Verification of Object-oriented Programs
Lecture 0: framing and abstraction

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.
Course prospectus

- **Goal:** introduce programming disciplines for specifying and reasoning about programs in languages like Java, using *ordinary first order logic*.

- Pre/post/modifies specifications using *ghost state*, as in JML ([www.jmlspecs.org](http://www.jmlspecs.org)). But not a JML course.

- Single threaded O-O programs with (opaque) references, garbage collection, nominal typing and inheritance, ...

- Disciplines for encapsulation with inheritance, heap sharing, and re-entrant callbacks in design patterns.

- Other topics: Behavioral subtyping. Model programs for reasoning about “higher order methods”.
Course prospectus

- **Goal:** introduce programming disciplines for specifying and reasoning about programs in languages like Java, using **ordinary first order logic.**

- Pre/post/modifies specifications using **ghost state**, as in JML ([www.jmlspecs.org](http://www.jmlspecs.org)). But not a JML course.

- Single threaded O-O programs with (opaque) references, garbage collection, nominal typing and inheritance, ...

- Disciplines for encapsulation with inheritance, heap sharing, and re-entrant callbacks in design patterns.

- Other topics: Behavioral subtyping. Model programs for reasoning about “higher order methods”.
Course prospectus

- Goal: introduce programming disciplines for specifying and reasoning about programs in languages like Java, using ordinary first order logic.

- Pre/post/modifies specifications using ghost state, as in JML (www.jmlspecs.org). But not a JML course.

- Single threaded O-O programs with (opaque) references, garbage collection, nominal typing and inheritance, ...

- Disciplines for encapsulation with inheritance, heap sharing, and re-entrant callbacks in design patterns.

- Other topics: Behavioral subtyping. Model programs for reasoning about “higher order methods”
Course prospectus

- **Goal**: introduce programming disciplines for specifying and reasoning about programs in languages like Java, using ordinary first order logic.

- **Pre/post/modifies specifications** using **ghost state**, as in JML ([www.jmlspecs.org](http://www.jmlspecs.org)). But not a JML course.

- Single threaded O-O programs with (opaque) references, garbage collection, nominal typing and inheritance, ...

- Disciplines for encapsulation with inheritance, heap sharing, and re-entrant callbacks in design patterns.

- Other topics: Behavioral subtyping. Model programs for reasoning about “higher order methods”.
Course prospectus

- Goal: introduce programming disciplines for specifying and reasoning about programs in languages like Java, using ordinary first order logic.

- Pre/post/modifies specifications using ghost state, as in JML (www.jmlspecs.org). But not a JML course.

- Single threaded O-O programs with (opaque) references, garbage collection, nominal typing and inheritance, ...

- Disciplines for encapsulation with inheritance, heap sharing, and re-entrant callbacks in design patterns.

- Other topics: Behavioral subtyping. Model programs for reasoning about “higher order methods”.
Problem: modular reasoning

Class DB: instance represents a database.
Class View: instance provides read/write access to a DB.
Share internal data structures and appear together in a module.

Want to reason about their implementations independent from implementations of clients or others.
Reasoning by contract

```java
class Client {
    meth main() {
        xs: List := new List();
xs.add(5);
xs.add(3);
        assert xs.member(5);
    }
}

class List {
    private head: Node;
    model as: seq<int>;
meth add(v: int)
        requires true;
        ensures as = (v :: old(as));
    { ...(implementation)... }

    ...
}
```

Pre/post specifications, using 2-state postconditions.
Public specification using model (spec-only) field for abstraction.

Note: as is as is as.
Reasoning by contract

```java
class Client {
    meth main() {
        xs: List := new List();
        xs.add(5);
        xs.add(3);
        assert xs.member(5);
    }
}

class List {
    private head: Node;
    model as: seq<int>;
    meth add(v: int)
        requires true;
        ensures as = (v :: old(as));
    { ... (implementation) ... }
    ...
}

Pre/post specifications, using 2-state postconditions.
Public specification using model (spec-only) field for abstraction.

Note: as is as is as.
Reasoning by contract

```java
class Client {
    meth main()
        xs: List := new List();
        xs.add(5);
        xs.add(3);
        assert xs.member(5);
    }

