Verification of Object-oriented Programs
Lecture 4: subtyping and model programs

Dave Naumann

Department of Computer Science
Stevens Institute of Technology

Ph.D. Fall School on Logics and Semantics of State
IT University, Copenhagen, October 2008

Partially supported by US NSF grants CNS-0627338 and CRI-0708330.
Outline of lecture

Behavioral subtyping

Mandatory calls

Summary
class Cell { v: int; meth set(x:int) ensures v = x; }
class BCell extends Cell {
  b: int;
  meth set(x:int) ensures v = x ∧ b = old(v); }

Well known, formalized in several OO verification logics and tools [Parkinson, Pierik, Oheimb/Nipkow, Müller, ...]

Logic-neutral account [Liskov, Wing] for impoverished programming model.

Our goal: semantical analysis of behavioral subtyping, for imperative OO programs and pre/post specifications.
class Cell { v: int; meth set(x:int) ensures v = x; }
class BCell extends Cell {
    b: int;
    meth set(x:int) ensures v = x ∧ b = old(v); }
class Client {
    c: Cell := ... ; c.set(2); assert c.v = 2; ...

\[ ST(\text{Cell, } \text{set}(x)) = (\text{true}, \text{self}.v = x) \]  (Specification Table)

\textit{Supertype abstraction:} Reasoning about c.set(2) in terms of static type. Because:
For any subtype \( S \leq \text{Cell} \), require \( ST(S, \text{set}) \sqsupseteq ST(\text{Cell, set}) \).

Intrinsic refinement order \( \sqsupseteq \) means that any implementation that satisfies \( ST(S, \text{set}) \) also satisfies \( ST(\text{Cell, set}) \).
Behavioral subtyping

For all types $S, T$, if $S \leq T$ then $(pre^S, post^S) \sqsubseteq (pre^T, post^T)$ which by definition means:

$C \models (pre^S, post^S)$ implies $C \models (pre^T, post^T)$ for all commands $C$.

Prove $(pre^S, post^S) \sqsubseteq (pre^T, post^T)$ by inferences like:

\[
\frac{pre^T \Rightarrow pre^S \quad post^S \Rightarrow post^T}{(pre^S, post^S) \sqsubseteq (pre^T, post^T)}
\]

Recall that $Val(S, r) \subseteq Val(T, r)$ (for any ref context $r$). As sets, $pre^T \subseteq pre^S$ could fail owing to type of self.
Behavioral subtyping

For all types $S$, $T$, if $S \leq T$ then $(\text{pre}^S, \text{post}^S) \sqsupseteq (\text{pre}^T, \text{post}^T)$ which by definition means:

$C \models (\text{pre}^S, \text{post}^S)$ implies $C \models (\text{pre}^T, \text{post}^T)$ for all commands $C$.

Prove $(\text{pre}^S, \text{post}^S) \sqsupseteq (\text{pre}^T, \text{post}^T)$ by inferences like:

$$
\frac{\text{pre}^T \Rightarrow \text{pre}^S \quad \text{post}^S \Rightarrow \text{post}^T}{(\text{pre}^S, \text{post}^S) \sqsupseteq (\text{pre}^T, \text{post}^T)}
$$

Recall that $\text{Val}(S, r) \subseteq \text{Val}(T, r)$ (for any ref context $r$).
As sets, $\text{pre}^T \subseteq \text{pre}^S$ could fail owing to type of self.
Behavioral subtyping

For all types $S$, $T$, if $S \leq T$ then $(\text{pre}^S, \text{post}^S) \sqsupseteq (\text{pre}^T, \text{post}^T)$ which by definition means:

$C \models (\text{pre}^S, \text{post}^S)$ implies $C \models (\text{pre}^T, \text{post}^T)$ for all commands $C$.

Prove $(\text{pre}^S, \text{post}^S) \sqsupseteq (\text{pre}^T, \text{post}^T)$ by inferences like:

\[
\frac{\text{pre}^T \Rightarrow \text{pre}^S \quad \text{post}^S \Rightarrow \text{post}^T}{(\text{pre}^S, \text{post}^S) \sqsupseteq (\text{pre}^T, \text{post}^T)}
\]

Recall that $\text{Val}(S, r) \subseteq \text{Val}(T, r)$ (for any ref context $r$).
As sets, $\text{pre}^T \subseteq \text{pre}^S$ could fail owing to type of self.
Characterizing refinement

How to reduce $\sqsubseteq$ to conditions on specifications? Take types into account, and in addition...

