# Modular Reasoning and a Definition of Supertype Abstraction

Gary T. Leavens and David A. Naumann
University of Central Florida and Stevens Inst. of Technology
Support from US National Science Foundation
NII Shonan Workshop on OO Specification and Verification,
September 22, 2015

jmlspecs.org www.cs.ucf.edu/~leavens

# Problem

- Modular reasoning for OO programs
  - **Proving** soundness and completeness
  - In general, without restriction to some particular proof system.

# Approach

- Supertype abstraction

  T x;
  
  // …
  
  $\{pre^T_m[x/\mathbf{self}]\}$ x.m(); $\{post^T_m[x/\mathbf{self}]\}$

- Formalize semantically:
  - **Independent** of program logic

# Contributions in [LN15]: Semantic treatments of:

- Refinement
- Modular correctness
- Supertype abstraction
- Behavioral Subtyping
- Necessity and sufficiency of behavioral subtyping
- Specification inheritance

# Related Work: Liskov 1988 (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 $T$."

Problems:

- "Unchanged" behavior is too restrictive,
- What does substitution mean in OO programs?

# Related Work: Liskov & Wing 1994 (fig. 4)

For $S$ to be a behavioral subtype of $T$,

- Subtype's invariant must imply the supertype's:
  $\forall\ self{:}S\ .\ inv^S(self) \Rightarrow inv^T(self)$
- "Subtype methods preserve the supertype method's behavior."
  For each method m of type T and self:S
  $pre^T_m(self) \Rightarrow pre^S_m(self)$
  $post^S_m(self) \Rightarrow post^T_m(self)$

Problems:

- No proofs of soundness
- Postcondition rule is too strong

# Why Liskov and Wing's postcondition rule is too strong

```
class TrustingAnimal {
   public model int age;
   int ag; represents age := ag;

   meth setAge(int a)
      requires 0 ≤ a ∧ a ≤ 150;
      ensures age = a;
      { ag := a; }              }


public class Animal extends TrustingAnimal {

   meth setAge(int a);
      requires (0 ≤ a ∧ a ≤ 150) || a < 0 ;
      ensures (old(0 ≤ a ∧ a ≤ 150) ⇒ age = a)
            && (old(a < 0) ⇒ age = old(age)) ;
      if (0 ≤ a ∧ a ≤ 150)  then { ag := a }          }
```

# An Idealized Java-like OO Language

- interfaces
- classes
- exceptions as objects
- type tests (**is**) and type casts
- expressions with effects

Omits:

- constructors
- super calls
- concurrency

# Language Semantics overview

- Denotational Semantics
- State transformers
  - Separate state spaces for initial and final states
  - Commands: final state variable exc
  - Expressions: final state variables exc and res
  - Only two kinds of outcomes: $\perp$ or a state
- Two kinds of semantics:
  - Dynamic, models dynamic dispatch
  - Static, models supertype abstraction in reasoning

# Language Grammar (Abstract Syntax)

$$
\begin{array}{lcl}
T & ::= & K \mid I \mid \textsf{bool} \mid \textsf{int} \\
msig & ::= & m(\overline{x : T}) : T \\
mdec & ::= & \textbf{meth}\ msig\ \{\ C\ \} \\
C & ::= & x := E \mid x.f := x \\
 & \mid & \textbf{var}\ x : T\ \textbf{in}\ C \\
 & \mid & C;\ C \mid \textbf{if}\ x\ \textbf{then}\ C\ \textbf{else}\ C \\
 & \mid & \textbf{throw}\ x \mid \textbf{try}\ C\ \textbf{catch}(x : T)\ C \\
E & ::= & x \mid \textsf{null} \mid \textbf{true} \mid 0 \ldots \\
 & \mid & x.f \mid x = y \\
 & \mid & x\ \textbf{is}\ T \mid (T)\ x \mid \textbf{new}\ K() \\
 & \mid & x.m(\overline{x}) \mid \textbf{let}\ x\ \textbf{be}\ E\ \textbf{in}\ E
\end{array}
$$

| | |
|---|---|
| $K, L \in ClassName$ | Names of declared classes |
| $I \in InterfaceName$ | Names of declared interfaces |
| $x, y, z$ | Variable names (for parameters and locals) |
| $f$ | Field names |
| $m$ | Method names |

# Basic Domains

$\Gamma \in TypeContext = Variable \rightharpoonup T$
$S, T, U \in RefType = ClassName \cup InterfaceName$
$o \in Ref$
$r \in RefCtx = Ref \rightharpoonup ClassName$

