf{exc}) \cap wp(ident)(Q)) \\ = & \quad wp \text{ distribute sequence, Lemma A.1, } wp(ident) \\ & wp([\![C_1]\!](\eta))((null(\mathsf{exc}) \cap wp(dropExc)(wp([\![C_2]\!](\eta))(Q))) \cup (\neg null(\mathsf{exc}) \cap Q)) \\ = & \quad \text{induction} \\ & \{\![C_1]\!\}(wp(\eta)))((null(\mathsf{exc}) \cap wp(dropExc)(\{\![C_2]\!\}(wp(\eta))(Q))) \cup (\neg null(\mathsf{exc}) \cap Q)) \\ = & \quad \text{definition of } \{\![C_1;\ C_2]\!\} \\ & \{\![C_1;\ C_2]\!\}(wp(\eta))(Q) \end{array}$$

The cases for **let** expressions, and all the remaining command forms, are similar to the preceding cases. Before embarking on those derivations, we mention how one could go astray in defining the semantics of constructs that involve sequence.

*Remark* A.2. The semantics for sequence is derived in Sect. 7.2, in a way that avoids an interesting misstep. To see what could go awry, consider the following calculation.

$$\begin{array}{ll} & \sigma \in wp([\![C_1;\ C_2]\!](\eta))(Q) \\ \iff & \quad \text{refactoring of sequence} \\ & \sigma \in wp([\![C_1]\!](\eta); \textsc{if}\ null(\mathsf{exc})\ \textsc{then}\ dropExc; [\![C_2]\!](\eta)\ \textsc{else}\ ident)(Q) \\ \iff & \quad wp \text{ distribute sequence, Lemma A.1} \\ & \sigma \in wp([\![C_1]\!](\eta))(wp(\textsc{if}\ null(\mathsf{exc})\ \textsc{then}\ dropExc; [\![C_2]\!](\eta)\ \textsc{else}\ ident)(Q)) \end{array}$$

If after the last step above we used the definition of *wp* (26), we could show that the last formula above was true just when

$$[\![C_1]\!](\eta)(\sigma) \in wp(\textsc{if}\ null(\mathsf{exc})\ \textsc{then}\ dropExc; [\![C_2]\!](\eta)\ \textsc{else}\ ident)(Q)$$

While this is valid, it is a misstep, because it leads to a flawed definition. To see this, consider that from this last formula, one would proceed via distribution of *wp* over sequence and alternatives, and induction, to the following putative definition:

$$\begin{array}{lll} \sigma \in \{\![C_1;\ C_2]\!\}(\theta)(Q) & \iff ? & \textbf{if } \sigma \in \{\![C_1]\!\}(\theta)(null(\mathsf{exc})) \\ & & \textbf{then } \sigma \in \{\![C_1]\!\}(\theta)(wp(dropExc)(\{\![C_2]\!\}(\theta)(Q))) \\ & & \textbf{else } \sigma \in \{\![C_1]\!\}(\theta)(Q) \end{array}$$

Although it validates Lemma 7.1, the misstep is dubious because it exploits determinacy of the state transformer semantics for $C_1$, whereas $\{\!|C_1|\!\}(\theta)$ may well inherit nondeterminacy from meanings in $\theta$. We need nondeterminacy in method environments for the main results where $\theta$ gets instantiated by $\{\!|ST|\!\}$. Informally, the putative definition can be read as follows: if $C_1$ ensures the absence of exceptions, the $C_2$ must establish $Q$; otherwise $C_1$ must establish $Q$. But in the case that $C_1$ nondeterministically allows the possibility of exception or not, this is not an accurate semantics. In particular, it would falsify the critical Theorem 8.10, which relates the dynamic to the static semantics. □

For **true** we have

$$\begin{array}{ll} & (r,h,s) \in wp([\![\Gamma \vdash \mathbf{true} : \mathsf{bool}]\!](\eta))(Q) \\ \iff & \quad \text{definition of } wp \\ & [\![\Gamma \vdash \mathbf{true} : \mathsf{bool}]\!](\eta)(r,h,s) \in Q \\ \iff & \quad \text{definition of } [\![\Gamma \vdash \mathbf{true} : \mathsf{bool}]\!] \\ & (r,h,[\mathsf{res} : \mathit{true}, \mathsf{exc} : \mathit{null}]) \in Q \\ \iff & \quad \text{definition of } \{\!|[\Gamma \vdash \mathbf{true} : \mathsf{bool}]|\!\} \\ & (r,h,s) \in \{\!|[\Gamma \vdash \mathbf{true} : \mathsf{bool}]|\!\}(wp(\eta))(Q) \end{array}$$

For $\Gamma \vdash x = y : \mathsf{bool}$ we have

$$\begin{array}{ll} & (r,h,s) \in wp([\![\Gamma \vdash x = y : \mathsf{bool}]\!](\eta))(Q) \\ \iff & \quad \text{definition of } wp \\ & [\![\Gamma \vdash x = y : \mathsf{bool}]\!](\eta)(r,h,s) \in Q \\ \iff & \quad \text{definition of } [\![-]\!] \\ & (\mathbf{let}\ v = (\mathbf{if}\ s\,x = s\,y\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ \mathit{false})\ \mathbf{in}\ (r,h,[\mathsf{res} : v, \mathsf{exc} : \mathit{null}])) \in Q \\ \iff & \quad \text{definition of } \{\!|[x = y]|\!\} \\ & (r,h,s) \in \{\!|[\Gamma \vdash x = y : \mathsf{bool}]|\!\}(wp(\eta)))(Q) \end{array}$$

Simplifications come to mind, but for our purposes we may as well read the definition directly off this proof. The subsequent calculations are similar so we omit hints.

For **new** $K()$ we have

$$\begin{array}{ll} & (r,h,s) \in wp([\![\Gamma \vdash \mathbf{new}\ K() : K]\!](\eta))(Q) \\ \iff & [\![\Gamma \vdash \mathbf{new}\ K() : K]\!](\eta)(r,h,s) \in Q \\ \iff & (\mathbf{let}\ o = \mathit{fresh}(r,h)\ \mathbf{in}\ \mathbf{let}\ r_0 = [r\,,\ o : K]\ \mathbf{in} \\ & \quad \mathbf{let}\ h_0 = [h\,,\ o : \mathit{defaultObrecord}\ K]\ \mathbf{in}\ (r_0, h_0, [\mathsf{res} : o, \mathsf{exc} : \mathit{null}])) \quad \in Q \\ \iff & (r,h,s) \in \{\!|[\Gamma \vdash \mathbf{new}\ K() : K]|\!\}(wp(\eta))(Q) \end{array}$$

For $\Gamma \vdash x.f : T$ we have

$$\begin{array}{ll} & (r,h,s) \in wp([\![\Gamma \vdash x.f : T]\!](\eta))(Q) \\ \iff & [\![\Gamma \vdash x.f : T]\!](\eta)(r,h,s) \in Q \\ \iff & (\mathbf{if}\ s\,x \neq \mathit{null}\ \mathbf{then}\ (r,h,[\mathsf{res} : h(s\,x)\,f, \mathsf{exc} : \mathit{null}])\ \mathbf{else}\ \mathit{except}(r,h,T,\mathsf{NullDeref})) \in Q \\ \iff & (r,h,s) \in \{\!|[\Gamma \vdash x.f : T]|\!\}(wp(\eta))(Q) \end{array}$$

For cast we have

$$\begin{array}{ll} & (r,h,s) \in wp([\![\Gamma \vdash (\,U\,)\ x : U]\!](\eta))(Q) \\ \iff & [\![\Gamma \vdash (\,U\,)\ x : U]\!](\eta)(r,h,s) \in Q \\ \iff & (\mathbf{if}\ s\,x = \mathit{null} \vee r(s\,x) \leq U\ \mathbf{then}\ (r,h,[\mathsf{res} : s\,x, \mathsf{exc} : \mathit{null}])\ \mathbf{else}\ \mathit{except}(r,h,U,\mathsf{ClassCast})) \in Q \\ \iff & (r,h,s) \in \{\!|[\Gamma \vdash (\,U\,)\ x : U]|\!\}(wp(\eta))(Q) \end{array}$$

The case of type test is similar to cast.