Example: auxiliary $a : \text{int}$, prog vars $x : \text{int}, r : \text{real}$

$\text{pre0} \triangleq a \leq r \leq a + 1$ and $\text{post0} \triangleq x = a \lor x = a + 1$

Observe that $(\text{pre0}, \text{post0}) \sqsubseteq (r = 0, x = 0)$

but we have neither $r = 0 \Rightarrow \text{pre0}$ nor $\text{post0} \Rightarrow x = 0$.

Note: If $C$ satisfies $(\text{pre0}, \text{post0})$ then by substitutions we get

$\{ -1 \leq r \leq 0 \} C \{ x = -1 \lor x = 0 \}$ and

$\{ 0 \leq r \leq 1 \} C \{ x = 0 \lor x = 1 \}$, so by rule of conjunction we get

$\{ r = 0 \} C \{ x = 0 \}$.

Adaptation rules are about weakest $\text{pre}$ such that

$(\text{pre0}, \text{post0}) \sqsubseteq (\text{pre}, \text{post})$. 
Characterizing refinement

How to reduce $\sqsubseteq$ to conditions on specifications? Take types into account, and in addition...

Example: auxiliary $a : \text{int}$, prog vars $x : \text{int}, r : \text{real}$

$pre0 \triangleq a \leq r \leq a + 1$ and $post0 \triangleq x = a \lor x = a + 1$

Observe that $(pre0, post0) \sqsubseteq (r = 0, x = 0)$
but we have neither $r = 0 \Rightarrow pre0$ nor $post0 \Rightarrow x = 0$.

Note: If $C$ satisfies $(pre0, post0)$ then by substitutions we get

$\{-1 \leq r \leq 0\} C \{x = -1 \lor x = 0\}$ and

$\{0 \leq r \leq 1\} C \{x = 0 \lor x = 1\}$, so by rule of conjunction we get

$\{r = 0\} C \{x = 0\}$.

*Adaptation rules* are about weakest $pre$ such that
$(pre0, post0) \sqsubseteq (pre, post)$. 

Lecture 4
Characterizing refinement

How to reduce ⊑ to conditions on specifications? Take types into account, and in addition...

Example: auxiliary $a : \text{int}$, prog vars $x : \text{int}$, $r : \text{real}$

$pre0 \triangleq a \leq r \leq a + 1$ and $post0 \triangleq x = a \lor x = a + 1$

Observe that $(pre0, post0) \sqsubseteq (r = 0, x = 0)$

but we have neither $r = 0 \Rightarrow pre0$ nor $post0 \Rightarrow x = 0$.

Note: If $C$ satisfies $(pre0, post0)$ then by substitutions we get

$\{ -1 \leq r \leq 0 \} C \{ x = -1 \lor x = 0 \}$ and

$\{ 0 \leq r \leq 1 \} C \{ x = 0 \lor x = 1 \}$, so by rule of conjunction we get

$\{ r = 0 \} C \{ x = 0 \}$.

Adaptation rules are about weakest $pre$ such that

$(pre0, post0) \sqsubseteq (pre, post)$. 
Characterizing refinement

How to reduce $\sqsupseteq$ to conditions on specifications? Take types into account, and in addition...

Example: auxiliary $a : \text{int}$, prog vars $x : \text{int}$, $r : \text{real}$

$pre_0 \triangleq a \leq r \leq a + 1$ and $post_0 \triangleq x = a \lor x = a + 1$

Observe that $(pre_0, post_0) \sqsupseteq (r = 0, x = 0)$
but we have neither $r = 0 \Rightarrow pre_0$ nor $post_0 \Rightarrow x = 0$.

Note: If $C$ satisfies $(pre_0, post_0)$ then by substitutions we get

$\{ -1 \leq r \leq 0 \} \ C \ \{ x = -1 \lor x = 0 \}$ and
$\{ 0 \leq r \leq 1 \} \ C \ \{ x = 0 \lor x = 1 \}$, so by rule of conjunction we get

$\{ r = 0 \} \ C \ \{ x = 0 \}$.