$o \in dom\ r$ means: $o$ is allocated and has type $r\ o$

*Values:*
$Val(\mathrm{int}, r) = \mathbb{Z}$
$Val(\mathrm{bool}, r) = \{true, false\}$
$Val(K, r) = \{null\} \cup \{o \mid o \in dom\ r \wedge r\ o \leq K\}$
$Val(I, r) = \{null\} \cup \{o \mid \exists K \cdot K \leq I \wedge o \in Val(K, r)\}$

# Stores, Heaps, States, and State Transformers

$$s \in Store(\Gamma, r) \Leftrightarrow s \in ((x : dom\ \Gamma) \to Val(x, r)) \wedge (\mathrm{self} \in dom\ \Gamma \Rightarrow s(\mathrm{self}) = null)$$

$$Obrecord(K, r) = Store(fields\ K, r)$$

$$h \in Heap(r) = (o : dom\ r) \to Obrecord(r\ o, r)$$

$$\sigma \in State(\Gamma) = (r : RefCtx) \times Heap(r) \times Store(\Gamma, r)$$

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

$$extState((r, h, s), (r', h', s')) \Leftrightarrow r \subseteq r'$$

$$imuSelf((r, h, s), (r', h', s')) \Leftrightarrow (\mathrm{self} \in (dom\ s \cap dom\ s') \Rightarrow s(\mathrm{self}) = s'(\mathrm{self})).$$

# Semantics of Expressions, Commands, and Methods

$SemExpr(\Gamma, T) = STrans(\Gamma, [\mathrm{res} : T, \mathrm{exc} : \mathrm{Exc}])$

$SemCommand(\Gamma, \Gamma') = STrans(\Gamma, [\Gamma, \mathrm{exc} : \mathrm{Exc}])$

$$SemMeth(T,m) = STrans([\mathrm{self} : T, z_1{:}U_1,\ldots,z_n{:}U_n], [\mathrm{res} : U, \mathrm{exc} : \mathrm{Exc}])$$

**where** $mtype(T,m) = (z_1{:}U_1,\ldots,z_n{:}U_n) \rightarrow U$

# Method Environments

Normal method environments:

$$\eta \in MethEnv = (K : ClassName) \times (m\colon Meths\ K) \to SemMeth(K,m)$$

Extended method environment:

$$\eta^{\cdot} \in XMethEnv = (T : RefType) \times (m\colon Meths\ T) \to SemMeth(T,m)$$

# Example Semantics Clauses Common to Dynamic and Static

$[[\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ E\ \mathbf{in}\ E1 : U]](\eta)(r, h, s)$

$= \mathbf{lets}\ (r_0, h_0, s_0) = [[\Gamma \vdash E : T]](\eta)(r, h, s)\ \mathbf{in}$

$\quad \mathbf{if}\ s_0\ \mathrm{exc} \neq \mathit{null}$

$\quad \mathbf{then}\ (r_0, h_0, [\mathrm{res} : \mathit{default}\ U, \mathrm{exc} : s_0\ \mathrm{exc}])$

$\quad \mathbf{else}\ \mathbf{let}\ s_1 = [s\ , x : s_0\ \mathrm{res}]\ \mathbf{in}$

$\qquad [[\Gamma, x : T \vdash E1 : U]](\eta)(r_0, h_0, s_1)$

$[[\Gamma \vdash x := E]](\eta)(r, h, s)$

$= \mathbf{lets}\ (r_1, h_1, s_1) = [[\Gamma \vdash E : T]](\eta)(r, h, s)\ \mathbf{in}$

$\quad \mathbf{if}\ s_1\ \mathrm{exc} = \mathit{null}$

$\quad \mathbf{then}\ (r_1, h_1, [\ [s \mid x : s_1\ \mathrm{res}]\ , \mathrm{exc} : \mathit{null}\ ])$

