Verification of Object-oriented Programs
Lecture 1: invariants, reentrancy, ownership

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

Reentrancy and object invariants

Ownership

Peer dependency
Previous lecture: framing

Rule of "invariance":

\[
\begin{align*}
\{ P \} C \{ Q \}[\varepsilon] \\
R \text{ independent from } \varepsilon \\
\{ P \land R \} C \{ Q \land R \}[\varepsilon]
\end{align*}
\]

Effect specification \( \varepsilon \) records what is written by \( C \); which must not be read by \( R \).

Don’t want \( \varepsilon \) to expose state encapsulated within \( C \).

Do want to allow \( R \) expressed in terms of pure methods or other abstractions.

This lecture: object invariants
Previous lecture: framing

if x.query() then
    x.update();
assert x.query() ???

Rule of “invariance”:

\[
\begin{array}{c}
\{ P \} C \{ Q \} [\epsilon] \\
R \text{ independent from } \epsilon \\
\{ P \land R \} C \{ Q \land R \} [\epsilon]
\end{array}
\]

Effect specification \( \epsilon \) records what is written by \( C \); which must not be read by \( R \).

Don’t want \( \epsilon \) to expose state encapsulated within \( C \).

Do want to allow \( R \) expressed in terms of pure methods or other abstractions.

This lecture: object invariants
Classical invariants

“Representation invariant”: on encapsulated state, so clients cannot falsify. (cf. rule of invariance)

Module assumes Inv on entry, establishes it on exit.

Unlike ordinary pre/post, clients are not responsible for Inv.

Client steps

A module

Lower layers

Works well with layered abstractions.
Reentrant callbacks

Shared mutable state + (conceptually) higher order \( \Rightarrow \) reentrant callbacks.

When must \textit{Inv} be re-established? vs. assumed

Which outgoing calls from module go to “lower layers” and which potentially lead to reentrant callbacks?
Reentrant callback; faults for Portion p with \( p\.view\.z = p \)

class Portion {
  private x, y: int := 0, 1;
  view: View := ...;
  invar 0 \leq x < y (that is, 0 \leq \text{self.x} \text{ and } \text{self.x} < \text{self.y})
  meth m() { x := x + 1; view.notify(); y := y + 1; }
  meth f(): real { return 1/(y-x); }
}

class View {
  z: Subject := ...;
  meth notify() { ...z.f() ... }
}
Reentrancy complicates reasoning about model fields

class Portion {
    private x,y: int := 0,1;
    view: View :=...;
    model f: real constraint f = 1/(y-x);
    invar 0 ≤ x < y (that is, 0≤ self.x and self.x < self.y)
    meth m() { x:= x+1; view.notify(); y:= y+1; }
    meth n() requires f < 0.2 {...}
}

class View {
    z: Subject := ...;
    meth notify() { ...z.n();... }
}
When do object invariants hold?

- **Disallow reentrant callbacks** (e.g., global call-graph analysis plus alias analysis). Too restrictive, too global.
- Use “call in progress” flag in precondition. Allow re-entrant calls that don’t depend on invariant.
- Require invariant to hold before every outgoing call. Incompatible with layered abstraction. But ok sometimes, e.g.,
  \[
  x := x + 1; \quad y := y + 1; \quad \text{view.notify();}
  \]
- Include in preconditions, as opaque predicate (pure method, model field, $\exists$-predicate). Abstraction, not hiding. Flexible.
When do object invariants hold?

- **Disallow reentrant callbacks** (e.g., global call-graph analysis plus alias analysis). Too restrictive, too global.
- **Use “call in progress” flag in precondition.** Allow re-entrant calls that don’t depend on invariant.
- **Require invariant to hold before every outgoing call.** Incompatible with layered abstraction. But ok sometimes, e.g.,
  
  \[
  x := x + 1; \quad y := y + 1; \quad \text{view.notify();}
  \]
- **Include in preconditions, as opaque predicate (pure method, model field, \(\exists\)-predicate).** Abstraction, not hiding. Flexible.
When do object invariants hold?

- **Disallow reentrant callbacks** (e.g., global call-graph analysis plus alias analysis). Too restrictive, too global.

- **Use “call in progress” flag in precondition.** Allow re-entrant calls that don’t depend on invariant.