Adaptation rules are about weakest $pre$ such that $(pre_0, post_0) \sqsupset (pre, post)$. 
Modular reasoning from behavioral subtyping

Let $ST$ have the semantic, ramified specifications.
Write $D[C](\eta)$ for dynamic dispatch semantics (method envt. $\eta$).

**Modular correctness** of $C$ w.r.t. specification $spec$ and $ST$

$\Delta \forall \eta | \eta \vdash ST \Rightarrow D[C](\eta) \models spec$

Conjecture: If $ST$ has behavioral subtyping, then modular correctness follows from static reasoning.

Write $S[C](\eta)$ for static dispatch semantics.

Conjecture made precise: modular correctness follows from

$\forall \eta | \eta \vdash ST \Rightarrow S[C](\eta) \models spec$
Modular reasoning from behavioral subtyping

Let $ST$ have the semantic, ramified specifications. Write $D[C](\eta)$ for dynamic dispatch semantics (method envt. $\eta$).

**Modular correctness of** $C$ w.r.t. specification $spec$ and $ST$

$\Delta \forall \eta \mid \eta \models ST \Rightarrow D[C](\eta) \models spec$

Conjecture: If $ST$ has behavioral subtyping, then modular correctness follows from static reasoning.

Write $S[C](\eta)$ for static dispatch semantics.

Conjecture made precise: modular correctness follows from$\forall \eta \mid \eta \models ST \Rightarrow S[C](\eta) \models spec$
Semantics of commands and correctness (lecture 0)

Denotational semantics: State $\sigma$ has form $(r, h, s)$ where $r$ maps allocated refs to class names; $h \in Heap(r)$ and $s \in Store(r)$.

$D[\Gamma \vdash C](\eta)$ is total function sending each $\Gamma$-state $\sigma$ to $\bot$ (fault), $\bot$ (divergence), or a $\Gamma$-state $\tau$.

Faults: null dereference. Typing implies no dangling refs.

Formulas denote sets of states.

\[\eta \models \{ P \} \ C \ \{ Q \} \ [\varepsilon] \triangleq \text{for all } \sigma \in [P] \text{ we have}
\]
- $D[C](\eta)(\sigma) \neq \bot$;
- if $D[C] \eta(\sigma) \neq \bot$ and $D[C](\sigma) = \tau$ then $(\sigma, \tau) \in [Q]$ and $\varepsilon$ allows transition from $\sigma$ to $\tau$. 
Counter-example

class K0 {
    meth m(y: Integer)
        requires y.val = 0
        ensures y.val = 1 \lor y.val = -1 \text{ effects wr y.val } } }

\( C0 \triangleq k,l: K0; \ y,z: \text{ int}; \ k.m(y); \ l.m(z); \)

\( spec0 \triangleq \text{ requires } k,l \text{ non-null } \land y.val = 0 = z.val \)

ensures y.val = z.val

Does \( C0 \) satisfy \( spec0 \)?

Strongest provable postcondition for \( y, z \) might be

\( (y = 1 \lor y = -1) \land (z = 1 \lor z = 1) \).

And in VC semantics with assume/assert for call, the two calls are nondeterministic and have possibly different results.