class List {
    private head: Node;
    model as: seq<int>;
    meth add(v: int)
        requires true;
        ensures as = (v :: old(as));
        {...(implementation)...}
    ...
}
```

Pre/post specifications, using 2-state postconditions.
Public specification using model (spec-only) field for abstraction.

Note: as is as is as.
Reasoning by contract

class Client {
    meth main() {
        xs: List := new List();
xs.add(5);
xs.add(3);
        assert xs.member(5);
    }
}

class List {
    private head: Node;
    model as: seq<int>;
meth add(v: int)
    requires true;
    ensures as = (v :: old(as));
    {
        ...(implementation)...\}
    ...
}

Pre/post specifications, using 2-state postconditions.
Public specification using model (spec-only) field for abstraction.

Note: as is as is as.
Frame specifications

Reasoning about client: need to know what may change.

class Client {
    meth main() {
        xs: List := new List();
        ws: List := new List();
        xs.add(5);
        print( xs.member(4) );
        xs.add(3);
        assert xs.member(5) 
            ∧ ws.empty();
    }
}

class List {
    private head: Node;
    model as: seq<int>;
    meth add(v: int)
        ensures as = (v :: old(as));
        modifies as;
    { ... }
    meth member(v: int): bool
        ensures result = (v in as);
        (modifies nothing);
    { ... }
}
Frame specifications

Reasoning about client: need to know what may change.

class Client {
    meth main(){
        xs: List := new List();
        ws: List := new List();
        xs.add(5);
        print( xs.member(4) );
        xs.add(3);
        assert xs.member(5)
            ∧ ws.empty();
    }
}

class List {
    private head: Node;
    model as: seq<int>;
    meth add(v: int)
        ensures as = (v :: old(as));
        modifies as;
    {
        ... 
    }
    meth member(v: int): bool
        ensures result = (v in as);
        (modifies nothing);
    {
        ... 
    }...
Reasoning about method implementations

Intermediate assertions.
Connect internal data structure with “model state”.

class List {
  private head: Node;
  model as: seq<int>;
  meth add(int v) {
    ensures as = v::old(as);
    modifies as;
    { assert seq(head) = as = old(as);
      Node t:= new Node();
      t.item:= v; t.nxt:= head;
      assert seq(t) = v::old(as);
      head:= t;
    }
    ...
  }
  constraint as = seq(head);

  Define seq recursively
  seq(null) = []
  seq(p) = p.item :: seq(p.nxt)
Outline of lecture

Overview/teaser: modular reasoning

Verification logic

Program language

Framing and abstraction
A foundation (sketched): Hoare logic

Partial-correctness judgement: \( \{ P \} C \{ Q \} [\varepsilon] \)
where \( P \) and \( Q \) are state predicates, \( C \) a command, and \( \varepsilon \) the effect ("modifies") clause.

Meaning: for all states \( \sigma \), if \( \sigma \models P \) then \( C \) does not fault on \( \sigma \).
And \( \sigma \xrightarrow{C} \tau \) implies \( \tau \models Q \) and \( \tau \) differs from \( \sigma \) in accord with \( \varepsilon \).

Sample proof rules:

\[
\frac{\{ P \} C0 \{ R \} [\varepsilon_0] \quad \{ R \} C1 \{ Q \} [\varepsilon_1]}{\{ P \} C0; C1 \{ Q \} [\varepsilon_0, \varepsilon_1]}
\quad \frac{P \Rightarrow (Q/x\rightarrow E)}{\{ P \} x := E \{ Q \} [x]}
\]

First order validity or provability. Substitution of \( E \) for \( x \) in \( Q \).
A foundation (sketched): Hoare logic

Partial-correctness judgement: \( \{ P \} C \{ Q \}[\varepsilon] \)
where \( P \) and \( Q \) are state predicates, \( C \) a command, and \( \varepsilon \) the effect (“modifies”) clause.

Meaning: for all states \( \sigma \), if \( \sigma \models P \) then \( C \) does not fault on \( \sigma \).
And \( \sigma \xrightarrow{C} \tau \) implies \( \tau \models Q \) and \( \tau \) differs from \( \sigma \) in accord with \( \varepsilon \).

Sample proof rules:

\[
\frac{\{ P \} \ C0 \ { R }[\varepsilon_0] \quad \{ R \} \ C1 \ { Q }[\varepsilon_1]}{\{ P \} \ C0; \ C1 \ { Q }[\varepsilon_0, \varepsilon_1]}
\]

\[
\frac{P \Rightarrow (Q/x\rightarrow E)}{\{ P \} \ x := E \ { Q }[x]}
\]

First order validity or provability. Substitution of \( E \) for \( x \) in \( Q \).
A foundation (sketched): Hoare logic

Partial-correctness judgement: \( \{ P \} C \{ Q \}[\varepsilon] \)

where \( P \) and \( Q \) are state predicates, \( C \) a command, and \( \varepsilon \) the effect (“modifies”) clause.

Meaning: for all states \( \sigma \), if \( \sigma \models P \) then \( C \) does not fault on \( \sigma \).
And \( \sigma \xrightarrow{C} \tau \) implies \( \tau \models Q \) and \( \tau \) differs from \( \sigma \) in accord with \( \varepsilon \).

Sample proof rules:

\[
\begin{align*}
\{ P \} C0 \{ R \}[\varepsilon_0] & \quad \{ R \} C1 \{ Q \}[\varepsilon_1] \\
\{ P \} C0; C1 \{ Q \}[\varepsilon_0, \varepsilon_1] & \\
\{ P \} x := E \{ Q \}[x] \\
\end{align*}
\]

First order validity or provability. Substitution of \( E \) for \( x \) in \( Q \).
A foundation (sketched): Hoare logic

Partial-correctness judgement: \{P\} C \{Q\} [\epsilon]
where \(P\) and \(Q\) are state predicates, \(C\) a command, and \(\epsilon\) the effect ("modifies") clause.

Meaning: for all states \(\sigma\), if \(\sigma \models P\) then \(C\) does not fault on \(\sigma\).
And \(\sigma \xrightarrow{C} \tau\) implies \(\tau \models Q\) and \(\tau\) differs from \(\sigma\) in accord with \(\epsilon\).