$\quad \mathbf{else}\ (r_1, h_1, [s, \mathrm{exc} : s_1\ \mathrm{exc}]).$

# Dynamic and Static Semantics for method calls

$\mathcal{D}[[\Gamma \vdash x.m(y_1,\ldots,y_n) : U]](\eta)(r, h, s)$

= **if** $s\ x = null$ **then** $except(r, h, U, \mathrm{NullDeref})$

**else let** $K = r(s\ x)$ **in let** $z_1,\ldots,z_n = formals(K,m)$ **in**

**let** $s_1 = [\mathrm{self} : s\ x, z_1{:}s\ y_1,\ldots,z_n{:}s\ y_n]$ **in**

$\eta(K,m)(r, h, s_1)$

$\mathcal{S}[[\Gamma \vdash x.m(y_1,\ldots,y_n) : U]](\eta')(r, h, s)$

= **if** $s\ x = null$ **then** $except(r, h, U, \mathrm{NullDeref})$

**else let** $T = \Gamma\ x$ **in let** $z_1,\ldots,z_n = formals(T,m)$ **in**

**let** $s_1 = [\mathrm{self} : s\ x, z_1{:}s\ y_1,\ldots,z_n{:}s\ y_n]$ **in**

$\eta'(T,m)(r, h, s_1)$

# Approximation Orderings

On State Transformers $\phi$ and $\psi$ in $STrans(\Gamma, \Gamma')$:

define $\phi \preccurlyeq \psi$
if and only if for all $\sigma$ in $State(\Gamma)$,
either $\phi\ \sigma = \psi\ \sigma$ or $\phi\ \sigma = \bot$.

On Method Environments $\eta$ and $\eta'$:

define $\eta \preccurlyeq \eta'$ if and only if
$\eta(K,m) \preccurlyeq \eta'(K,m)$, for all $K,m$.

Dynamic semantics of class tables: $\mathcal{D}[[CT]]$
is lub of chains of method environments

# Specification Semantics Basics

- Semantics, not syntax
- One-state predicate on $\Gamma$-*states* = $\wp(State(\Gamma))$

General specifications of methods:

def: A ***general specification of type*** $\Gamma \rightsquigarrow \Gamma'$ is a triple ($J$, *pre*, *post*) consisting of:

a nonempty set $J$ and

$J$-indexed families of predicates:

$pre \in J \to \wp(State(\Gamma))$ and

$post \in J \to \wp(State(\Gamma'))$.

# Relation to Two-State Specifications

Consider a method specification of the form:

**requires** $0 \leq age \wedge age < 150$;
**ensures** $age = \mathbf{old}(age+1)$;

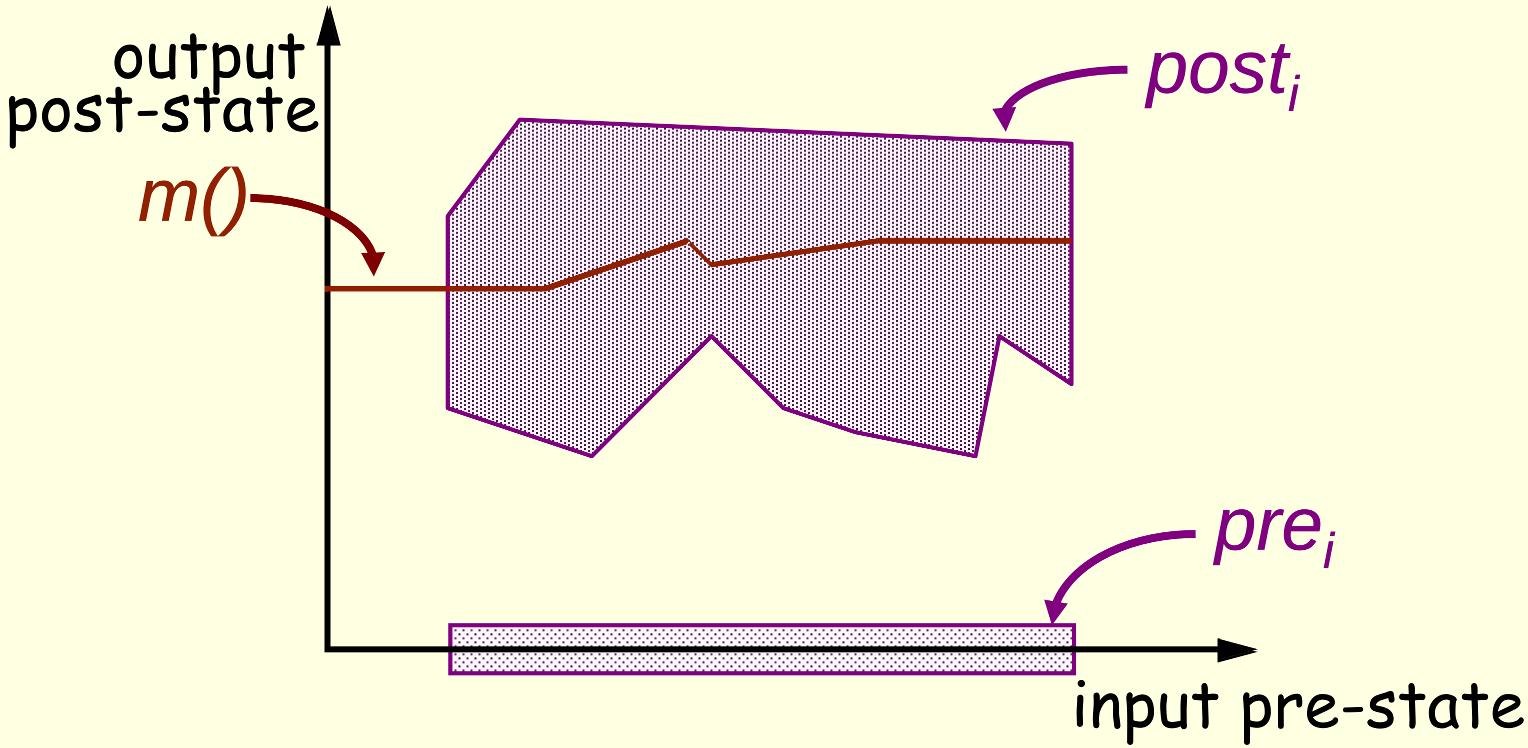
Can encode this as the general specification of type [age: int] $\rightsquigarrow$ [exc: Exc] with index set [age: int]-States: $(\wp(State([age{:}int])), pre_\sigma, post_\sigma)$
*where* $pre_\sigma = \{\tau \mid \sigma = \tau \wedge \sigma = (r,h,s)$
$\wedge\ 0 \leq s(age) \wedge s(age) < 150\}$
*and* $post_\sigma = \{\tau \mid \sigma = (r,h,s) \wedge \tau = (r',h',s')$
$\wedge\ s'(age) = s(age)+1\ \}$