Here is the case for **let**.

$$\begin{array}{ll} & (r,h,s) \in wp([\![\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ E\ \mathbf{in}\ E1 : U]\!](\eta))(Q) \\ \iff & \quad \text{refactoring (54)} \\ & (r,h,s) \in wp(\langle([\![\Gamma \vdash E : T]\!](\eta)\,;\ \mathit{rename})\,,\ \mathit{id}\rangle;\ \mathit{doE1})(Q) \\ \iff & \quad wp \text{ distribution Lemma A.1} \\ & (r,h,s) \in wp(\langle([\![\Gamma \vdash E : T]\!](\eta)\,;\ \mathit{rename})\,,\ \mathit{id}\rangle)(wp(\mathit{doE1})(Q)) \end{array}$$

To proceed further, first observe the following.

$$\begin{array}{ll} & wp(doE1)(Q) \\ = & \quad \text{definition of } doE1 \\ & wp(\text{IF } null(\mathsf{exc}) \text{ THEN } init\,;\, [\![\Gamma, x:T \vdash E1 : U]\!](\eta) \text{ ELSE } propag)(Q) \\ = & \quad \text{distribution Lemma A.1 for if and seq} \\ & (null(\mathsf{exc}) \cap wp(init)(wp([\![\Gamma, x:T \vdash E1:U]\!](\eta))(Q))) \cup (\neg null(\mathsf{exc}) \cap wp(propag)(Q)) \end{array}$$

Let us abbreviate the last line as $wpDoE1(Q)$ and write $ptDoE1(Q)$ for the following:

$$(null(\mathsf{exc}) \cap wp(init)(\{\![\Gamma, x:T \vdash E1:U]\!\}(wp(\eta)))(Q))) \cup (\neg null(\mathsf{exc}) \cap wp(propag)(Q)) \qquad (59)$$

So the main calculation for **let** continues

$$\begin{array}{ll} & (r,h,s) \in wp(\langle([\![\Gamma \vdash E:T]\!](\eta)\,;\, rename)\,, id\rangle)(wp(doE1)(Q)) \\ \iff & \quad \text{above} \\ & (r,h,s) \in wp(\langle([\![\Gamma \vdash E:T]\!](\eta)\,;\, rename)\,, id\rangle)(wpDoE1(Q)) \\ \iff & \quad \text{induction for } E1 \\ & (r,h,s) \in wp(\langle([\![\Gamma \vdash E:T]\!](\eta)\,;\, rename)\,, id\rangle)(ptDoE1(Q)) \\ \iff & \quad wp \text{ distribution Lemma A.1(58)} \\ & (r,h,s) \in (wp([\![\Gamma \vdash E:T]\!](\eta)\,;\, rename) \oplus id)(ptDoE1(Q)) \\ \iff & \quad wp \text{ distribution, writing ; for sequence of predicate transformers} \\ & (r,h,s) \in (wp([\![\Gamma \vdash E:T]\!](\eta))\,;\, wp(rename)) \oplus id)(ptDoE1(Q)) \end{array}$$

We refrain from developing general algebraic properties of $\oplus id$. We just observe that for any $f$ and $P$ we have

$$\begin{array}{ll} & (r,h,s) \in ((f\,;\, wp(rename)) \oplus id)(P) \\ \iff & \quad \text{definition of } \oplus \text{ and ;} \\ & (r,h,s) \in f(wp(rename)(P \oplus s)) \\ \iff & \quad \text{definition below} \\ & (r,h,s) \in f(ptRename(s)(P)) \end{array}$$

where we define $ptRename(s)(P) = wp(rename)(P \oplus s)$. Taking $f := wp([\![\Gamma \vdash E:T]\!](\eta))$ and $P := ptDoE1(Q)$, the main calculation can conclude

$$\begin{array}{ll} & (r,h,s) \in (wp([\![\Gamma \vdash E:T]\!](\eta))\,;\, wp(rename)) \oplus id)(ptDoE1(Q)) \\ \iff & \quad \text{above} \\ & (r,h,s) \in wp([\![\Gamma \vdash E:T]\!](\eta))(ptRename(s)(ptDoE1(Q))) \\ \iff & \quad \text{induction for } E \\ & (r,h,s) \in \{\![\Gamma \vdash E:T]\!\}(wp(\eta))(ptRename(s)(ptDoE1(Q))) \\ \iff & \quad \text{definition below} \\ & (r,h,s) \in \{\![\Gamma \vdash \textbf{let } x \textbf{ be } E \textbf{ in } E1:U]\!\}(wp(\eta))(Q) \end{array}$$

For **if** $x$ **then** $C_1$ **else** $C_2$ the derivation is

$$\begin{array}{ll} & \sigma \in wp([\![\textbf{if } x \textbf{ then } C_1 \textbf{ else } C_2]\!](\eta))(Q) \\ \iff & \quad \text{definition of wp} \\ & [\![\textbf{if } x \textbf{ then } C_1 \textbf{ else } C_2]\!](\eta)(\sigma) \in Q \\ \iff & \quad \text{semantics} \\ & \textbf{if } \sigma(x) = true \textbf{ then } [\![C_1]\!](\eta)(\sigma) \in Q \textbf{ else } [\![C_2]\!](\eta)(\sigma) \in Q \\ \iff & \quad \text{definition of wp, twice} \\ & \textbf{if } \sigma(x) = true \textbf{ then } \sigma \in wp([\![C_1]\!](\eta))(Q) \textbf{ else } \sigma \in wp([\![C_2]\!](\eta))(Q) \\ \iff & \quad \text{induction, twice} \\ & \textbf{if } \sigma(x) = true \textbf{ then } \sigma \in \{\![C_1]\!\}(wp(\eta))(Q) \textbf{ else } \sigma \in \{\![C_2]\!\}(wp(\eta))(Q) \\ \iff & \quad \text{definition of } \{\![\textbf{if } x \textbf{ then } C_1 \textbf{ else } C_2]\!\} \\ & \sigma \in \{\![\textbf{if } x \textbf{ then } C_1 \textbf{ else } C_2]\!\}(wp(\eta))(Q) \end{array}$$

For **throw** $x$ and $x.f := x$ the derivations are similar to those for other primitives. For **var** $x : T$ **in** $C$ the derivation is similar to that for assignment, but simpler as the refactoring involves only sequencing, not store pairing. For **try** $C$ **catch**$(x : T)$ $C$ the refactoring is similar to that for sequence together with **var**; we omit it along with derivation of predicate transformer semantics.