Sample proof rules:

\[
\begin{align*}
\frac{\{P\} C0 \{R\} [\epsilon_0] \quad \{R\} C1 \{Q\} [\epsilon_1]}{\{P\} C0; C1 \{Q\} [\epsilon_0, \epsilon_1]} & \quad P \Rightarrow (Q/x \rightarrow E) \\
\frac{\{P\} x := E \{Q\} [x]}{\{P\} x := E \{Q\} [x]} & \quad \text{First order validity or provability. Substitution of } E \text{ for } x \text{ in } Q.}
\end{align*}
\]
A foundation (sketched): Hoare logic

Partial-correctness judgement: \{ P \} C \{ Q \}[\varepsilon]
where \( P \) and \( Q \) are state predicates, \( C \) a command, and \( \varepsilon \) the
\textit{effect} ("modifies") clause.

Meaning: for all states \( \sigma \), if \( \sigma \models P \) then \( C \) does not fault on \( \sigma \).
And \( \sigma \xrightarrow{C} \tau \) implies \( \tau \models Q \) and \( \tau \) differs from \( \sigma \) in accord with \( \varepsilon \).

Sample proof rules:

\[
\begin{align*}
\{ P \} C0 \{ R \}[\varepsilon_0] & \quad \{ R \} C1 \{ Q \}[\varepsilon_1] \\
\{ P \} C0; C1 \{ Q \}[\varepsilon_0, \varepsilon_1] & \quad P \Rightarrow (Q/x\rightarrow E) \\
& \quad \{ P \} x := E \{ Q \}[x]
\end{align*}
\]

First order validity or provability. Substitution of \( E \) for \( x \) in \( Q \).
On 2-state postconditions

For method specifications, want 2-state postconditions.

For 2-state $Q$, the judgement $\{ P \} C \{ Q \}[\epsilon]$ means for all states $\sigma$, if $\sigma \models P$ then $C$ does not fault on $\sigma$. And $\sigma \xrightarrow{C} \tau$ implies $(\sigma, \tau) \models Q$ and $\tau$ differs from $\sigma$ in accord with $\epsilon$.

In JML, old is with respect to pre-state of method invocation.

\[
x := x + 1;
\]
\[
assert x = \text{old}(x) + 1;
\]
\[
x := x + 1;
\]
\[
assert x = \text{old}(x) + 2;
\]

Introduce fresh variable oldx and work with 1-state predicates:

\[
\text{oldx} := x; \ x := x + 1; \ assert x = \text{oldx} + 1; \ x := x + 1; \ assert x = \text{oldx} + 2;
\]
On 2-state postconditions

For method specifications, want 2-state postconditions.

For 2-state $Q$, the judgement $\{ P \} C \{ Q \}[\varepsilon]$ means for all states $\sigma$, if $\sigma \models P$ then $C$ does not fault on $\sigma$. And $\sigma \xrightarrow{C} \tau$ implies $(\sigma, \tau) \models Q$ and $\tau$ differs from $\sigma$ in accord with $\varepsilon$.

In JML, $\text{old}$ is with respect to pre-state of method invocation.

\[
\begin{align*}
x & := x+1; \\
\text{assert } x & = \text{old}(x)+1; \\
x & := x+1; \\
\text{assert } x & = \text{old}(x)+2;
\end{align*}
\]

Introduce fresh variable $\text{oldx}$ and work with 1-state predicates:

\[
\begin{align*}
\text{oldx} & := x; x := x+1; \text{assert } x = \text{oldx}+1; x := x+1; \text{assert } x = \text{oldx}+2;
\end{align*}
\]
On 2-state postconditions

For method specifications, want 2-state postconditions.

For 2-state $Q$, the judgement $\{ P \} C \{ Q \}[\varepsilon]$ means for all states $\sigma$, if $\sigma \models P$ then $C$ does not fault on $\sigma$. And $\sigma \xrightarrow{C} \tau$ implies $(\sigma, \tau) \models Q$ and $\tau$ differs from $\sigma$ in accord with $\varepsilon$.

In JML, old is with respect to pre-state of method invocation.

\[
\begin{align*}
x & := x+1; \\
\text{assert } x & = \text{old}(x)+1; \\
x & := x+1; \\
\text{assert } x & = \text{old}(x)+2;
\end{align*}
\]

Introduce fresh variable old$x$ and work with 1-state predicates:

\[
\begin{align*}
\text{old$x$} & := x; x:=x+1; \text{assert } x = \text{old$x$}+1; x:=x+1; \text{assert } x = \text{old$x$}+2;
\end{align*}
\]
Reasoning about procedure call

Hypothetical judgement $\Delta \vdash \{ P \} C \{ Q \}[\varepsilon]$ where $\Delta$ is a list of procedure specifications.

Call axiom (where scopes/types ok):

$$\Delta, \{ P \} m(y) \{ Q \}[\varepsilon] \vdash \{ \text{P/y}\rightarrow\text{E} \} m(\text{E}) \{ \text{Q/y}\rightarrow\text{E} \}[\ldots]$$

For dynamically dispatched calls, we need behavioral subtyping — in subsequent lecture.

Linking rule:

$$\Delta, \{ P \} m(y) \{ Q \}[\varepsilon] \vdash \{ R \} C_0 \{ S \}[\ldots]$$
$$\Delta \vdash \{ P \} C_m \{ Q \}[\varepsilon]$$
$$\Delta \vdash \{ R \} \text{let } m(y) = C_m \text{ in } C_0 \{ S \}[\ldots]$$
Reasoning about procedure call

Hypothetical judgement \( \Delta \vdash \{ P \} C \{ Q \}[\varepsilon] \) where \( \Delta \) is a list of procedure specifications.

*Call axiom* (where scopes/types ok):

\[
\Delta, \{ P \} m(y) \{ Q \}[\varepsilon] \vdash \{ P/y\rightarrow E \} m(E) \{ Q/y\rightarrow E \}[\ldots]
\]

For dynamically dispatched calls, we need *behavioral subtyping* — in subsequent lecture.

*Linking rule:*

\[
\Delta, \{ P \} m(y) \{ Q \}[\varepsilon] \vdash \{ R \} C_0 \{ S \}[\ldots] \\
\Delta \vdash \{ P \} C_m \{ Q \}[\varepsilon] \\
\Delta \vdash \{ R \} \text{ let } m(y) = C_m \text{ in } C_0 \{ S \}[\ldots]
\]
Reasoning about procedure call

Hypothetical judgement $\Delta \vdash \{P\} C \{Q\}[\varepsilon]$ where $\Delta$ is a list of procedure specifications.

*Call axiom* (where scopes/types ok):

$\Delta,\{P\} m(y) \{Q\}[\varepsilon] \vdash \{P/y \rightarrow E\} m(E) \{Q/y \rightarrow E\}[\ldots]$  

For dynamically dispatched calls, we need *behavioral subtyping* — in subsequent lecture.

*Linking rule*:

$\Delta,\{P\} m(y) \{Q\}[\varepsilon] \vdash \{R\} C_0 \{S\}[\ldots]$

$\Delta \vdash \{P\} C_m \{Q\}[\varepsilon]$

$\Delta \vdash \{R\}$ let $m(y) = C_m$ in $C_0 \{S\}[\ldots]$
Verification conditions (VCs)

**Annotated code:**