# Satisfaction (total correctness) for General Specifications

def: $\phi$ |= ($J$, $pre$, $post$) if and only if for all $i \in J$,

$$\forall \sigma \cdot \sigma \in pre_i \Rightarrow \phi(\sigma) \in post_i.$$

# Correctness for Method Specifications



# Intrinsic Refinement of General Specifications

Idea: Subtype's (stronger) specifications have implementations that can be used in place of those of supertype's (weaker) specifications.

Problem:

Subtype's specification knows that **self** has its subtype (or lower).

Thus type of **self** changes covariantly!

So types of the corresponding state transformers are not related by subtyping!

# Dealing with type of **self**

- Two flavors:
  - Exact: **self** has exactly the subtype
  - Downward: **self** has the subtype or lower

*Define:*

$$selftype(r, h, s) = r(s(\mathbf{self}))$$

$$\sigma \in pre{\downharpoonleft}T \Leftrightarrow selftype(\sigma) = T \land \sigma \in pre$$

$$\sigma \in pre{\downharpoonleft}^{*}T \Leftrightarrow selftype(\sigma) \leq T \land \sigma \in pre$$

# Refinement (standard)

Let $spec_0 : \Gamma \rightsquigarrow \Gamma'$ and $spec_1 : \Delta \rightsquigarrow \Delta'$,

where $\Delta \rightsquigarrow \Delta' \leq \Gamma \rightsquigarrow \Gamma'$ (i.e., $\Gamma \leq \Delta$ and $\Delta' \leq \Gamma'$).

Then $spec_1$ ***refines*** $spec_0$, written $spec_1 \sqsupseteq spec_0$,

if and only if for all $\phi \in STrans(\Delta, \Delta')$,

$\phi \models spec_1 \Rightarrow \phi \models spec_0$

# Refinement at a Subtype

Let $spec_0 : \Gamma \rightsquigarrow \Gamma'$ and $spec_2 : [\Delta \mid \mathbf{self} : S] \rightsquigarrow \Delta'$.

where $\Delta \rightsquigarrow \Delta' \leq \Gamma \rightsquigarrow \Gamma'$ (i.e., $\Gamma \leq \Delta$ and $\Delta' \leq \Gamma'$)

and $S \leq \Gamma$ **self**.