But \( C0 \) does satisfy \( spec0 \) in our semantics, because we quantify over deterministic implementations.
Counter-example

```java
class K0 {
    meth m(y: Integer)
        requires y.val = 0
        ensures y.val = 1 ∨ y.val = -1 effects wr y.val } }

C0 ⊢ k,l: K0; y,z: int; k.m(y); l.m(z);
spec0 ⊢ requires k,l non-null ∧ y.val = 0 = z.val
    ensures y.val = z.val

Does C0 satisfy spec0?

Strongest provable postcondition for y, z might be
(y = 1 ∨ y = −1) ∧ (z = 1 ∨ z − 1).

And in VC semantics with assume/assert for call, the two calls are nondeterministic and have possibly different results.

But C0 does satisfy spec0 in our semantics, because we quantify over deterministic implementations.
Counter-example

class K0 {
    meth m(y: Integer)
        requires y.val = 0
        ensures y.val = 1 ∨ y.val = -1 effects wr y.val
    }

C0 ⊢ k,l: K0; y,z: int; k.m(y); l.m(z);
spec0 ⊢ requires k,l non-null ∧ y.val = 0 = z.val
        ensures y.val = z.val

Does C0 satisfy spec0?
Strongest provable postcondition for y, z might be
(y = 1 ∨ y = -1) ∧ (z = 1 ∨ z - 1).
And in VC semantics with assume/assert for call, the two calls
are nondeterministic and have possibly different results.

But C0 does satisfy spec0 in our semantics, because we quantify
over deterministic implementations.
class K0 {
  meth m(y: Integer)
      requires y.val = 0
      ensures y.val = 1 ∨ y.val = -1 effects wr y.val }
}
class K1 extends K0 {
  //For m, inherits spec, overrides or inherits implementation
  C0  ∆ k,l: K0; y,z: int; k.m(y); l.m(z)
spec0  ∆ requires k,l all non-null ∧ y.val = 0 = z.val
      ensures y.val = z.val

We have ∀η | η ⊨ ST ⇒ S[C0](η) ⊨ spec0 because if η satisfies
ST then η(K0, m) sets val to 1 or to −1.

But not ∀η | η ⊨ ST ⇒ D[C0](η) ⊨ spec0.
E.g., let η0(K0, m) set val to 1 but η0(K1, m) sets val to −1.
Initially k is K0 but l is K1.
class K0 {
    meth m(y: Integer)
        requires y.val = 0
        ensures y.val = 1 \lor y.val = -1 \enskip \text{effects wr } y.val \} }
class K1 extends K0 {
    //For m, inherits spec, overrides or inherits implementation
C0 \triangleq k,l: K0; y,z: int; k.m(y); l.m(z)
spec0 \triangleq requires k,l all non-null \land y.val = 0 = z.val
    ensures y.val = z.val

We have \forall \eta \mid \eta \models ST \Rightarrow S[C0](\eta) \models spec0 \text{ because if } \eta \text{ satisfies } ST \text{ then } \eta(K0, m) \text{ sets val to } 1 \text{ or to } -1.

But not \forall \eta \mid \eta \models ST \Rightarrow D[C0](\eta) \models spec0.
E.g., let \eta0(K0, m) set val to 1 but \eta0(K1, m) sets val to -1. Initially \( k \) is \( K0 \) but \( l \) is \( K1 \).
Define predicate transformer semantics, dynamic $D[[C]](\eta)$ and static $S[[C]](\eta)$ dispatch.

Def $\{\text{spec}\}$ as a predicate transformer, so that for any spec and any state transformer $\psi$ we have $\psi \Vdash \text{spec}$ iff $wp(\psi) \sqsupseteq \{\text{spec}\}$

Def $\{ST\} \triangleq$ the envt. that maps each $T, m$ to $[[ST(T, m)]]$.

Thm: If $ST$ is satisfiable and has behavioral subtyping then for any $\Gamma \vdash C$ we have $D[[\Gamma \vdash C]]([[ST]]) \sqsubseteq S[[\Gamma \vdash C]]([[ST]])$. 