### A.7. Proof of Lemma 7.6 in Sect. 7.3

The left side of (33) is refined by the right side due to Lemma 7.5 and the meet property Eq. (32)). For predicate transformers, refinement is antisymmetric, hence it remains to prove the reverse: $\{\!|(J, pre, post)|\!\} \sqsupseteq (\dot{\sqcap}\varphi \mid \varphi \models (J, pre, post) \cdot wp(\varphi))$.

By definition (27) of $\sqsupseteq$ we must show for all $\sigma, Q$ that

$$\sigma \in (\dot{\sqcap}\varphi \mid \varphi \models (J, pre, post) \cdot wp(\varphi))(Q) \;\Rightarrow\; \sigma \in \{\!|(J, pre, post)|\!\}(Q)$$

By definition of $\dot{\sqcap}$, this is the same as

$$(\forall \varphi \cdot \varphi \models (J, pre, post) \;\Rightarrow\; \sigma \in wp(\varphi)(Q)) \;\Rightarrow\; \sigma \in \{\!|(J, pre, post)|\!\}(Q) \tag{60}$$

To show (60), assume the antecedent. According to Def. 7.3, the consequent is equivalent to

$$(\exists j \cdot \sigma \in pre_j) \wedge (\forall \tau \cdot (\forall i \cdot \sigma \in pre_i \;\Rightarrow\; \tau \in post_i) \;\Rightarrow\; \tau \in Q) \tag{61}$$

We prove the first conjunct by contradiction. Choose any state transformer $\varphi$ that satisfies $(J, pre, post)$. Suppose there is no $j$ with $\sigma \in pre_j$. Then $[\varphi \mid \sigma : \bot]$ also satisfies $(J, pre, post)$. Now $\sigma$ is not in $wp([\varphi \mid \sigma : \bot])(Q)$ which contradicts our assumed antecedent of (60).

To prove the second conjunct of (61), consider any state $\tau$. We must show

$$(\forall i \cdot \sigma \in pre_i \;\Rightarrow\; \tau \in post_i) \;\Rightarrow\; \tau \in Q$$

Choose any $\varphi$ that satisfies the specification $(J, pre, post)$. If $(\forall i \cdot \sigma \in pre_i \Rightarrow \tau \in post_i)$ then $[\varphi \mid \sigma : \tau]$ also satisfies the specification. By the antecedent of (60) we get $\sigma \in wp([\varphi \mid \sigma : \tau])(Q)$, which by definition of $wp$ is $[\varphi \mid \sigma : \tau](\sigma) \in Q$. This simplifies to $\tau \in Q$, which concludes the proof.

### A.8. Proof of Lemma 7.10 in Sect. 7.3

Observe first that for any $(J, pre, post)$ and $\sigma$ in $State([\Gamma \mid \mathsf{self} : T])$ and $Q$ we have

$$\begin{array}{ll}
 & \sigma \in \{\!|(J, pre, post)\lfloor T|\!\}(Q) \\
\iff & \quad \text{definition of } \lfloor T, \text{ let } pre'_j = pre_j \lfloor T \\
 & \sigma \in \{\!|(J, pre', post)|\!\}(Q) \\
\iff & \quad \text{definition 7.3} \\
 & (\exists j \cdot \sigma \in pre'_j) \wedge (\forall \tau \cdot (\forall i \cdot \sigma \in pre'_i \;\Rightarrow\; \tau \in post_i) \;\Rightarrow\; \tau \in Q) \\
\iff & \quad \text{by } \sigma \in pre'_j \text{ iff } selftype(\sigma) = T \text{ (from def of } pre' \text{ in first step) and } \sigma \in pre_j \\
 & (\exists j \cdot \sigma \in pre_j \wedge selftype(\sigma) = T) \\
 & \wedge (\forall \tau \cdot (\forall i \cdot \sigma \in pre_i \wedge selftype(\sigma) = T \;\Rightarrow\; \tau \in post_i) \;\Rightarrow\; \tau \in Q) \\
\iff & \quad \text{predicate calculus, range of } j \in J \text{ nonempty (by Def. 5.1 of specifications)} \\
 & selftype(\sigma) = T \wedge (\exists j \cdot \sigma \in pre_j) \\
 & \wedge (\forall \tau \cdot (\forall i \cdot \sigma \in pre_i \wedge selftype(\sigma) = T \;\Rightarrow\; \tau \in post_i) \;\Rightarrow\; \tau \in Q) \\
\iff & \quad \text{predicate calculus (range diffusion)} \\
 & selftype(\sigma) = T \wedge (\exists j \cdot \sigma \in pre_j) \wedge (\forall \tau \cdot (\forall i \cdot \sigma \in pre_i \;\Rightarrow\; \tau \in post_i) \;\Rightarrow\; \tau \in Q) \\
\iff & \quad \text{semantics} \\
 & selftype(\sigma) = T \wedge \sigma \in \{\!|(J, pre, post)|\!\}(Q)
\end{array}$$

Now to prove the Lemma we calculate

$$\begin{array}{ll}
 & spec_1 \sqsupseteq^T spec_0 \\
\iff & \quad \text{definition of } \sqsupseteq^T \\
 & spec_1 \sqsupseteq spec_0 \lfloor T \\
\iff & \quad \text{Lemma 7.7, } spec_1 \text{ is satisfiable} \\
 & \{\!|spec_1|\!\} \sqsupseteq \{\!|spec_0 \lfloor T|\!\} \\
\iff & \quad \text{definition } \sqsupseteq \\
 & \forall \sigma, Q \cdot \sigma \in \{\!|spec_1|\!\}(Q) \Leftarrow \sigma \in \{\!|spec_0 \lfloor T|\!\}(Q) \\
\iff & \quad \text{observation above} \\
 & \forall \sigma, Q \cdot \sigma \in \{\!|spec_1|\!\}(Q) \Leftarrow (selftype(\sigma) = T \wedge \sigma \in \{\!|spec_0|\!\}(Q)) \\
\iff & \quad \text{predicate calculus} \\
 & \forall \sigma, Q \cdot selftype(\sigma) = T \;\Rightarrow\; (\sigma \in \{\!|spec_1|\!\}(Q) \Leftarrow \sigma \in \{\!|spec_0|\!\}(Q)) \\
\iff & \quad \text{Definition 7.9} \\
 & \{\!|spec_1|\!\} \sqsupseteq^T \{\!|spec_0|\!\}
\end{array}$$

### A.9. Remarks on conjunctivity

The proof of Lemma 7.7 uses the meet operator, in a way that might seem relevant to proving other results. This subsection makes some observations about what can and cannot be done using meets.

It is straightforward to show for any state transformer $\varphi$ that $wp(\varphi)$ is ***universally disjunctive***, i.e., it distributes over arbitrary unions of predicates. This is characteristic of predicate transformers that model deterministic programs, and is not crucial for our results. Also, $wp(\varphi)$ is ***positively conjunctive***, i.e., distributes over the intersection of any non-empty set of predicates: That is, for non-empty $J$ we have

$$wp(\varphi)(\cap i \mid i \in J \cdot Q_i) = (\cap i \mid i \in J \cdot wp(\varphi)(Q_i)).$$

FACT A.3. Let $\mathcal{P}$ be a phrase-in-context. If $\theta(T, m)$ is positively conjunctive for every pair $T, m$ then so are $\mathcal{S}\{\!|\mathcal{P}|\!\}(\theta)$ and $\mathcal{D}\{\!|\mathcal{P}|\!\}(\theta)$.