- **Require invariant to hold before every outgoing call.** Incompatible with layered abstraction. But ok sometimes, e.g.,

  \[ x := x + 1; \ y := y + 1; \ \text{view.notify();} \]

- **Include in preconditions, as opaque predicate (pure method, model field, \( \exists \)-predicate).** Abstraction, not hiding. Flexible.
When do object invariants hold?

- **Disallow reentrant callbacks** (e.g., global call-graph analysis plus alias analysis). Too restrictive, too global.

- **Use “call in progress” flag in precondition.** Allow re-entrant calls that don’t depend on invariant.

- **Require invariant to hold before every outgoing call.** Incompatible with layered abstraction. But ok sometimes, e.g.,
  
  \[ x := x + 1; \quad y := y + 1; \quad \text{view.notify();} \]

- **Include in preconditions, as opaque predicate** (pure method, model field, $\exists$-predicate). *Abstraction, not hiding.* Flexible.
Boogie methodology (inv bit)

*Ghost field:* assignable auxiliary (vs. defined model field).

Add ghost field `inv: bool` and maintain this hidden invariant in all states: \( \forall p \mid p.inv \Rightarrow I(p) \) where \( I \) is the object invariant.

- Pre- (and post-) condition of many methods; reentrant call won’t verify if cannot establish `inv = true`.
- Method implementation can use \( I(self) \) and even \( I(other) \)

“Expose” idiom (unpack/pack)

```plaintext
assume inv ∧ ... // method precondition
assert I(self) ∧ ... // from the all-states invariant
inv := false;
... updates that eventually re-establish ...
assert I(self)
inv := true;
```

Lecture 1
Boogie methodology (inv bit)

*Ghost field:* assignable auxiliary (vs. defined model field).

Add ghost field **inv**: bool and maintain this hidden invariant in all states: \( \forall p \mid p.inv \Rightarrow J(p) \) where \( J \) is the object invariant.

- Pre- (and post-) condition of many methods; reentrant call won’t verify if cannot establish inv = true.
- Method implementation can use \( J(self) \) and even \( J(other) \)

“Expose” idiom (unpack/pack)

 assume inv ∧… // method precondition
 assert \( J(self) \) ∧… // from the all-states invariant
 inv:= false;
 ...
 updates that eventually re-establish ...
 assert \( J(self) \)
 inv:= true;
Boogie methodology (inv bit)

*Ghost field:* assignable auxiliary (vs. defined model field).

Add ghost field `inv: bool` and maintain this hidden invariant in all states: \( \forall p \mid p.inv \Rightarrow I(p) \) where \( I \) is the object invariant.

- Pre- (and post-) condition of many methods; reentrant call won’t verify if cannot establish `inv = true`.
- Method implementation can use \( I(self) \) and even \( I(other) \)

“Expose” idiom (unpack/pack)

```plaintext
assume inv ∧... // method precondition
assert I(self) ∧... // from the all-states invariant
inv:= false;
... updates that eventually re-establish ...
assert I(self)
inv:= true;
```

What is ghost state?

Alert: inconsistent terminology in literature.

**Ghost:** Mutable auxiliary variables and object fields, added to program for specification and verification.

**Spec variables:** Not assigned in code; used to avoid 2-state post, e.g., \( \forall Y \{ x = Y \} \; x := x + 1 \{ x = Y + 1 \} [x] \)

Not in branch conditions; nor in assignments to program variables.

Suppose \( \{ P \} C \{ Q \} \ldots \) for \( P, Q \) that do not mention ghost state. Then \( \{ P \} erase(C) \{ Q \} \ldots \), where *erase* removes all assignments (and declarations) for ghosts.
State-dependent independence?

\[
\{ P \} \ C \ \{ Q \} \ [\varepsilon] \quad R \ \text{independent from} \ \varepsilon \\
\{ P \land R \} \ C \ \{ Q \land R \} \ [\varepsilon]
\]

assume \( \text{inv} \land (\text{inv} \Rightarrow \text{I}(\text{self})) \) // method precondition +
\( \text{inv} := \text{false}; \) // unpack
assert \( \neg \text{inv} \land (\text{inv} \Rightarrow \text{I}(\text{self})) \)
self.f := ...
assert \( \neg \text{inv} \land (\text{inv} \Rightarrow \text{I}(\text{self})) \)