Define predicate transformer semantics, dynamic $\mathcal{D}([C])(\eta)$ and static $\mathcal{S}([C])(\eta)$ dispatch.

Def $[spec]$ as a predicate transformer, so that for any $spec$ and any state transformer $\psi$ we have $\psi \models spec$ iff $wp(\psi) \supseteq [spec]$.

Def $[ST] \triangle$ the envt. that maps each $T, m$ to $[ST(T, m)]$.

Thm: If $ST$ is satisfiable and has behavioral subtyping then for any $\Gamma \vdash C$ we have $\mathcal{D}([\Gamma \vdash C])([ST]) \supseteq \mathcal{S}([\Gamma \vdash C])([ST])$. 
Supertype abstraction and behavioral subtyping

Def \textit{Supertype abstraction is valid for } ST \triangle \textit{ implies } S\{C\}\{ST\} \sqsubseteq \{spec\} \textit{ for all } C, spec.

Thm: For satisfiable \textit{ST}, the following are equivalent:

- \textit{ST} has behavioral subtyping
- supertype abstraction is valid for \textit{ST}
Supertype abstraction and behavioral subtyping

Def **Supertype abstraction is valid for** $ST \triangleleft$

$S\{C\}\{ST\} \supseteq \{spec\}$ implies $\forall \eta \mid \eta \models ST \Rightarrow D[C](\eta) \models spec$

for all $C, spec$.

Thm: For satisfiable $ST$, the following are equivalent:

- $ST$ has behavioral subtyping
- supertype abstraction is valid for $ST$
(Quasi) higher order methods

update methods in the Observer pattern make "mandatory calls" to notify in observers.

Template Method pattern: mandatory calls to several abstract "primitive operations"

HandleRequest in Chain of Responsibility pattern: if cannot directly handle a request, call the next.

Client calls into pattern: Interpret method in the Interpreter pattern, Execute method in the Command pattern, Accept method in the Visitor pattern, strategy method in a Strategy object.

"higher order method" (HOM): one specified to make mandatory calls under some conditions.
(Quasi) higher order methods

update methods in the Observer pattern make “mandatory calls” to notify in observers.

Template Method pattern: mandatory calls to several abstract “primitive operations”

HandleRequest in Chain of Responsibility pattern: if cannot directly handle a request, call the next.

Client calls into pattern: Interpret method in the Interpreter pattern, Execute method in the Command pattern, Accept method in the Visitor pattern, strategy method in a Strategy object.

“higher order method” (HOM): one specified to make mandatory calls under some conditions.
(Quasi) higher order methods

update methods in the Observer pattern make “mandatory calls” to notify in observers.

Template Method pattern: mandatory calls to several abstract “primitive operations”

HandleRequest in Chain of Responsibility pattern: if cannot directly handle a request, call the next.

Client calls into pattern: Interpret method in the Interpreter pattern, Execute method in the Command pattern, Accept method in the Visitor pattern, strategy method in a Strategy object.

“higher order method” (HOM): one specified to make mandatory calls under some conditions.
Weak specs for mandatory calls