PROOF. Straightforward induction on $\mathcal{P}$. We consider just the case of method call in the static dispatch semantics. For any state $(r, h, s)$, non-empty $I$, and $I$-indexed family $Q$,

$$\begin{array}{ll}
 & (r,h,s) \in \mathcal{S}\{\!|\Gamma \vdash x.m(\overline{y}) : U|\!\}(\theta)(\cap i \mid i \in I \cdot Q_i) \\
\iff & \quad \text{semantics (Fig. 6)} \\
 & \quad \textbf{if } s\,x = \mathit{null} \textbf{ then } \mathit{except}(r,h,U,\mathsf{NullDeref}) \in (\cap i \mid i \in I \cdot Q_i) \\
 & \quad \textbf{else let } T = \Gamma\,x \textbf{ in let } \overline{z} = \mathit{formals}(T,m) \textbf{ in let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in} \\
 & \qquad (r,h,s_1) \in \theta(T,m)(\cap i \mid i \in I \cdot Q_i) \\
\iff & \quad \text{conjunctivity of } \theta(T,m),\ I \text{ nonempty} \\
 & \quad \textbf{if } s\,x = \mathit{null} \textbf{ then } (\forall i \cdot i \in I \Rightarrow \mathit{except}(r,h,U,\mathsf{NullDeref}) \in Q) \\
 & \quad \textbf{else let } T = \Gamma\,x \textbf{ in let } \overline{z} = \mathit{formals}(T,m) \textbf{ in let } s_1 = [\mathsf{self} : s\,x, \overline{z : s\,y}] \textbf{ in} \\
 & \qquad (\forall i \cdot i \in I \Rightarrow (r,h,s_1) \in \theta(T,m)(Q_i)) \\
\iff & \quad \text{logic, semantics} \\
 & (\forall i \cdot i \in I \Rightarrow (r,h,s) \in \mathcal{S}\{\!|\Gamma \vdash x.m(\overline{y}) : U|\!\}(\theta)(Q_i))
\end{array}$$

For the other primitive commands, with semantics independent from the method environment, the result essentially follows from Lemma 7.1 and the fact that $wp(\varphi)$ is positively conjunctive for any state transformer $\varphi$. □

By similar reasoning one can check that $\{\!|\Gamma \vdash C|\!\}(\theta)$ is $\emptyset$-strict, or that it distributes over arbitrary unions, provided every $\theta(T, m)$ has the same property. This is because junctivity is preserved by control structures and the language's primitives are deterministic —with the exception of method call when the environment contains nondeterministic methods, as may happen when the environment is derived from specifications.

In light of Fact A.3 and our earlier observation that $\{\!|spec|\!\}$ is positively conjunctive for any *spec*, we have that $\{\!|\Gamma \vdash C|\!\}(\{\!|ST|\!\})$ is positively conjunctive for any $\Gamma \vdash C$, and also $\emptyset$-strict provided that $ST$ is satisfiable.

Meets lift pointwise, i.e., for any set $X$ of method environments we define

$$(\dot{\sqcap}\theta \mid \theta \in X \cdot \theta)(T, m) = (\dot{\sqcap}\theta \mid \theta \in X \cdot \theta(T, m))$$

This inherits the meet property, i.e., $(\dot{\sqcap}\theta \mid \theta \in X \cdot \theta) \sqsupseteq \theta' \iff \forall \theta \cdot \theta \in X \Rightarrow \theta \sqsupseteq \theta'$ for all $X$ and all $\theta'$, because refinement of method environments is pointwise.

LEMMA A.4. If $ST$ is satisfiable then $\{\!|ST|\!\} = (\dot{\sqcap}\eta \mid \eta \models ST \cdot wp(\eta))$.

PROOF. Note that $ST$ is satisfiable just if each $ST(T, m)$ is. Because refinement and meet of method environments are defined pointwise, the result is a direct consequence of Lemma 7.6. □

A non-empty meet of positively conjunctive predicate transformers is positively conjunctive.

For any phrase-in-context $\mathcal{P}$ and any set $X$ of environments, monotonicity of $\{\!|\mathcal{P}|\!\}(-)$ (Lemma 7.2) yields the refinement

$$(\dot{\sqcap}\theta \mid \theta \in X \cdot \{\!|\mathcal{P}|\!\}(\theta)) \sqsupseteq \{\!|\mathcal{P}|\!\}(\dot{\sqcap}\theta \mid \theta \in X \cdot \theta) \tag{62}$$

which holds for both static dispatch $\mathcal{S}\{\!|-|\!\}$ and dynamic $\mathcal{D}\{\!|-|\!\}$.

In light of Lemma A.4 it would be nice if the refinement strengthened to an equality. But it is not the case that $\{\!|C|\!\}(-)$ distributes over the meet of method environments, even in case $X$ is non-empty and for every $\theta \in X$, every $\theta(T, m)$ is positively conjunctive.

*Example* A.5. Consider the command $x : \mathsf{int}, y : \mathsf{int} \vdash C$, with $C \equiv x := \mathsf{self}.m(); y := \mathsf{self}.m()$, in a class table with a single class $K$ and method $m$. Let $\theta_0$ (resp. $\theta_1$) be the method environment $wp(\eta_0)$ where $\eta_0(K, m)$ (resp. $\eta_1(K, m)$) always returns 0 (resp. 1). Let $Q$ be the set of states where $x = y$. Now $(\theta_0 \dot{\sqcap} \theta_1)$ makes $m$ nondeterministically choose between 0 and 1, so the command does not establish $x = y$. That is, $\{\!|C|\!\}(\theta_0 \dot{\sqcap} \theta_1)(Q)$ is the empty set. On the other hand, for $i = 0$ and $i = 1$, $\{\!|C|\!\}(\theta_i)(Q)$ is the set of all states, hence so is $(\{\!|C|\!\}(\theta_0) \dot{\sqcap} \{\!|C|\!\}(\theta_0))(Q)$. So $\{\!|C|\!\}(\theta_0) \dot{\sqcap} \{\!|C|\!\}(\theta_0)$ strictly refines $\{\!|C|\!\}(\theta_0 \dot{\sqcap} \theta_1)$.

### A.10. Proof of Lemma 8.11 in Sect. 8.2

By induction on derivation of typing $\Gamma \vdash E$ and then by induction on $\Gamma \vdash C$ using the result for expressions. In each command case we must show for arbitrary predicate $Q$ that

$$\mathcal{D}\{\!|\Gamma \vdash C|\!\}(\theta)(Q) \supseteq \mathcal{S}\{\!|\Gamma \vdash C|\!\}(\theta)(Q)$$

Some cases in the proof expand this set inclusion to the level of states:

$$\forall \sigma \cdot \sigma \in \mathcal{D}\{\!|\Gamma \vdash C|\!\}(\theta)(Q) \Leftarrow \sigma \in \mathcal{S}\{\!|\Gamma \vdash C|\!\}(\theta)(Q)$$

Starting with expressions, the case $x : T$ need not even be expanded to the level of predicates: we have $\mathcal{D}\{\!|\Gamma \vdash x : T|\!\}(\theta) = \mathcal{S}\{\!|\Gamma \vdash x : T|\!\}(\theta)$ directly from the semantics, which are the same (see Fig. 6).