Def \( R \triangleq (\text{inv} \Rightarrow \text{I}(\text{self})), \)
so \( R \) independent from \text{wr} \( f \) provided that \( P \Rightarrow \neg \text{inv} \).
Sharing can break encapsulation

Suppose class **IntCell** has mutable field **val**.

```java
class Portion {
    private x,y: IntCell := new IntCell(0),IntCell(1);
    view: View :=...;
    invar J: 0 ≤ x.val < y.val

    meth m() requires inv;
    { inv:= false; x.incr(); y.incr(); inv:= true; }

    meth leak() { return x; }
}
```

Client can use **leak** and falsify the global invariant
\[ \forall p : Portion \ | \ p.inv \Rightarrow J(p). \]
Sharing can break encapsulation

Suppose class \texttt{IntCell} has mutable field \texttt{val}.

class Portion {
    \texttt{private} x,y: IntCell := new IntCell(0),IntCell(1);
    \texttt{view}: View := . . . ;
    \texttt{invar} J: 0 \leq x.\texttt{val} < y.\texttt{val}
    \texttt{meth} m() \texttt{requires} inv;
    \{ \texttt{inv}:= false; x.\texttt{incr}(); y.\texttt{incr}(); \texttt{inv}:= true; \}
    \texttt{meth} leak() \{ \texttt{return} x; \}
}

Client can use \texttt{leak} and falsify the global invariant
\[ \forall p: Portion \mid p.\texttt{inv} \Rightarrow J(p). \]
Ownership

Hidden invariant should depend only on owned or representation objects.

- Or only on private fields of self [Liskov, Wing]
- Alias confinement: Ownership [Clarke] or Universe [Müller] types—static rules to ensure dominator property: every path to an owned object goes through its owner. Lightweight but no definitive solution. Ownership transfer breaks type invariant.
- Ownership hierarchy encoded in ghost state [Barnett et al: Boogie]. (Direct use, or to justify confinement regime.)
Ownership

Hidden invariant should depend only on **owned** or **representation** objects.

- Or only on private fields of **self** [Liskov, Wing]
- Alias **confinement**: Ownership [Clarke] or Universe [Müller] types—static rules to ensure **dominator property**: every path to an owned object goes through its owner.

Lightweight but no definitive solution.
Ownership transfer breaks type invariant.

- Ownership hierarchy encoded in ghost state [Barnett et al: Boogie]. (Direct use, or to justify confinement regime.)
Ownership

Hidden invariant should depend only on owned or representation objects.

- Or only on private fields of self [Liskov, Wing]
- Alias confinement: Ownership [Clarke] or Universe [Müller] types—static rules to ensure dominator property: every path to an owned object goes through its owner. Lightweight but no definitive solution. Ownership transfer breaks type invariant.
- Ownership hierarchy encoded in ghost state [Barnett et al: Boogie]. (Direct use, or to justify confinement regime.)
Ownership

Hidden invariant should depend only on owned or representation objects.

- Or only on private fields of self [Liskov, Wing]
- Alias confinement: Ownership [Clarke] or Universe [Müller] types—static rules to ensure dominator property: every path to an owned object goes through its owner. Lightweight but no definitive solution. Ownership transfer breaks type invariant.
- Ownership hierarchy encoded in ghost state [Barnett et al: Boogie]. (Direct use, or to justify confinement regime.)
Nested islands

Disallow or control usage of reference from client into encapsulated island.
Nested islands encoded using ghost pointers

Disallow or control usage of reference from client into encapsulated island.

Island reified by transitive ownership.

List sorted on content?
Nested islands encoded using ghost pointers

Disallow or control usage of reference from client into encapsulated island.

Island reified by transitive ownership.
List sorted on content?
Boogie methodology (inv/own discipline)

Mutable ownership hierarchy with nested unpack/pack.
Ghost fields `com : bool` and `own : Object`.

Stipulated precondition for any update `x.f := E` is `¬x.inv`
Stipulated precondition for `unpack o` is `o.inv ∧ ¬o.com`.