```java
class Counter {
    private count: int := 0;
    private lstnr: Listener;
    
    meth register(Listener lnr)
        effects wr lstnr;  // (spec public)
        ensures lstnr = lnr;
    {
        this.lstnr := lnr;
    }

    meth bump() {
        count := count+1;
        if lstnr != null then
            lstnr.notify(count);
    }
}

interface Listener {
    
    meth notify(x: int)
        effects wr self.objState;
        ensures true;

    Spec of notify is weak.

    (The data group
    objState abstracts
    from concrete fields
    declared to be “in” the
    group.)
```
Weak specs for mandatory calls

class Counter {
    private count: int := 0;
    private Lstnr: Listener;
    meth register(Listener Lnr)
        effects wr Lstnr; // (spec public)
        ensures Lstnr = Lnr;
    { this.Lstnr := Lnr; }

    meth bump() {
        count := count + 1;
        if Lstnr != null then
            Lstnr.notify(count);
    } }

interface Listener {
    meth notify(x: int)
        effects wr self.objState;
        ensures true;

    Spec of notify is weak.

    (The data group
    objState abstracts
    from concrete fields
    declared to be “in” the
group.)
Weak specs for mandatory calls

class Counter {
    private count: int := 0;
    private lstnr: Listener;
    meth register(Listener ln)
        effects wr lstnr;  // (spec public)
        ensures lstnr = ln;
    { this.lstnr := ln; }

    meth bump() {
        count := count+1;
        if lstnr != null then
            lstnr.notify(count);
    } } } 

interface Listener {
    meth notify(x: int)
        effects wr self.objState;
        ensures true;

    Spec of notify is weak.

    (The data group
    objState abstracts
    from concrete fields
    declared to be “in” the
group.)
Client reasoning example

class Counter {
    private count: int := 0;
    private lsnr: Listener;
    meth register(Listener lnr)
        effects wr lsnr;
        ensures lsnr = lnr; ...
    meth bump() {
        count := count+1;
        if lsnr != null then
            lsnr.notify(count); }}

interface Listener {
    meth notify(x: int)
        effects wr self.objState;
        ensures true;
}

class LastVal implements Listener {
    private val: int := 0 in objState
    meth notify(x: int) effects wr objState;
        ensures val = x;
    ...}

lv: LastVal := new LastVal();
assert lv != null \lv.val = 0;
c: Counter := new Counter();
c.register(lv);
assert c.lsnr=lv \lv != null\ c.count=0;
c.bump();
assert lv.val = 1;
Client reasoning example

class Counter {
    private count: int := 0;
    private lstnr: Listener;
    meth register(Listener lnr)
        effects wr lstnr;
        ensures lstnr = lnr; ...
    meth bump() {
        count := count + 1;
        if lstnr != null then
            lstnr.notify(count); }}
}

interface Listener {
    meth notify(x: int)
        effects wr self.objState;
        ensures true;
}

class LastVal implements Listener {
    private val: int := 0 in objState
    meth notify(x: int) effects wr objState;
        ensures val = x;
    ...
}

lv: LastVal := new LastVal();
assert lv != null ∧ lv.val = 0;
c: Counter := new Counter();
c.register(lv);
assert c.lstnr = lv ∧ lv != null ∧ c.count = 0;
c.bump();
assert lv.val = 1;
Client reasoning possibilities

class Counter {
    private count: int:= 0; ... 
meth bump() { spec??
        count := count+1;
        if lsnr != null then
            lsnr.notify(count); }}

H.O.: bump does whatever the notify methods of its registered listeners do.
Traces: it calls lsnr.notify.

class LastVal implements Listener {
    private val: int:= 0 in objState 
meth notify(x: int) effects wr objState;
    ensures val = x;
}

lv: LastVal := new LastVal();
assert lv != null ∧ lv.val = 0;
c: Counter := new Counter();
c.register(lv);
assert c.lsnr=lv ∧ lv != null∧ c.count=0;
c.bump();
assert lv.val = 1;
Client reasoning possibilities

class Counter {
    private count: int := 0; ...
    meth bump() { spec??
        count := count+1;
        if lsnr != null then
            lsnr.notify(count);
    }
}

H.O.: bump does whatever the notify methods of its registered listeners do.

Traces: it calls lsnr.notify.

class LastVal implements Listener {
    private val: int := 0 in objState
    meth notify(x: int) effects wr objState;
        ensures val = x;
    }

lv: LastVal := new LastVal();
assert lv != null ∧ lv.val = 0;
c: Counter := new Counter();
c.register(lv);
assert c.lsnr=lv ∧ lv != null∧ c.count=0;
c.bump();
assert lv.val = 1;
Model program as specification

```plaintext
meth bump();
    modelProg {
        spec ensures count = old(count+1) effects wr count; end
        if lstnr != null then
            lstnr.notify(count); }
    { count := count+1;
        if lstnr != null then
            lstnr.notify(count); }

Model program: code, that must be matched by any implementation. Allowed to contain specification statements, which abstract from subprograms.

spec requires P ensures Q effects ε end
```
Model program as specification

meth bump();
modelProg {
    spec ensures count = old(count+1) effects wr count; end
    if lsnr != null then
        lsnr.notify(count); }
{ count := count+1;
    if lsnr != null then
        lsnr.notify(count); }

Model program: code, that must be matched by any implementation. Allowed to contain specification statements, which abstract from subprograms.

spec requires P ensures Q effects e end
Client reasoning by copy rule

\[
\text{l}: \text{LastVal} := \text{new LastVal}();
\]
\[
\text{assert } \text{l} \neq \text{null } \land \text{l}.\text{val} = 0;
\]
\[
\text{c}: \text{Counter} := \text{new Counter}();
\]
\[
\text{c}.\text{register}(\text{l});
\]
\[
\text{assert } \text{c}.\text{lstnr=I} \land \text{l} \neq \text{null } \land \text{c}.\text{count}=0;
\]
\[
\text{spec ensures } \text{c}.\text{count} = \text{old}(\text{c}.\text{count}+1) \text{ effects wr c}.\text{count}; \text{ end} \quad \text{// bump}
\]
\[
\text{if } \text{c}.\text{lstnr} \neq \text{null} \text{ then } \text{c}.\text{lstnr}.\text{notify}(\text{c}.\text{count}); \quad \text{ // inlined}
\]
\[
\text{assert } \text{l}.\text{val} = 1;
\]

Provable using specifications.