The argument is the same for every primitive expression or command other than method call (that is: literals, cast, type test, equality test, field access and update, **new**, **throw**).

For an assignment command $\Gamma \vdash x := E$ we argue at the level of states as needed to use the semantic definition:

$$\begin{array}{ll} & (r, h, s) \in \mathcal{D}\{\!|\Gamma \vdash x := E|\!\}(\theta)(Q) \\ \iff & \quad \text{semantic definition (Fig. 6)} \\ & (r, h, s) \in \mathcal{D}\{\!|\Gamma \vdash E : T|\!\}(\theta)(wp(rename)(wp(assg)(Q) \oplus s)) \\ \Leftarrow & \quad \text{the result for expressions} \\ & (r, h, s) \in \mathcal{S}\{\!|\Gamma \vdash E : T|\!\}(\theta)(wp(rename)(wp(assg)(Q) \oplus s)) \\ \iff & \quad \text{semantic definition} \\ & (r, h, s) \in \mathcal{S}\{\!|\Gamma \vdash x := E|\!\}(\theta)(Q) \end{array}$$

For a method call, $\Gamma \vdash x.m(\overline{y}) : U$, we distinguish two sub-cases, according to the condition in the semantics. In case $s\, x = \mathit{null}$ we have

$$\begin{array}{ll} & (r, h, s) \in \mathcal{D}\{\!|x.m(\overline{y})|\!\}(\theta)(Q) \\ \iff & \quad \text{using } s\, x = \mathit{null}\text{, semantic definition for } \mathcal{D} \\ & \mathit{except}(r, h, U, \mathsf{NullDeref}) \in Q \\ \iff & \quad \text{using } s\, x = \mathit{null}\text{, semantic definition for } \mathcal{S} \\ & (r, h, s) \in \mathcal{S}\{\!|x.m(\overline{y})|\!\}(\theta)(Q) \end{array}$$

In case $s\, x \neq \mathit{null}$ we have the following, where the static type $\Gamma(x)$ is $T$.

$$\begin{array}{ll} & (r, h, s) \in \mathcal{D}\{\!|x.m(\overline{y})|\!\}(\theta)(Q) \\ \iff & \quad \text{using } s\, x \neq \mathit{null}\text{, semantic definition for } \mathcal{D} \\ & \textbf{let } K = r(s\, x) \textbf{ in let } \overline{z} = \mathit{formals}(K, m) \textbf{ in} \\ & \textbf{let } s_1 = [\mathsf{self} : s\, x,\ \overline{z : s\, y}] \textbf{ in } (r, h, s_1) \in \theta(K, m)(Q) \\ \Leftarrow & \quad \text{assumption Eq. (36), i.e. } \theta(K, m) \sqsupseteq^K \theta(T, m) \\ & \textbf{let } K = r(s\, x) \textbf{ in let } \overline{z} = \mathit{formals}(K, m) \textbf{ in} \\ & \textbf{let } s_1 = [\mathsf{self} : s\, x,\ \overline{z : s\, y}] \textbf{ in } (r, h, s_1) \in \theta(T, m)(Q) \\ \iff & \quad \text{logic, invariance of formal names (item (W4) in def. of w.f. class table in Sect. 4.2)} \\ & \textbf{let } T = \Gamma\, x \textbf{ in let } \overline{z} = \mathit{formals}(T, m) \textbf{ in} \\ & \textbf{let } s_1 = [\mathsf{self} : s\, x,\ \overline{z : s\, y}] \textbf{ in } (r, h, s_1) \in \theta(T, m)(Q) \\ \iff & \quad s\, x \neq \mathit{null}\text{, semantic definition for } \mathcal{S} \\ & (r, h, s) \in \mathcal{S}\{\!|x.m(\overline{y})|\!\}(\theta)(Q) \end{array}$$

The case of **if** $x$ **then** $C_1$ **else** $C_2$ is straightforward owing to the desugared syntax in which the guard condition is a boolean variable. We argue by sub-cases on whether $x$ is true or false, the only

possible values. In case $x$ is true we have

$$\begin{array}{ll} & (r,h,s) \in \mathcal{D}\{\!|\textbf{if } x \textbf{ then } C_1 \textbf{ else } C_2|\!\}(\theta)(Q) \\ \iff & \quad \text{semantics, } s\,x = \mathit{true} \\ & (r,h,s) \in \mathcal{D}\{\!|C_1|\!\}(\theta)(Q) \\ \Leftarrow & \quad \text{induction for } C_1 \\ & (r,h,s) \in \mathcal{S}\{\!|C_1|\!\}(\theta)(Q) \\ \iff & \quad \text{semantics, } s\,x = \mathit{true} \\ & (r,h,s) \in \mathcal{S}\{\!|\textbf{if } x \textbf{ then } C_1 \textbf{ else } C_2|\!\}(\theta)(Q) \end{array}$$

The other subcase is symmetric.

The case of **var** $x : T$ **in** $C$ is also a straightforward use of induction and is left to the reader.

The remaining cases are $C_1;\ C_2$, **let** $x$ **be** $E$ **in** $E1$, and **try** $C_1$ **catch**$(x : T)\ C_2$. These are all forms of sequencing that branch on whether the first phrase throws an exception. They differ in that **let** $x$ **be** $E$ **in** $E1$ binds a variable to a non-exception value and **try** $C_1$ **catch**$(x : T)\ C_2$ binds a variable to an exception value. Moreover the second constituent in $C_1;\ C_2$ and **let** $x$ **be** $E$ **in** $E1$ is executed only in absence of an exception, whereas in **try** $C_1$ **catch**$(x : T)\ C_2$ it is executed only in the presence of an exception. The implicit branching complicates the semantics. It is most transparent in the case of $C_1;\ C_2$ which does not involve variable binding. For this case, observe

$$\begin{array}{ll} & \mathcal{D}\{\!|C_1;\ C_2|\!\}(\theta)(Q) \\ = & \quad \text{semantics} \\ & \mathcal{D}\{\!|C_1|\!\}(\theta)((\mathit{noX} \cap \mathit{wp}(\mathit{dropExc})(\mathcal{D}\{\!|C_2|\!\}(\theta)(Q))) \cup (\neg \mathit{noX} \cap Q)) \\ \supseteq & \quad \text{induction for } C_2\text{, monotonicity of } \mathit{wp}(\mathit{dropExc}), \cap, \cup, \text{ and } \mathcal{D}\{\!|C_1|\!\}(\{\!|CT|\!\}) \\ & \mathcal{D}\{\!|C_1|\!\}(\theta)((\mathit{noX} \cap \mathit{wp}(\mathit{dropExc})(\mathcal{S}\{\!|C_2|\!\}(\theta)(Q))) \cup (\neg \mathit{noX} \cap Q)) \\ \supseteq & \quad \text{induction for } C_1 \\ & \mathcal{S}\{\!|C_1|\!\}(\theta)((\mathit{noX} \cap \mathit{wp}(\mathit{dropExc})(\mathcal{S}\{\!|C_2|\!\}(\theta)(Q))) \cup (\neg \mathit{noX} \cap Q)) \end{array}$$

The proof for **let** $x$ **be** $E$ **in** $E1$ is similar: the predicate transformer semantics takes more ink, but what matters is that the constituents $\{\!|E|\!\}$ and $\{\!|E1|\!\}$ are in monotonic positions as can be seen by inspection of Fig. 14. Similarly for try/catch.