The global invariants:  
\[ ∀p \ | \ p.inv \Rightarrow I(p) \]
\[ ∀o, p \ | \ o.inv ∧ p.own = o \Rightarrow p.com \]
\[ ∀o \ | \ o.com \Rightarrow o.inv \]

\[ ¬p.inv \Rightarrow ¬o.inv \] in any state where `p` is transitively owned by `o`
Boogie methodology (inv/own discipline)

Mutable ownership hierarchy with nested unpack/pack.
Ghost fields \textit{com} : bool and \textit{own} : Object.

Stipulated precondition for any update \( x.f := E \) is \( \neg x.inv \)
Stipulated precondition for \texttt{unpack o} is \( o.inv \land \neg o.com \).

The global invariants:
\[ \forall p \; | \; p.inv \Rightarrow I(p) \]
\[ \forall o, p \; | \; o.inv \land p.own = o \Rightarrow p.com \]
\[ \forall o \; | \; o.com \Rightarrow o.inv \]

\( \neg p.inv \Rightarrow \neg o.inv \) in any state where \( p \) is transitively owned by \( o \).
Boogie methodology (inv/own discipline)

Mutable ownership hierarchy with nested unpack/pack.
Ghost fields \textit{com : bool} and \textit{own : Object}.

Stipulated precondition for any update $x.f := E$ is $\neg x.inv$
Stipulated precondition for $\text{unpack o}$ is $o.inv \land \neg o.com$.

The global invariants: 
\[
\forall p \mid p.inv \Rightarrow I(p) \\
\forall o, p \mid o.inv \land p.own = o \Rightarrow p.com \\
\forall o \mid o.com \Rightarrow o.inv
\]

\[\neg p.inv \Rightarrow \neg o.inv\] in any state where $p$ is transitively owned by $o$
Boogie methodology (inv/own discipline)

Mutable ownership hierarchy with nested unpack/pack.
Ghost fields \( \text{com} : \text{bool} \) and \( \text{own} : \text{Object} \).

Stipulated precondition for any update \( x.f := E \) is \( \neg x.inv \)
Stipulated precondition for \( \text{unpack o} \) is \( o.inv \land \neg o.com \).

![Diagram of unpack/pack and global invariants](image)

The global invariants:

\[
\forall p \mid p.inv \Rightarrow I(p) \\
\forall o, p \mid o.inv \land p.own = o \Rightarrow p.com \\
\forall o \mid o.com \Rightarrow o.inv
\]

\( \neg p.inv \Rightarrow \neg o.inv \) in any state where \( p \) is transitively owned by \( o \)
class Portion {
    private rep x,y: IntCell := new IntCell(0), IntCell(1);
    view: View := ... ;
    invar J: 0 \leq x.val < y.val \land x.own = self = y.own
    meth m() requires inv \land \neg com; {
        unpack self;
        x.incr(); y.incr();
        assert J \land x.inv \land y.inv \land \neg x.com \land \neg y.com;
        pack self;  }
}

unpack o requires o.inv \land \neg o.com, sets o.inv := false
    also sets p.com := false for all p with p.own = o.
pack o requires \neg o.inv \land J(o)
    also \forall p \: p.own = o \Rightarrow p.inv \land \neg p.com; reverses pack.
class Portion {
  private rep x,y: IntCell := new IntCell(0),IntCell(1);
  view: View := . . . ;
  invar J: 0 ≤ x.val < y.val ∧ x.own= self= y.own
  meth m() requires inv ∧ − com; {
    unpack self;
    x.incr(); y.incr();
    assert J ∧ x.inv ∧ y.inv ∧ −x.com ∧ −y.com;
    pack self;  } ...  }

unpack o requires o.inv ∧ −o.com, sets o.inv := false
also sets p.com := false for all p with p.own = o.
pack o requires −o.inv ∧ J(o)
also ∀p | p.own = o ⇒ p.inv ∧ −p.com; reverses pack.
class Portion {
  private rep x,y: IntCell := new IntCell(0),IntCell(1);
  view: View := ...;
  invar J: 0 ≤ x.val < y.val ∧ x.own = self = y.own

  meth m() requires inv ∧ ¬ com; {
    unpack self;
    x.incr(); y.incr();
    assert J ∧ x.inv ∧ y.inv ∧ ¬ x.com ∧ ¬ y.com;
    pack self; } ... }