***Refinement at exact subtype*** $S$, $spec_2 \sqsupseteq^S spec_0$, is defined by

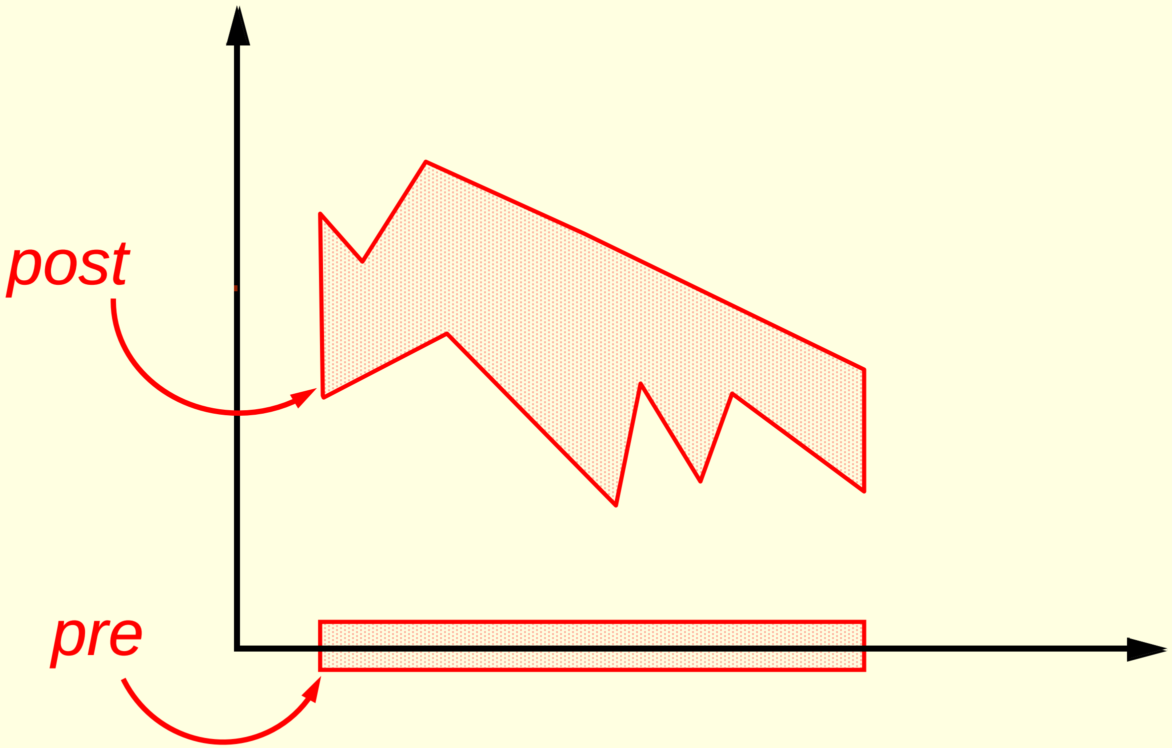
$$spec_2 \sqsupseteq^S spec_0 \Leftrightarrow spec_2 \sqsupseteq spec_0 \downharpoonleft S.$$