### A.11. Inadequacy of an alternative definition of supertype abstraction

Here is a variation on the notion of supertype abstraction:

$$\forall \dot{\eta} \in \mathit{XMethEnv} \cdot \dot{\eta} \models \mathit{ST} \;\Rightarrow\; (\mathcal{S}[\![\mathcal{P}]\!](\dot{\eta}) \models \mathit{spec} \;\Rightarrow\; \mathcal{D}[\![\mathcal{P}]\!](\dot{\eta}) \models \mathit{spec}) \tag{63}$$

This is like the form of supertype abstraction in Eq. (35), but differs in that the same method environment is used in both the antecedent and consequent. (To be pedantic, we should apply $\mathcal{D}[\![-]\!]$ to the normal method environment obtained from $\dot{\eta}$ by projecting onto class names, discarding the interface types.)

By predicate calculus, (63) implies condition (35). The implication is strict, as shown by the following, which also shows that —unlike (35)— condition (63) does not follow from behavioral subtyping.

*Example* A.6. Consider a class table with just two classes, $K, L$ with $L < K$. Suppose $K$ declares only a single method

$$\textbf{meth } m() : \mathsf{int}\ \{\ \mathsf{res} := 0\ \}$$

$L$ declares only the override **meth** $m() : \mathsf{int}\ \{\ \mathsf{res} := 1\ \}$. Let $ST(K, m) = (\mathit{true}, \mathit{true}) = ST(L, m)$, and note that $ST$ has robust behavioral subtyping. It is straightforward to check that $\hat{\eta} \models ST$ where $\hat{\eta}$ is denoted by the class table.

Let *spec* be $(\mathit{true}, \mathsf{res} = 0)$, suitable for specifying the method call $\mathsf{self} : K \vdash \mathsf{self}.m()$. We have $\mathcal{S}[\![\mathsf{self} : K \vdash \mathsf{self}.m()]\!](\dot{\eta}) \models (\mathit{true}, \mathsf{res} = 0)$, because regardless of whether the initial state has self of type $K$ or $L$, the static dispatch goes to the implementation in $K$. However, $\mathcal{D}[\![\mathsf{self} : K \vdash \mathsf{self}.m()]\!](\dot{\eta})$ does not satisfy $(\mathit{true}, \mathsf{res} = 0)$. This contradicts (63). It does not contradict Eq. (35). Rather, the antecedent in (35) is falsified for the specification $(\mathit{true}, \mathsf{res} = 0)$ because there are other method environments $\eta$ such that $\eta \models ST$ but not $\mathcal{S}[\![\mathsf{self} : K \vdash \mathsf{self}.m()]\!](\eta) \models (\mathit{true}, \mathsf{res} = 0)$.

### A.12. Proof of Proposition 10.1 in Sect. 10

Similar to the proof of Prop. 5.18(b) in Sect. A.3, we unfold definitions to rewrite item (a) of Prop. 10.1 to

$$\forall i, \sigma \cdot \sigma \in pre_i \;\Rightarrow\; (\forall \varphi \cdot (\forall j, \tau \cdot \tau \in pre'_j \;\Rightarrow\; \varphi(\tau) \in post'_j \cup \{\bot\}) \;\Rightarrow\; \varphi(\sigma) \in post_i \cup \{\bot\})$$

Thus items (a) and (b) are equivalent if the following are equivalent for all $i \in I$ and all $\sigma \in pre_i$:

(a') $\forall \varphi \cdot (\forall j, \tau \cdot \tau \in pre'_j \;\Rightarrow\; \varphi(\tau) \in post'_j \cup \{\bot\}) \;\Rightarrow\; \varphi(\sigma) \in post_i \cup \{\bot\}$
(b') $\forall \tau \cdot (\forall j \cdot \sigma \in pre'_j \;\Rightarrow\; \tau \in post'_j) \;\Rightarrow\; \tau \in post_i$

For arbitrary $i$ and $\sigma$ with $\sigma \in pre_i$ we argue by mutual implication.

To show (a') follows from (b'), consider any $\varphi$. Assume the antecedent in (a'), i.e.,

$$\forall j, \tau \cdot \tau \in pre'_j \;\Rightarrow\; \varphi(\tau) \in post'_j \cup \{\bot\}$$

to show $\varphi(\sigma) \in post_i \cup \{\bot\}$. Instantiate $\tau := \sigma$ to get $\forall j \cdot \sigma \in pre'_j \;\Rightarrow\; \varphi(\sigma) \in post'_j \cup \{\bot\}$. If $\varphi(\sigma) = \bot$ we are done proving $\varphi(\sigma) \in post_i \cup \{\bot\}$. So suppose $\varphi(\sigma)$ is a proper state, in which case we have $\forall j \cdot \sigma \in pre'_j \;\Rightarrow\; \varphi(\sigma) \in post'_j$. Now instantiate (b') by $\tau := \varphi(\sigma)$, and since we established its antecedent we get the consequent, $\varphi(\sigma) \in post_i$.

Now we show that (b') follows from (a'). We can define a state transformer $\psi$, that satisfies $(J, pre', post')$, as follows: For any state $\tau$

$\psi(\tau) = \bot$ if there is no $\rho$ with $\forall j \cdot \tau \in pre'_j \;\Rightarrow\; \rho \in post'_j$
$\psi(\tau) = \rho$ if $\rho$ is chosen such that $\forall j \cdot \tau \in pre'_j \;\Rightarrow\; \rho \in post'_j$

Instantiating (a') with $\varphi := \psi$ we get

$$(\forall j, \tau \cdot \tau \in pre'_j \;\Rightarrow\; \psi(\tau) \in post'_j \cup \{\bot\}) \;\Rightarrow\; \psi(\sigma) \in post_i \cup \{\bot\}$$

The antecedent holds, by definition of $\psi$; so we have $\psi(\sigma) \in post_i \cup \{\bot\}$. To show (b'), consider any $\tau$ and assume the antecedent of (b'), i.e., $(\forall j \cdot \sigma \in pre'_j \;\Rightarrow\; \tau \in post'_j)$. Let $\psi$ be defined as above, choosing, in light of the assumption, $\psi(\sigma) := \tau$. As $\tau$ is a state, $\psi(\sigma) \in post_i \cup \{\bot\}$ implies $\tau \in post_i$.

### A.13. Proof of Lemma 7.5 adapted to partial correctness

Consider any $\varphi$ and $(J, pre, post)$ of the same type. We show that $wlp(\varphi) \sqsupseteq \{\!|J, pre, post|\!\}_l$ iff $\varphi \models_l spec$. Without loss of generality, we assume that $(J, pre, post)$ is broad. We can do this because $\varphi \models_l spec$ iff $\varphi \models_l broaden(spec)$, in accord with Eq. (48).