```
assert P;
x := E;
assert Q;
```

**Assignment rule:**

\[
P \Rightarrow (Q / x \rightarrow E) \\
\{ P \} x := E \{ Q \}[x]
\]

Verification condition \( P \Rightarrow (Q / x \rightarrow E) \).

VCs are often generated using **weakest preconditions:**

\[ \{ P \} C \{ Q \}[...] \text{ iff } P \Rightarrow \wp(C)(Q), \text{ where } \wp \text{ is definable} \]

provided loop invariants and method specs are given.

**Amounts to validity of** \( \wp(\text{assume } P; C; \text{assert } Q)(\text{true}) \).

**Def:** \( \wp(\text{assert } P)(Q) \triangleq P \land Q \)

**Def:** \( \wp(\text{assume } P)(Q) \triangleq P \Rightarrow Q \)
Verification conditions (VCs)

Annotated code:

\[ \begin{align*}
\text{assert } P ; \\
x & := E ; \\
\text{assert } Q ;
\end{align*} \]

Assignment rule:

\[ \begin{align*}
P \Rightarrow ( Q / x \to E ) \\
\{ P \} \ x := E \ { Q \} [x]
\end{align*} \]

Verification condition \( P \Rightarrow ( Q / x \to E ) \).

VCs are often generated using weakest preconditions:
\( \{ P \} \ C \ { Q \} [...] \) iff \( P \Rightarrow wp(C)(Q) \), where \( wp \) is definable provided loop invariants and method specs are given.

Amounts to validity of \( wp(assume \ P ; C ; assert \ Q ) (\text{true}) \).

Def: \( wp(\text{assert } P)(Q) \triangleq P \land Q \)
Def: \( wp(\text{assume } P)(Q) \triangleq P \Rightarrow Q \)
Verification conditions (VCs)

Annotated code:

\[
\begin{align*}
\text{assert } P; \\
x := E; \\
\text{assert } Q;
\end{align*}
\]

Assignment rule:

\[
\begin{align*}
P & \Rightarrow (Q/x \rightarrow E) \\
\{ P \} \; x := E \; \{ Q \} [x]
\end{align*}
\]

Verification condition \( P \Rightarrow (Q/x \rightarrow E) \).

VCs are often generated using \textit{weakest preconditions}:

\[
\{ P \} \; C \; \{ Q \} [\ldots] \text{ iff } P \Rightarrow \wp(C)(Q), \text{ where } \wp \text{ is definable}
\]

provided loop invariants and method specs are given.

Amounts to validity of \( \wp(\text{assume } P; C; \text{assert } Q)(true) \).

Def: \( \wp(\text{assert } P)(Q) \triangleq P \wedge Q \)

Def: \( \wp(\text{assume } P)(Q) \triangleq P \Rightarrow Q \)
Verification conditions (VCs)

Annotated code:

assert $P$;
$x := E$;
assert $Q$;

Assignment rule:

$$\frac{P \Rightarrow (Q/x \rightarrow E)}{\{P\} \ x := E \ {Q} \ [x]}$$

Verification condition $P \Rightarrow (Q/x \rightarrow E)$.

VCs are often generated using weakest preconditions:

$$\{P\} \ C \ {Q} \ [\ldots] \text{ iff } P \Rightarrow \text{wp}(C)(Q), \text{ where wp is definable}$$

provided loop invariants and method specs are given.

Amounts to validity of $\text{wp(assume } P; C; \text{assert } Q)(true)$.

Def: $\text{wp(assume } P)(Q) \triangleq P \land Q$

Def: $\text{wp(assume } P)(Q) \triangleq P \Rightarrow Q$
A verification system architecture

User → Analyze

Code, specs, annotation → Translate

Specification
\{ P \} m(y) \{ Q \}[\varepsilon]

- VC for the method body \( C_m \) may take form
  assume \( P; C_m; \) assert \( Q \land VC(\varepsilon) \)

- To verify caller of the form \( \ldots; m(E); \ldots \)
  replace call by
  assert \( P/y \rightarrow E; \)
  havoc \( \varepsilon; \) assume \( Q/y \rightarrow E. \)

Generate

Annotated IL (with invars)

Prove

Proof or error info

VCs + semantics/math axioms/defs

Modular verification of each method w.r.t. specs of callees.

Prover: SMT solver (decision procedures + quantifier heuristics),
possibly combined with interactive prover
A verification system architecture

User → Code, specs, annotation → Analyze

Code, specs, annotation → Translate

Translate → Annotated IL (with invars) → Generate

Annotated IL (with invars) → VCs + semantics/math axioms/defs → Prove

Prove → Proof or error info → Prove

Specification
\[ \{ P \} \ m(y) \ \{ Q \} [\varepsilon] \]

- VC for the method body \( C_m \) may take form
  assume \( P; C_m; \) assert \( Q \land VC(\varepsilon) \)

- To verify caller of the form \( \ldots; m(E); \ldots \)
  replace call by
  assert \( P/y \rightarrow E; \)
  havoc \( \varepsilon \); assume \( Q/y \rightarrow E. \)

Modular verification of each method w.r.t. specs of callees.

Prover: SMT solver (decision procedures + quantifier heuristics), possibly combined with interactive prover.
A verification system architecture