***Refinement at a downward subtype*** $S$,

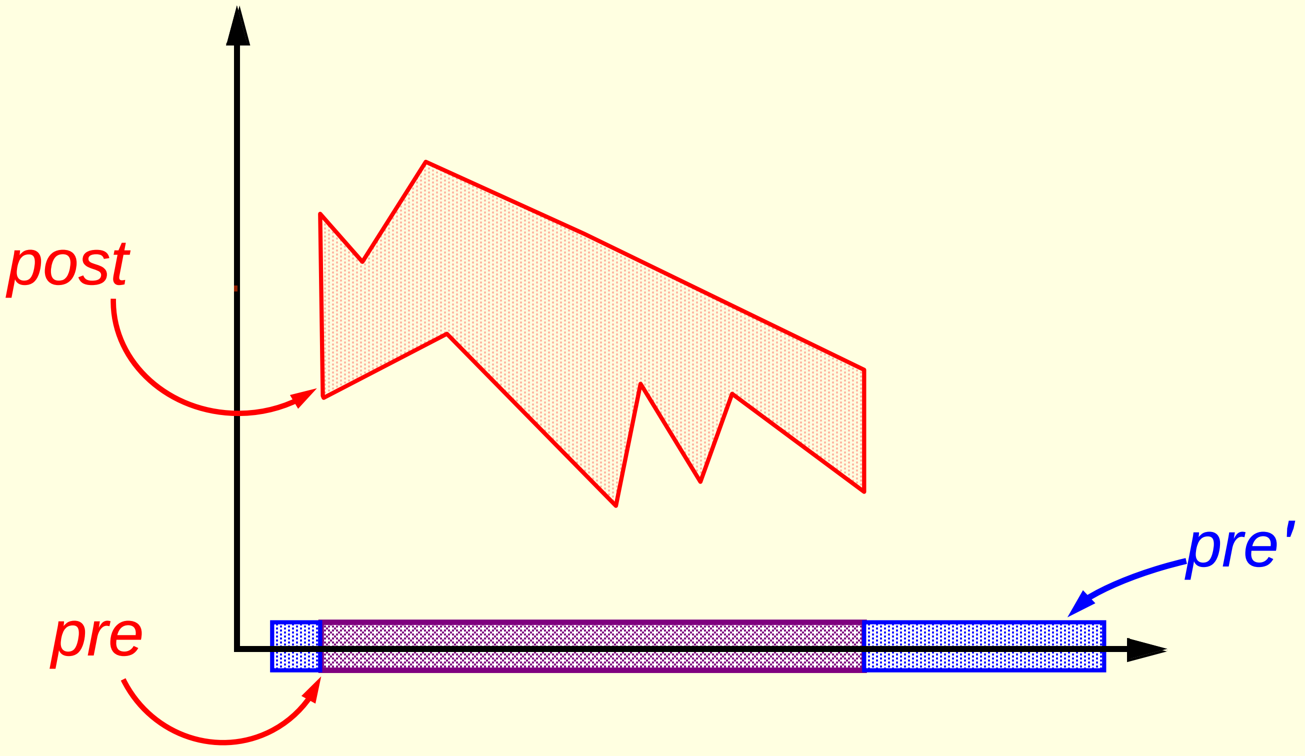
$spec_2 \sqsupseteq^{*S} spec_0$, is defined by

$$spec_2 \sqsupseteq^{*S} spec_0 \Leftrightarrow spec_2 \sqsupseteq spec_0 \downharpoonleft^* S.$$