By (27), the definition of refinement, $wlp(\varphi) \sqsupseteq \{\!|J, pre, post|\!\}_l$ is equivalent to

$$\forall \sigma, Q \cdot \sigma \in wlp(\varphi)(Q) \;\Leftarrow\; \sigma \in \{\!|J, pre, post|\!\}_l(Q)$$

with $Q$ ranging over state sets. This in turn is equivalent, by definition (47) of $wlp$ and definition (50), to

$$\forall \sigma, Q \cdot \varphi(\sigma) \in Q \cup \{\bot\} \;\Leftarrow\; (\forall \tau \cdot (\forall i \cdot \sigma \in pre_i \;\Rightarrow\; \tau \in post_i) \;\Rightarrow\; \tau \in Q) \tag{64}$$

By definition, $\varphi \models_l (J, pre, post)$ is equivalent to

$$\forall k, \sigma \cdot \sigma \in pre_k \;\Rightarrow\; \varphi(\sigma) \in post_k \cup \{\bot\} \tag{65}$$

It remains to prove that (65) is equivalent to (64), which we do by mutual implication.

Assume (64). To show (65), observe that for any $k$ in $J$ and any $\sigma$ in $pre_k$ we can instantiate (64) with $post_k$ for $Q$ to obtain

$$\varphi(\sigma) \in post_k \cup \{\bot\} \;\Leftarrow\; (\forall \tau \cdot (\forall i \cdot \sigma \in pre_i \;\Rightarrow\; \tau \in post_i) \;\Rightarrow\; \tau \in post_k) \tag{66}$$

To prove $\varphi(\sigma) \in post_k \cup \{\bot\}$ it is enough to establish the antecedent, and we observe for any $\tau$ that

$$\begin{array}{ll} & \forall i \cdot \sigma \in pre_i \;\Rightarrow\; \tau \in post_i \\ \Rightarrow & \quad \text{logic (instantiate } i := k) \\ & \sigma \in pre_k \;\Rightarrow\; \tau \in post_k \\ \Rightarrow & \quad \text{using } \sigma \in pre_k \\ & \tau \in post_k \end{array}$$

which concludes the proof of (65).

Now assume (65). To show (64), consider any $\sigma, Q$. If $\varphi(\sigma) = \bot$ then we are done proving (64). We proceed with the case $\varphi(\sigma) \neq \bot$. Assume the antecedent of (64), i.e., $\forall \tau \cdot (\forall i \cdot \sigma \in pre_i \Rightarrow \tau \in post_i) \Rightarrow \tau \in Q$. By broadness of $(J, pre, post)$ we may choose some $j$ such that $\sigma \in pre_j$. Then by (65) we get $\varphi(\sigma) \in post_j \cup \{\bot\}$, whence $\varphi(\sigma) \in post_j$ as we are considering the case that $\varphi(\sigma) \neq \bot$. Thus we can instantiate $\tau$ by the state $\varphi(\sigma)$, to obtain

$$(\forall i \cdot \sigma \in pre_i \Rightarrow \varphi(\sigma) \in post_i) \Rightarrow \varphi(\sigma) \in Q$$

The antecedent of this formula is a direct consequence of (65) and $\varphi(\sigma) \neq \bot$, so we obtain the consequent $\varphi(\sigma) \in Q$ which concludes the proof of (64).

### A.14. Proof of Lemma 7.6 adapted to partial correctness

To show $\{\![J, pre, post]\!\}_l = (\dot\sqcap \varphi \mid \varphi \models_l (J, pre, post) \cdot wlp(\varphi))$ note that the left side is refined by the right side due to (the adapted version of) Lemma 7.5 and the meet property Eq. (32)). For predicate transformers, refinement is antisymmetric, hence it remains to prove the reverse: $\{\![J, pre, post]\!\}_l \sqsupseteq (\dot\sqcap \varphi \mid \varphi \models_l (J, pre, post) \cdot wlp(\varphi))$.

By definition (27) of $\sqsupseteq$ we must show for all $\sigma, Q$ that

$$\sigma \in (\dot\sqcap \varphi \mid \varphi \models_l (J, pre, post) \cdot wlp(\varphi))(Q) \Rightarrow \sigma \in \{\![(J, pre, post)]\!\}_l(Q)$$

By definition of $\dot\sqcap$, this is the same as

$$(\forall \varphi \cdot \varphi \models_l (J, pre, post) \Rightarrow \sigma \in wlp(\varphi)(Q)) \Rightarrow \sigma \in \{\![(J, pre, post)]\!\}_l(Q) \tag{67}$$

To show (67), assume the antecedent. By definition (50), the consequent is equivalent to

$$\forall \tau \cdot (\forall i \cdot \sigma \in pre_i \Rightarrow \tau \in post_i) \Rightarrow \tau \in Q \tag{68}$$

To prove (68), consider any state $\tau$. We must show $(\forall i \cdot \sigma \in pre_i \Rightarrow \tau \in post_i) \Rightarrow \tau \in Q$. Choose any $\varphi$ that satisfies the specification $(J, pre, post)$ in the sense of partial correctness. From $(\forall i \cdot \sigma \in pre_i \Rightarrow \tau \in post_i)$ it follows that $[\varphi \mid \sigma : \tau]$ also satisfies the specification. By the antecedent of (67) we get $\sigma \in wlp([\varphi \mid \sigma : \tau])(Q)$, which by definition of $wlp$ is $[\varphi \mid \sigma : \tau](\sigma) \in Q \cup \{\bot\}$. This simplifies to $\tau \in Q \cup \{\bot\}$, and since $\tau$ is a state we have $\tau \in Q$.

## REFERENCES


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

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

America, P. 1991. Designing an object-oriented programming language with behavioural subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds. LNCS, vol. 489. Springer-Verlag, New York, 60–90.

America, P. and de Boer, F. 1990. Proving total correctness of recursive procedures. *Information and Computation 84,* 2, 129–164.

Apt, K. 1981. Ten years of Hoare's logic, a survey, part I. *ACM Trans. Program. Lang. Syst. 3,* 4, 431–483.

Apt, K. R., de Boer, F. S., and Olderog, E.-R. 2009. *Verification of Sequential and Concurrent Programs*, 3 ed. Springer.

Apt, K. R., de Boer, F. S., Olderog, E.-R., and de Gouw, S. 2012. Verification of object-oriented programs: A transformational approach. *J. Comput. Syst. Sci. 78,* 3, 823–852.

Back, R. 1988. A calculus of refinements for program derivations. *Acta Inf. 25*, 593–624.

Back, R.-J. and von Wright, J. 1998. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag.

Banerjee, A., Heintze, N., and Riecke, J. G. 2001. Design and correctness of program transformations based on control-flow analysis. In *Intl. Symp. on Theoretical Aspects of Computer Software.* 420–447. LNCS 2215.

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

Banerjee, A. and Naumann, D. A. 2013. Local reasoning for global invariants, part II: Dynamic boundaries. *Journal of the ACM 60,* 3.

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

Barnett, M., DeLine, R., Jacobs, B., Fhndrich, M., Leino, K. R. M., Schulte, W., and Venter, H. 2005. The Spec# programming system: Challenges and directions. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, B. Meyer and J. C. P. Woodcock, Eds. Post-proceedings, to appear.

Barnett, M., Leino, K. R. M., and Schulte, W. 2005. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop (CASSIS 2004), Revised Selected Papers*, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds. LNCS, vol. 3362. 49–69.

Beckert, B., Hähnle, R., and Schmitt, P. H. 2007. *Verification of Object-Oriented Software. The KeY Approach: Foreword by K. Rustan M. Leino*. LNCS, vol. 4334. Springer-Verlag.