User ➔ Code, specs, annotation ➔ Analyze ➔ Translate ➔ Generate

Annotated IL (with invars) ➔ VCs + semantics/math axioms/defs ➔ Prove ➔ Proof or error info

Specify:
\[ \{ P \} m(y) \{ Q \}[\varepsilon] \]

- VC for the method body \( C_m \) may take form
\[ \text{assume } P; C_m; \text{assert } Q \land VC(\varepsilon) \]

- To verify caller of the form \( \ldots; m(E); \ldots \)
replace call by
\[ \text{assert } P/y \rightarrow E; \]
\[ \text{havoc } \varepsilon; \text{assume } Q/y \rightarrow E. \]

Modular verification of each method w.r.t. specs of callees.

Prover: SMT solver (decision procedures + quantifier heuristics), possibly combined with interactive prover
Why **First Order** logic assertions?

- Much code is first order. Some OO design patterns are FO means to use Higher Order concepts.
- Many mature tools exist for FO validity/satisfiability; simple semantics facilitates shallow embedding in HO provers.
- Fast decision procedures exist for useful fragments (linear arithmetic, arrays, ...). We’ll use fixed interpretation of mathematical types (integers, sets,...)
- Adequate axiomatizations for finite sets (also decision procedures [Kuncak]).
- But reachability is problematic (hard to find useful, decidable fragments of FO+transitive closure).
  Later we’ll consider how to avoid predicates like \( \forall p \in \text{head.nxt}^* \mid p.nxt.prev = p \).
- Frame conditions: \( \forall p, f \mid p.f = \text{old}(p.f) \lor "p.f" \in \text{modifs}(\varepsilon) \)
Why First Order logic assertions?

- Much code is first order. Some OO design patterns are FO means to use Higher Order concepts.
- Many mature tools exist for FO validity/satisfiability; simple semantics facilitates shallow embedding in HO provers.
- Fast decision procedures exist for useful fragments (linear arithmetic, arrays, ...). We’ll use fixed interpretation of mathematical types (integers, sets,...)
- Adequate axiomatizations for finite sets (also decision procedures [Kuncak]).
- But reachability is problematic (hard to find useful, decidable fragments of FO+transitive closure).
  Later we’ll consider how to avoid predicates like $\forall p \in \text{head}.\text{nxt}^* \mid p.\text{nxt}.\text{prev} = p$.
- Frame conditions: $\forall p, f \mid p.f = \text{old}(p.f) \lor "p.f" \in \text{modifs}(\varepsilon)$
Why First Order logic assertions?

- Much code is first order. Some OO design patterns are FO means to use Higher Order concepts.
- Many mature tools exist for FO validity/satisfiability; simple semantics facilitates shallow embedding in HO provers.
- Fast decision procedures exist for useful fragments (linear arithmetic, arrays, ...). We’ll use fixed interpretation of mathematical types (integers, sets,...)
- Adequate axiomatizations for finite sets (also decision procedures [Kuncak]).
- But reachability is problematic (hard to find useful, decidable fragments of FO+transitive closure). Later we’ll consider how to avoid predicates like \( \forall p \in \text{head.nxt}^* \mid p.nxt.prev = p \).
- Frame conditions: \( \forall p, f \mid p.f = \text{old}(p.f) \lor \text{“p.f”} \in \text{modifs}(\epsilon) \)
Why First Order logic assertions?

- Much code is first order. Some OO design patterns are FO means to use Higher Order concepts.
- Many mature tools exist for FO validity/satisfiability; simple semantics facilitates shallow embedding in HO provers.
- Fast decision procedures exist for useful fragments (linear arithmetic, arrays, ...). We’ll use fixed interpretation of mathematical types (integers, sets, ...)
- Adequate axiomatizations for finite sets (also decision procedures [Kuncak]).
- But reachability is problematic (hard to find useful, decidable fragments of FO+transitive closure).
Later we’ll consider how to avoid predicates like \( \forall p \in head.nxt^* \mid p.nxt.prev = p \).
- Frame conditions: \( \forall p, f \mid p.f = old(p.f) \lor “p.f” \in \text{modifs}(\varepsilon) \)
Why **First Order** logic assertions?

- Much code is first order. Some OO design patterns are FO means to use Higher Order concepts.
- Many mature tools exist for FO validity/satisfiability; simple semantics facilitates shallow embedding in HO provers.
- Fast decision procedures exist for useful fragments (linear arithmetic, arrays, ...). We’ll use fixed interpretation of mathematical types (integers, sets,...)