unpack o requires o.inv ∧ ¬ o.com, sets o.inv := false
also sets p.com := false for all p with p.own = o.
pack o requires ¬ o.inv ∧ J(o)
also ∀ p | p.own = o ⇒ p.inv ∧ ¬ p.com; reverses pack.
class Portion {
    private rep x,y: IntCell := new IntCell(0),IntCell(1);
    view: View := ...;
    invar J: 0 ≤ x.val < y.val ∧ x.own= self= y.own

    meth m() requires inv ∧ ¬ com; {
        unpack self;
x.incr(); y.incr();
        assert J ∧ x.inv ∧ y.inv ∧ ¬x.com ∧ ¬y.com;
pack self;   } ...   }

unpack o requires o.inv ∧ ¬o.com, sets o.inv := false
also sets p.com := false for all p with p.own = o.
pack o requires ¬o.inv ∧ J(o)
also ∀p | p.own = o ⇒ p.inv ∧ ¬p.com; reverses pack.
Parameters

class Portion {
    private x, y: IntCell := new IntCell(0), IntCell(1);
    meth equals(Portion other): bool
        requires inv ∧ other.inv;
    { return x = other.x ∧ y = other.y; }
Soundness of Boogie methodology

Admissible invariants:
\( J(o) \) depends on \( p.f \) only if \( p = o \) or \( p.own = o \).

Alleged global invariants:
\[
\forall o \mid o.inv \Rightarrow J(o) \\
\forall o, p \mid o.inv \land p.own = o \Rightarrow p.com \\
\forall o \mid o.com \Rightarrow o.inv
\]

Stipulated preconditions:

assert \( \neg x.inv; x.f := E; \)  // for every field update
assert \( \neg x.inv \land (y = \text{null} \lor \neg y.inv); x.own := y; \)  // transfer
assert \( x.inv \land \neg x.com; \) unpack \( x; \)
assert \( \neg x.inv \land J(x) \land \forall p \mid p.own = x \Rightarrow \neg p.com \land p.inv; \) pack \( x; \)
also \( x \neq \text{null} \)
Soundness of Boogie methodology

Admissible invariants:
\( I(o) \) depends on \( p.f \) only if \( p = o \) or \( p.own = o \).

Alleged global invariants:
\[
\forall o \mid o.inv \Rightarrow I(o) \\
\forall o, p \mid o.inv \land p.own = o \Rightarrow p.com \\
\forall o \mid o.com \Rightarrow o.inv
\]

Stipulated preconditions:
assert \( \neg x.inv \); \( x.f := E \); \hfill // for every field update
assert \( \neg x.inv \land (y = \text{null} \lor \neg y.inv) \); \( x.own := y \); \hfill // transfer
assert \( x.inv \land \neg x.com \); unpack \( x \);
assert \( \neg x.inv \land I(x) \land \forall p \mid p.own = x \Rightarrow \neg p.com \land p.inv \); pack \( x \);
also \( x \neq \text{null} \)
Simple admissible invariants

*Rep fields:*  
\[ \text{class List} \{ \text{rep head: Node}; \ldots \} \]

Implicit invariant  
\[ \text{head} = \text{null} \lor \text{head.own} = \text{self} \]

*Peers:*  
\[ \text{class Node} \{ \text{item: T; peer nxt: Node}; \ldots \} \]

Implicit invariant  
\[ \text{nxt} = \text{null} \lor \text{nxt.own} = \text{self.own} \]

*Admissible:* \( J(\text{self}) \) can depend on

- \( \text{self.i} \) (any field \( i \))
- \( \text{self.f.i} \) for rep \( f \)
- \( \text{self.f.g \ldots h.i} \) for rep \( f \), rep/peer \( g \ldots h \)
Simple admissible invariants

Rep fields: class List { rep head: Node; ...
Implicit invariant head = null \lor head.own = self

Peers: class Node { item: T; peer nxt: Node; ... }
Implicit invariant nxt = null \lor nxt.own = self.own

Admissible: J(self) can depend on

• self.i (any field i)
• self.f.i for rep f
• self.f.g...h.i for rep f, rep/peer g...h
Simple admissible invariants

Rep fields:  class List { rep head: Node; ...
Implicit invariant  head = null ∨ head.own = self

Peers:  class Node { item: T; peer nxt: Node; ...
Implicit invariant  nxt = null ∨ nxt.own = self.own

Admissible:  J(self) can depend on
  • self.i (any field i)
  • self.f.i for rep f
  • self.f.g...h.i for rep f, rep/peer g...h
Ownership abstraction in effect specs

Client can do little with a committed object. We can weaken the semantics of effect specs to allow update of committed objects. Amounts to only relying on their invariants.