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

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

Borba, P., Sampaio, A., Cavalcanti, A., and Cornélio, M. 2004. Algebraic reasoning for object-oriented programming. *Sci. Comput. Program. 52,* 1-3, 53–100.

Bruce, K. B. and Wegner, P. 1986. An algebraic model of subtypes in object-oriented languages (draft). *ACM SIGPLAN Notices 21,* 10 (Oct.), 163–172.

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

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

Cheon, Y., Leavens, G. T., Sitaraman, M., and Edwards, S. 2005. Model variables: Cleanly supporting abstraction in design by contract. *Software—Practice and Experience 35,* 6 (May), 583–599.

Chin, W.-N., David, C., Nguyen, H. H., and Qin, S. 2008. Enhancing modular OO verification with separation logic. In *ACM Symposium on Principles of Programming Languages (POPL)*, P. Wadler, Ed. ACM, 87–99.

de Roever, W.-P. and Engelhardt, K. 1998. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press.

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

Dijkstra, E. W. 1976. *A Discipline of Programming*. Prentice-Hall.

Dovland, J., Johnsen, E. B., Owe, O., and Steffen, M. 2008. Lazy behavioral subtyping. In *FM 2008: Formal Methods*. LNCS, vol. 5014. Springer-Verlag, Berlin, 52–67.

ECMA International 2006. *Eiffel: Analysis, Design, and Programming Language*, 2nd Edition ed. ECMA International, Rue du Rhone 114, CH-1204, Geneva.

Findler, R. B. and Felleisen, M. 2001. 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*. 1–15.

Findler, R. B., Latendresse, M., and Felleisen, M. 2001. Behavioral contracts and behavioral subtyping. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. ESEC/FSE-9. ACM, New York, NY, USA, 229–236.

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

Greenberg, M., Pierce, B. C., and Weirich, S. 2010. Contracts made manifest. In *ACM Symposium on Principles of Programming Languages (POPL)*. ACM, New York, NY, 353–364.

Harel, D., Pnueli, A., and Stavi, J. 1977. A complete axiomatic system for proving deductions about recursive programs. In *STOC*. 249–260.

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

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

Kleymann, T. 1999. Hoare logic and auxiliary variables. *Formal Aspects of Computing 11*, 541–566.

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

Leavens, G. T. 1990. Modular verification of object-oriented programs with subtypes. Tech. Rep. 90-09, Department of Computer Science, Iowa State University, Ames, Iowa, 50011. July. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

Leavens, G. T. 2006. JML's rich, inherited specifications for behavioral subtypes. In *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, Z. Liu and H. Jifeng, Eds. LNCS, vol. 4260. Springer-Verlag, New York, NY, 2–34.

Leavens, G. T., Baker, A. L., and Ruby, C. 2006. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes 31,* 3 (Mar.), 1–38.

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

Leavens, G. T. and Naumann, D. A. 2006a. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Tech. Rep. 06-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011. Dec.

Leavens, G. T. and Naumann, D. A. 2006b. Behavioral subtyping, specification inheritance, and modular reasoning. Tech. Rep. 06-20, Department of Computer Science, Iowa State University, Ames, Iowa, 50011. July.

Leavens, G. T., Naumann, D. A., and Rosenberg, S. 2006. Preliminary definition of Core JML. CS Report 2006-07, Stevens Institute of Technology. Sept.

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

Leino, K. 1995. Toward reliable modular programs. Ph.D. thesis, California Institute of Technology. Available as Technical Report Caltech-CS-TR-95-03.

Leino, K. R. M. 2005. Efficient weakest preconditions. *Inf. Process. Lett. 93,* 6, 281–288.

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

Leino, K. R. M. and Nelson, G. 2002. Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst. 24,* 5, 491–553.

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

Liskov, B. and Guttag, J. 2001. *Program Development in Java.* The MIT Press, Cambridge, Mass.

Liskov, B. H. and Wing, J. M. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16,* 6 (Nov.), 1811–1841.

Meyer, B. 1985. Eiffel: A langauge for software engineering. Tech. Rep. TR-CS-85-19, University of California, Santa Barbara, Santa Barbara, Calif. Dec.

Meyer, B. 1997. *Object-oriented Software Construction*, second ed. Prentice Hall, New York.

Morgan, C. 1988. The specification statement. *ACM Trans. Program. Lang. Syst. 10,* 3, 403–419.

Morgan, C. 1994. *Programming from Specifications, second edition.* Prentice Hall.

Müller, P. 2002. *Modular Specification and Verification of Object-Oriented Programs.* LNCS, vol. 2262. Springer-Verlag.

Müller, P., Poetzsch-Heffter, A., and Leavens, G. T. 2006. Modular invariants for layered object structures. *Sci. Comput. Program. 62,* 3 (Oct.), 253–286.

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

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

Naumann, D. A. and Barnett, M. 2006. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theor. Comput. Sci. 365*, 143–168.

Noble, J., Vitek, J., and Potter, J. 1998. Flexible alias protection. In *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, E. Jul, Ed. LNCS, vol. 1445. Springer-Verlag, 158–185.

O'Hearn, P. W., Reynolds, J. C., and Yang, H. 2001. Local reasoning about programs that alter data structures. In *Proceedings of Computer Science Logic.* LNCS, vol. 2142. Springer-Verlag, 1–19.

O'Hearn, P. W., Yang, H., and Reynolds, J. C. 2009. Separation and information hiding. *ACM Trans. Program. Lang. Syst. 31,* 3, 1–50. Extended version of [?].

Oheimb, D. v. and Nipkow, T. 2002. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In *Formal Methods – Getting IT Right (FME'02)*, L.-H. Eriksson and P. A. Lindsay, Eds. LNCS, vol. 2391. Springer, 89–105.

Olderog, E.-R. 1983. On the notion of expressiveness and the rule of adaptation. *Theor. Comput. Sci. 24*, 337–347.

Parkinson, M. and Bierman, G. 2008. Separation logic, abstraction and inheritance. In *ACM Symposium on Principles of Programming Languages (POPL)*, P. Wadler, Ed. ACM, 75–86.

Parkinson, M. J. 2005. Local reasoning for Java. Tech. Rep. 654, University of Cambridge Computer Laboratory. Nov. Dissertation.

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

Pierik, C. and de Boer, F. 2005a. On behavioral subtyping and completeness. In *ECOOP Workshop on Formal Techniques for Java-like Programs.*

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

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

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

Reus, B. 2003. Modular semantics and logics of classes. In *Computer Science Logic (CSL)*, M. Baaz and J. A. Makowsky, Eds. LNCS, vol. 2803. 456–469.

Reynolds, J. C. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*. Vol. 2. ACM, New York, 717–740.

Reynolds, J. C. 1982. Idealized ALGOL and its specification logic. In *Tools and Notions for Program Construction*, D. Néel, Ed. Cambridge Univ. Press, 121–161. Reprinted in ALGOL-like Languages, Volume 1, Birkhauser Boston Inc., 1997, pp 125–156.

Van Roy, P. and Haridi, S. 2004. *Concepts, Techniques, and Models of Computer Programming*. MIT Press.

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

Wing, J. M. 1983. A two-tiered approach to specifying programs. Tech. Rep. TR-299, MIT Lab for Computer Science.

## Contents

## Index