```java
class LastVal implements Listener {
    
    @ Meth notify(x: int)
    
    ensures val = x;
}
```
Verifying implementations

If $SpecTable(T, m)$ is $modelProg(C_{mod})$ then an implementation $C_{imp}$ must refine $C$.

To avoid refinement calculus, use syntax refining $P, Q$ as $C$ to mark subprograms of $C_{imp}$ intended to match specification statements in $C_{mod}$.

\[
SpecTable(T, m) = modelProg(C_{mod}) \\
matches(C_{imp}, C_{mod}) \\
\vdash \{P\}C\{Q\} \text{ for all (refining } P, Q \text{ as } C \text{) in } C_{imp} \\
\]  

$C_{imp} sat SpecTable(T, m)$

Note that $C_{imp}$ has no spec statements. But we need them for following.
Verifying implementations

If \( \text{SpecTable}(T, m) \) is \( \text{modelProg}(C_{mod}) \) then an implementation \( C_{imp} \) must refine \( C \).

To avoid refinement calculus, use syntax \textit{refining} \( P, Q \) as \( C \) to mark subprograms of \( C_{imp} \) intended to match specification statements in \( C_{mod} \).

\[
\begin{align*}
\text{SpecTable}(T, m) &= \text{modelProg}(C_{mod}) \\
&\text{matches}(C_{imp}, C_{mod}) \\
\vdash \{P\}C\{Q\} \text{ for all (refining } P, Q \text{ as } C \text{) in } C_{imp} \\
\end{align*}
\]

\( C_{imp} \text{ satSpecTable}(T, m) \)

Note that \( C_{imp} \) has no spec statements. But we need them for following.
Verifying clients

Rule for spec statements: \( \vdash \{ P \} \text{spec}(P, Q) \{ Q \} \)

Used to prove antecedent of copy rule:

\[
\begin{align*}
\text{SpecTable}(T', m) &= \text{modelProg}(C_{mod}) \\
\text{mtype}(T', m) &= \overline{z} : \overline{T} \rightarrow \text{void} \\
T' \leq T &\quad C = C_{mod} / \text{self}, \overline{z} \rightarrow x, \overline{y} \\
\Gamma, x : T' \vdash \{ P \} &\quad C \{ Q \} \\
\Gamma, x : T \vdash \{ P \land x \text{ is } T' \} &\quad x.(\overline{y}) \{ Q \}
\end{align*}
\]

Well formed: \( \text{self} : T', \overline{z} : \overline{T} \vdash C_{mod} \) and does not assign to \( \overline{z} \).
Extracting model programs

Rather than copy parts of the implementation, introduce syntax to designate that the implementation serves also as a model program, with parts abstracted by designated specification statements. (See the paper.)
Course summary

- Modular reasoning requires framing of local conditions and of invariants.
- Reentrancy and sharing can break encapsulation and framing. Tame them by reasoning about state dependent encapsulation boundaries reified using (mutable) ghost state.
- Ownership and related disciplines have been explored extensively using FOL/SMT solvers. Connections with separation logic? (e.g., mixed embedding)
- Dynamic framing: use ghost regions in effect specification.
- Region logic supports a second order frame rule. Goal: way to deploy reasoning disciplines on per-module basis, without hard-coding in proof system or verifier.
- Behavioral subtyping (refinement) iff supertype abstraction.
- First order reasoning about higher order methods via “Reference implementations”.
References for this lecture

[Liskov,Wing’94] A behavioral notion of subtyping

[Leavens,Naumann TR ’06] Behavioral subtyping is equivalent to modular reasoning for object-oriented programs (corrected version to appear)

[Büchi,Weck] The greybox approach: when blackbox specifications hide too much

[Shaner,Leavens,Naumann’07] Modular verification of higher-order methods with mandatory calls specified by model programs

[O’Hearn,Yang,Reynolds’04,’08] Separation and information hiding (using “specification statements” in environment)