**Def**  \[ \models \{ P \} C \{ Q \} [\varepsilon] \quad \Delta \text{ for all } \sigma \in \llbracket P \rrbracket \text{ we have} \]

- \[ [C](\sigma) \neq \emptyset; \]
- \[ [C](\sigma) \neq \bot \text{ then } (\sigma, \tau) \in \llbracket Q \rrbracket \]
  where \( \tau = [C](\sigma) \); and then:
- \( \tau(x) = \sigma(x) \) for every \( x \in \text{dom}(\Gamma) \), unless allowed by \( \varepsilon \)
- \( \tau(p.f) = \sigma(p.f) \) for every allocated \( p \) and every field \( f \), unless \( \sigma(p.com) = \text{true} \) or update allowed by \( \varepsilon \)
Ownership abstraction in effect specs

Client can do little with a committed object. We can weaken the semantics of effect specs to allow update of committed objects. Amounts to only relying on their invariants.

\[
\text{Def } \models \{ P \} C \{ Q \} [\varepsilon] \quad \Delta \quad \text{for all } \sigma \in \llbracket P \rrbracket \text{ we have}
\]

- \( \llbracket C \rrbracket(\sigma) \neq \emptyset \);
- \( \llbracket C \rrbracket(\sigma) \neq \bot \) then \((\sigma, \tau) \in \llbracket Q \rrbracket\umberland \tau = \llbracket C \rrbracket(\sigma)\umberland \text{ and then:}
  - \( \tau(x) = \sigma(x) \) for every \( x \in \text{dom}(\Gamma) \), unless allowed by \( \varepsilon \)
  - \( \tau(p.f) = \sigma(p.f) \) for every allocated \( p \) and every field \( f \), unless \( \sigma(p.com) = \text{true} \) or update allowed by \( \varepsilon \)
Ownership abstraction in effect specs

Client can do little with a committed object. We can weaken the semantics of effect specs to allow update of committed objects.

Amounts to only relying on their invariants.

\[
\text{Def } \models \{ P \} C \{ Q \}[\varepsilon] \quad \Delta \quad \text{for all } \sigma \in [P] \text{ we have}
\]

- \([C](\sigma) \neq \bot;
- \([C](\sigma) \neq \bot\) then \((\sigma, \tau) \in [Q]\)
  where \(\tau = [C](\sigma)\); and then:
- \(\tau(x) = \sigma(x)\) for every \(x \in \text{dom}(\Gamma)\), unless allowed by \(\varepsilon\)
- \(\tau(p.f) = \sigma(p.f)\) for every allocated \(p\) and every field \(f\), unless \(\sigma(p.com) = \text{true}\) or update allowed by \(\varepsilon\)
Notes on Boogie methodology

Constructors: Initially \( \textit{inv}, \textit{own}, \textit{com} := \textit{false}, \text{null}, \textit{false} \);
end with \texttt{pack}.

Subclasses: \textit{own} ranges over pairs \((o, K)\)
\textit{inv} to ranges over superclasses of \texttt{type}(self).
Global invariants:
\[ \forall o, K \mid o.\textit{inv} <: K \Rightarrow J^K(o) \]
\[ \forall o \mid o.\text{com} \Rightarrow o.\textit{inv} = \texttt{type}(o) \]
(Where \(<: \) means subtype or equal)

unpack \texttt{x} from \texttt{K}:
precondition \(x.\textit{inv} = K\) and effect \(x.\textit{inv} := \text{super}(K)\)

Admissibility: If \(p.\textit{own} = (o, K)\) then an invariant \(J^L(o)\) may
depend on \(p\) for types \(L\) with \(\texttt{type}(o) <: L <: K\).
Notes on Boogie methodology

Constructors: Initially $\text{inv}, \text{own}, \text{com} := \text{false}, \text{null}, \text{false}$; end with pack.