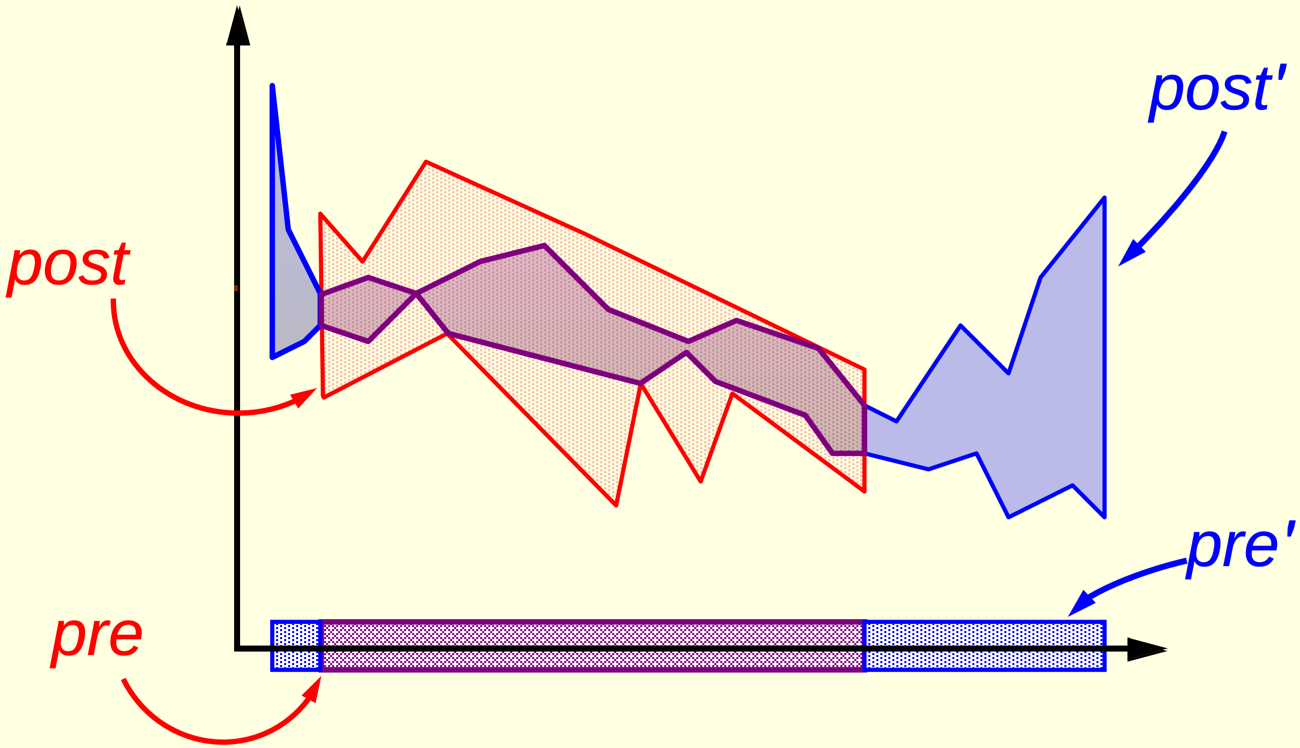
# Refinement at type $S$



# Refinement at type $S$



# Refinement at type $S$



# Characterization of Refinement at a subtype

Suppose that $(I, pre, post) : \Gamma \rightsquigarrow \Gamma'$ and $(J, pre', post') : spec_2 : [\Delta \mid \mathbf{self} : S] \rightsquigarrow \Delta'$, where $S \leq \Gamma\,\mathbf{self}$ and $\Delta \rightsquigarrow \Delta' \leq \Gamma \rightsquigarrow \Gamma'$.
If $(J, pre', post')$ is satisfiable, then the following are equivalent:

(a) $(J, pre', post') \sqsupseteq^{S} (I, pre, post)$

(b) $\forall i \in I, \sigma \in State(\Gamma) \cdot \sigma \in pre_i \downharpoonleft S$

$$\Rightarrow (\exists\, j \in J \cdot \sigma \in pre'_j)$$
$$\wedge\ (\forall \tau \in State(\Delta')$$
$$\cdot\ (\forall\, k \in J \cdot \sigma \in pre'_k \Rightarrow \tau \in post'_k)$$
$$\Rightarrow \tau \in post_i).$$

# Modular Correctness

Modular verifiers and proof systems:

- Focus on one method at a time
- Assume specification of all other methods

# Domains for Modular Correctness

$CT \in ClassTable =$

$(K{:}ClassName) \times (m{:}MethodName)$

$\rightarrow SemMeth(K,m)$

$ST \in SpecTable =$

$(T{:}RefType) \times (m{:}MethodName)$

$\rightarrow ([\mathbf{self}{:}\mathrm{T}, formals(T,m)]$

$\rightsquigarrow [res{:}\ resType(T,m), exc{:}Exc])$

# Satisfaction for Spec Tables

An extended method environment $\eta^{\cdot}$ ***satisfies*** $ST$, written $\eta^{\cdot} \models ST$, if and only if for all ref types $T$ and $m \in \mathit{Meths}\ T$,

$$\eta^{\cdot}(T,m) \models ST(T,m).$$

An normal method environment $\eta$ ***satisfies*** $ST$, written $\eta \models ST$, if and only if for all classes $K$ and $m \in \mathit{Meths}\ K$,

$$\eta(K,m) \models ST(K,m).$$

# Modular Correctness

For command $\Gamma \vdash C$ and $\Gamma$-specification *spec*, $C$ **modularly satisfies** *spec* **with respect to** $ST$, written $ST, (\Gamma \vdash C) \models^{\mathcal{D}} spec$ if and only if

$$\forall \eta \in MethEnv \cdot \eta \models ST \Rightarrow \mathcal{D}[[\Gamma \vdash C]](\eta) \models spec.$$

For command $\Gamma \vdash C$ and $\Gamma$-specification *spec*, $C$ **modularly satisfies** *spec* **with respect to** $ST$ **under static dispatch**, *written* $ST, (\Gamma \vdash C) \models^{\mathcal{S}} spec$ if and only if

$$\forall \eta' \in XMethEnv \cdot \eta' \models ST \Rightarrow \mathcal{S}[[\Gamma \vdash C]](\eta') \models spec.$$

# Supertype Abstraction (1)

A specification table *ST* *allows supertype abstraction* when
$ST, (\Gamma \vdash C) \models^{\mathcal{S}} spec$ implies $ST, (\Gamma \vdash C) \models^{\mathcal{D}} spec$
and similarly for expressions.

However, we don't want to reason about all method environments as in the definitions of satisfaction!