- Adequate axiomatizations for finite sets (also decision procedures [Kuncak]).
- But reachability is problematic (hard to find useful, decidable fragments of FO+transitive closure). Later we’ll consider how to *avoid* predicates like \( \forall p \in \text{head.nxt}^* | p.nxt.prev = p \).
- Frame conditions: \( \forall p, f | p.f = \text{old}(p.f) \lor \text{“p.f”} \in \text{modifs}(\varepsilon) \)
Why _First Order_ logic assertions?

- Much code is first order. Some OO design patterns are FO means to use Higher Order concepts.
- Many mature tools exist for FO validity/satisfiability; simple semantics facilitates shallow embedding in HO provers.
- Fast decision procedures exist for useful fragments (linear arithmetic, arrays, ...) _We’ll use fixed interpretation of mathematical types_ (integers, sets, ...)
- Adequate axiomatizations for finite sets (also decision procedures [Kuncak]).
- But reachability is problematic (hard to find useful, decidable fragments of FO+transitive closure). Later we’ll consider how to _avoid_ predicates like
  \[ \forall p \in \text{head}.nxt^* \mid p.nxt.prev = p. \]
- Frame conditions: \[ \forall p, f \mid p.f = \text{old}(p.f) \lor \text{“p.f”} \in \text{modifs}(\varepsilon) \]
Program language

*Class table:* collection of well formed classes.
*Class declaration:* methods; fields \( f : T \); superclass.

*Method declaration:* \( \text{meth } m(\overline{x} : \overline{T}) : T \{ C \} \) in class \( K \) such that \( \Gamma \vdash C \) where context, \( \Gamma \), is \( \overline{x} : \overline{T}, \text{self} : K, \text{result} : T \).

“private” means class scope.

\[
x, y \in \text{VarName} \quad f, g \in \text{FieldName} \quad K \in \text{DeclaredClassName} \\
T ::= \text{int} | K \\
E ::= x | c | \text{null} | E \oplus E \quad (c \text{ is in } \mathbb{Z}, \oplus \text{ is in } \{=, +, -, *, >, \ldots \}) \\
C ::= x := E \mid x := \text{new } K \mid x := x.f \mid x.f := E \\
\mid \text{if } x \text{ then } C \text{ else } C \mid \text{while } x \text{ do } C \mid C ; C \\
\mid \text{var } x : T \text{ in } C \text{ end}
\]
Program language semantics

Given infinite set \texttt{Ref} of references and \texttt{null} not in \texttt{Ref}.

\textit{Ref context:} finite partial function \( r : \texttt{Ref} \rightarrow \texttt{Types} \).

\texttt{Def} \( \texttt{Val}(T, r) \triangleq \{ \texttt{null} \} \cup \{ p \mid p \in \texttt{dom}(r) \land r(p) \leq T \} \).

\texttt{Store:} \( s \in \texttt{Store}(\Gamma, r) \) if \( s \) maps each \( x \) in \( \Gamma \) to some \( s(x) \in \texttt{Val}(T, r) \) where \( x : T \) in \( \Gamma \).

\texttt{Def} \( \texttt{Store}(\Gamma, r) \triangleq (x : \texttt{dom}(\Gamma)) \rightarrow \texttt{Val}(\Gamma(x), r) \).

\texttt{Heap:} \( h \in \texttt{Heap}(r) \) if \( h \) maps each \( p \in \texttt{dom}(r) \) to some store for \( \texttt{fields}(K) \) where \( K \) is \( r(p) \).

\texttt{Def} \( \texttt{Heap}(r) \triangleq (p : \texttt{dom}(r)) \rightarrow \texttt{Store}(\texttt{fields}(r(p))) \).

\texttt{State:} \( (r, h, s) \) where \( h \in \texttt{Heap}(r) \) and \( s \in \texttt{Store}(\Gamma, r) \).

Abbreviate state as \( \sigma \), writing \( \sigma(x) \), \( \sigma(o.f) \), etc.
Program language semantics

Given infinite set \texttt{Ref} of references and \texttt{null} not in \texttt{Ref}.

\textit{Ref context:} finite partial function \( r : \text{Ref} \rightarrow \text{Types} \).

\textbf{Def} \( \text{Val}(T, r) \triangleq \{ \text{null} \} \cup \{ p \mid p \in \text{dom}(r) \land r(p) \leq T \} \).

\textit{Store:} \( s \in \text{Store}(\Gamma, r) \) if \( s \) maps each \( x \) in \( \Gamma \) to some \( s(x) \in \text{Val}(T, r) \) where \( x : T \) in \( \Gamma \).

\textbf{Def} \( \text{Store}(\Gamma, r) \triangleq (x : \text{dom}(\Gamma)) \rightarrow \text{Val}(\Gamma(x), r) \).

\textit{Heap:} \( h \in \text{Heap}(r) \) if \( h \) maps each \( p \in \text{dom}(r) \) to some store for \( \text{fields}(K) \) where \( K \) is \( r(p) \).

\textbf{Def} \( \text{Heap}(r) \triangleq (p : \text{dom}(r)) \rightarrow \text{Store}(\text{fields}(r(p))) \).

\textit{State:} \( (r, h, s) \) where \( h \in \text{Heap}(r) \) and \( s \in \text{Store}(\Gamma, r) \).