Subclasses: own ranges over pairs $(o, K)$
inv to ranges over superclasses of type(self).
Global invariants: $\forall o, K \mid o.\text{inv} <: K \Rightarrow J^K(o)$
$\forall o \mid o.\text{com} \Rightarrow o.\text{inv} = \text{type}(o)$
(where $:<$ means subtype or equal)

unpack x from K:
precondition $x.\text{inv} = K$ and effect $x.\text{inv} := \text{super}(K)$

Admissibility: If $p.\text{own} = (o, K)$ then an invariant $J^L(o)$ may depend on $p$ for types $L$ with $\text{type}(o) <: L <: K$. 

Lecture 1
Model fields as ghosts

class Portion {
    private x,y: int := 0,1;
    view: View := ...;
    invar 0 ≤ x < y
    ghost f: real;
    invar f = 1/(y − x);

    meth m() requires inv {
        unpack self;
        x := x+1; view.notify(); y := y+1;
        f := 1/(y − x);
        pack self;
    }

    meth n() requires inv ∧ f < 0.1 {...}
}
Still need decentralized, cluster invariants

\[(NS = \text{null} \lor NS\_.prev = \text{null}) \land \forall p \in NS\_.next^* \ (p\_.next = \text{null} \lor p\_.next\_.prev = p) \land p\_.item \in Ilist\_.next^* \land (\exists x, j \ | \ Clist[x][j] = p\_.item \land p\_.item\_.where = p)\]
Summary

Hidden invariants: at risk due to reentrancy and sharing.

Ghost flag $inv$ in preconditions to prevent undesired reentrance. Ownership hierarchy to confine sharing; encode in ghost state to facilitate transfer.

Deploying a methodology:
- By hand, if assertion language expressive (we don’t need much) — but prescribed annotation of field updates;
- Baked into VC generator, e.g., methodology invariants as axioms in Spec#.
- I want to see it as part of module declarations; design/reasoning patterns.

General idea of $inv/own$ (“valid”) being used in Verisoft/MS hypervisor project [Schulte, Paul, et al]
Summary

Hidden invariants: at risk due to reentrancy and sharing.
Ghost flag *inv* in preconditions to prevent undesired reentrance.
Ownership hierarchy to confine sharing; encode in ghost state to facilitate transfer.

Deploying a methodology:
- By hand, if assertion language expressive (we don’t need much) —but prescribed annotation of field updates;
- Baked into VC generator, e.g., methodology invariants as axioms in Spec#.
- I want to see it as part of module declarations; design/reasoning patterns.

General idea of *inv/own* (“valid”) being used in Verisoft/MS hypervisor project [Schulte, Paul, et al]
Summary

Hidden invariants: at risk due to reentrancy and sharing.
Ghost flag inv in preconditions to prevent undesired reentrance.
Ownership hierarchy to confine sharing; encode in ghost state to facilitate transfer.

Deploying a methodology:

- By hand, if assertion language expressive (we don’t need much) — but prescribed annotation of field updates;
- Baked into VC generator, e.g., methodology invariants as axioms in Spec#.
- I want to see it as part of module declarations; design/reasoning patterns.

General idea of inv/own (“valid”) being used in Verisoft/MS hypervisor project [Schulte, Paul, et al]
Summary

Hidden invariants: at risk due to reentrancy and sharing.
Ghost flag \textit{inv} in preconditions to prevent undesired reentrance.
Ownership hierarchy to confine sharing; encode in ghost state to facilitate transfer.

Deploying a methodology:

- By hand, if assertion language expressive (we don’t need much) —but prescribed annotation of field updates;
- Baked into VC generator, e.g., methodology invariants as axioms in Spec#.
- I want to see it as part of module declarations; design/reasoning patterns.

General idea of \textit{inv/own} (“valid”) being used in Verisoft/MS hypervisor project [Schulte, Paul, et al]
Some references

[Leino, Müller’04] Object invariants in dynamic contexts (Boogie)

[Leino, Schulte’06] A verifying compiler for a multi-threaded object-oriented language (VC generation including Boogie methodology invariants; also extensions to concurrency)

[Naumann’07] On assertion-based encapsulation for object invariants and simulations (survey)

[Rehof, Mogensen’99] Tractable constraints in finite semilattices (Solver)