# Predicate Transformers to the Rescue

- A proof system would use axiomatic semantics
- Method m in type T would be dealt with as:

  **assert** $pre^{T}_{m}$;
  **assume** $post^{T}_{m}$;

  which acts as a predicate transformer.
- Notation:
  - $\{[spec]\}$ is the predicate transformer for *spec*.
  - $\{[ST]\}$ is the extended method environment composed of such transformers
    = least refined environment that satisfies *ST*.
  - $\mathbb{S}\{[\Gamma \vdash C]\}(\{[ST]\})$ is the predicate transformer denoted by $C$ in $\{[ST]\}$.

# Modular Verification (2)

For command $\Gamma \vdash C$ and $\Gamma$-specification *spec*, $C$ is ***modularly verified for*** *spec* ***with respect to*** *ST*, if and only if

$$\mathbb{S}\{[\Gamma \vdash C]\}(\{[ST]\}) \sqsupseteq \{[spec]\}.$$

# Supertype Abstraction

**Modular verification implies modular correctness** when:

$$\mathbb{S}\{[\Gamma \vdash C]\}\{[ST]\}) \sqsupseteq \{[spec]\}$$

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

and similarly for expressions.

# Main Results

The following are equivalent:

(a) $ST$ has behavioral subtyping.

(b) Modular correctness under static dispatch implies modular correctness.

(c) Modular verification implies modular correctness.

$$
\begin{tikzcd}
\mathcal{S}[\![\mathcal{P}]\!](\{\![ST]\!\}) \sqsupseteq \{\![spec]\!\} \arrow[rr, Rightarrow, "Thm.\ 8.8(a)"] \arrow[d, Rightarrow, "Thm.\ 8.10^{+}"'] \arrow[drr, Rightarrow, "Def.\ 8.6^{+}"] & & ST, \mathcal{P} \models^{\mathcal{S}} spec \arrow[d, Rightarrow, "Def.\ 8.7^{+}"] \\
\mathcal{D}[\![\mathcal{P}]\!](\{\![ST]\!\}) \sqsupseteq \{\![spec]\!\} \arrow[rr, Rightarrow, "Thm.\ 8.8(b)"] & & ST, \mathcal{P} \models^{\mathcal{D}} spec
\end{tikzcd}
$$

# Related Work

- Work with Naumann [LN06][LN15], basis for this talk. Proved exact conditions on behavioral subtyping for validity of supertype abstraction
- Liskov and Wing [LW94] "subtype requirement" like supertype abstraction.
  Abstraction functions implicit in JML.
- Several program logics for Java, [Mül02] [Par05] [Pie06] [PHM99], use supertype abstraction.
- America [Ame87] [Ame91] first proved soundness with behavioral subtyping.

# Conclusions

- Supertype abstraction defined semantically, based on modular reasoning.
- Supertype abstraction is valid if:
  - invariant methodology enforced, and
  - subtypes are behavioral subtypes.

Plus: a story about specification inheritance.

# References


[Ame87] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, NY, June 1987. Springer-Verlag. Lecture Notes in Computer Science, volume 276.

[Ame91] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, NY, 1991.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is ISU CS TR #95-20c, rlhttp://tinyurl.com/s2krg.

[FF01] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *OOPSLA '01 Conference Proceedings, Object-Oriented Programming, Systems, Languages, and Applications, October 14-18, 2001, Tampa Bay, Florida, USA*, pages 1–15, October 2001.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[LD00] Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[Lei98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM, October 1998.

[LN06] Gary T. Leavens and David A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20b, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, September 2006.

[LN15] Gary T. Leavens and David A. Naumann. Behavioral Subtyping, Specification Inheritance, and Modular Reasoning. ACM TOPLAS, 37(4):13:1-13:88, Aug. 2015. http://doi.acm.org/10.1145/2766446.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[MPHL06] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253– 286, October 2006.

[Mül02] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[Par05] Matthew J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, November 2005. The author's Ph.D. dissertation.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

[Pie06] Cees Pierik. *Validation Techniques for Object-Oriented Proof Outlines*. PhD thesis, Universiteit Utrecht, 2006.

[SBC92] Susan Stepney, Rosalind Barden, and David Cooper, editors. *Object Orientation in Z*. Workshops in Computing. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[Wil92] Alan Wills. Specification in Fresco. In Stepney et al. [SBC92], chapter 11, pages 127–135.

[Win83] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.