Abbreviate state as \( \sigma \), writing \( \sigma(x), \sigma(o.f) \), etc.
Program language semantics

Given infinite set Ref of references and null not in Ref.

Ref context: finite partial function \( r : \text{Ref} \rightarrow \text{Types} \).

Def \( \text{Val}(T, r) \triangleq \{ \text{null} \} \cup \{ p \mid p \in \text{dom}(r) \land r(p) \leq T \} \).

Store: \( s \in \text{Store}(\Gamma, r) \) if \( s \) maps each \( x \) in \( \Gamma \) to some \( s(x) \in \text{Val}(T, r) \) where \( x : T \) in \( \Gamma \).

Def \( \text{Store}(\Gamma, r) \triangleq (x : \text{dom}(\Gamma)) \rightarrow \text{Val}(\Gamma(x), r) \).

Heap: \( h \in \text{Heap}(r) \) if \( h \) maps each \( p \in \text{dom}(r) \) to some store for \( \text{fields}(K) \) where \( K \) is \( r(p) \).

Def \( \text{Heap}(r) \triangleq (p : \text{dom}(r)) \rightarrow \text{Store}(\text{fields}(r(p))) \).

State: \((r, h, s)\) where \( h \in \text{Heap}(r) \) and \( s \in \text{Store}(\Gamma, r) \).

Abbreviate state as \( \sigma \), writing \( \sigma(x), \sigma(o.f) \), etc.
Program language semantics

Given infinite set $\text{Ref}$ of references and $\text{null}$ not in $\text{Ref}$.

*Ref context:* finite partial function $r : \text{Ref} \rightarrow \text{Types}$.

**Def** $Val(T, r) \triangleq \{\text{null}\} \cup \{p \mid p \in \text{dom}(r) \land r(p) \leq T\}$.

*Store:* $s \in \text{Store}(\Gamma, r)$ if $s$ maps each $x$ in $\Gamma$ to some $s(x) \in Val(T, r)$ where $x : T$ in $\Gamma$.

**Def** $\text{Store}(\Gamma, r) \triangleq (x : \text{dom}(\Gamma)) \rightarrow Val(\Gamma(x), r)$.

*Heap:* $h \in \text{Heap}(r)$ if $h$ maps each $p \in \text{dom}(r)$ to some store for $\text{fields}(K)$ where $K$ is $r(p)$.

**Def** $\text{Heap}(r) \triangleq (p : \text{dom}(r)) \rightarrow \text{Store}(\text{fields}(r(p)))$.

*State:* $(r, h, s)$ where $h \in \text{Heap}(r)$ and $s \in \text{Store}(\Gamma, r)$.

Abbreviate state as $\sigma$, writing $\sigma(x)$, $\sigma(o.f)$, etc.
Semantics of commands and correctness

Denotational semantics: \([\Gamma \vdash C]\) is total function sending each \(\Gamma\)-state \(\sigma\) to \(\bot\) (fault), \(\perp\) (divergence), or a \(\Gamma\)-state \(\tau\).

Faults: null dereference. Typing implies no dangling refs. Formulas denote sets of states (or pairs, if \textbf{old} used).

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

- \([C](\sigma) \neq \bot;
- \text{if } [C](\sigma) \neq \perp \text{ then } (\sigma, \tau) \in [Q]
- \text{where } \tau = [C](\sigma); \text{ and then:}
- \tau(x) = \sigma(x) \text{ for every } x \in \text{dom}(\Gamma), \text{ unless allowed by } \varepsilon
- \tau(p.f) = \sigma(p.f) \text{ for every allocated } p \text{ and every field } f, \text{ unless allowed by } \varepsilon

But what exactly is the effect clause \(\varepsilon\)?
Semantics of commands and correctness

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

Faults: null dereference. Typing implies no dangling refs. Formulas denote sets of states (or pairs, if old used).

**Def** $\models \{P\} C \{Q\} [\varepsilon]$ $\triangle$ for all $\sigma \in [P]$ we have

- $[C](\sigma) \neq \bot$;
- if $[C](\sigma) \neq \perp$ 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 allowed by $\varepsilon$

But what exactly is the effect clause $\varepsilon$?
Semantics of commands and correctness

Denotational semantics: $[\Gamma \vdash C]$ 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 (or pairs, if old used).

**Def:** $\models \{ P \} C \{ Q \}[\varepsilon] \triangleq$ for all $\sigma \in [P]$ we have

- $[C](\sigma) \neq \bot$;
- if $[C](\sigma) \neq \bot$ 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 allowed by $\varepsilon$

But what exactly is the effect clause $\varepsilon$?
Semantics of commands and correctness

Denotational semantics: $[\Gamma \vdash C]$ 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 (or pairs, if old used).

**Def** $\vdash \{P\} C \{Q\} [\varepsilon]$ $\triangleq$ for all $\sigma \in [P]$ we have

- $[C](\sigma) \neq \bot$;
- if $[C](\sigma) \neq \bot$ 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 allowed by $\varepsilon$

But what exactly is the effect clause $\varepsilon$?
Framing – client’s view

Client of List:

```java
xs: List:= new List; j: int := 0;
xs.add(5);
assert 5 in xs.as ∧ j=0;
xs.add(4 );
assert 5 in xs.as ∧ j=0;
```

Rule of invariance:

\[
\{ P \} C \{ Q \}[\varepsilon]
\]

\[
R \text{ independent from } \varepsilon
\]

\[
\{ P \land R \} C \{ Q \land R \}[\varepsilon]
\]

To conclude 5 in xs.as, use functional spec:

```
{ true } add(v) { as = v :: old(as) } [ wr as]
```

To conclude j=0, use “frame axiom”, embodied in rule above.

Beautiful alternative: separation logic.
Framing – client’s view

Client of List:

\[ \text{xs: List:= new List; j: int := 0;} \]
\[ \text{xs.add(5);} \]
\[ \text{assert 5 in xs.as \land j=0; } \]
\[ \text{xs.add(4);} \]
\[ \text{assert 5 in xs.as \land j=0; } \]

Rule of invariance:

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

To conclude \( 5 \text{ in } \text{xs.as} \), use functional spec:

\[ \{ \text{true} \} \text{ add(v)} \{ \text{as = v :: old(as)} \} [\text{wr as}] \]

To conclude \( j=0 \), use “frame axiom”, embodied in rule above.

Beautiful alternative: separation logic.
Framing – client’s view

Client of List:

\[
\begin{align*}
\text{xs: List} & := \text{new List; } j \text{ int := 0;} \\
\text{xs.add(5);} \\
\text{assert 5 in xs.as } \land \text{ } j=0; \\
\text{xs.add(4) ;} \\
\text{assert 5 in xs.as } \land \text{ } j=0;
\end{align*}
\]

Rule of invariance:

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

To conclude \(5 \text{ in xs.as}\), use functional spec:

\[
\begin{align*}
\{\text{true}\} \ \text{add(v) } \{as = v :: \text{old(as)}\} \ [\text{wr as}]
\end{align*}
\]

To conclude \(j=0\), use “frame axiom”, embodied in rule above.

Beautiful alternative: separation logic.
class List {
    private head: Node;
    model as: seq<int>  constraint as = f(head);
    meth add(v: int)
        ensures as = v::old(as);
        effects wr as;
    { Node t:= new Node;
        t.item:= v;  t.nxt:= head;  head:= t; } ... } 

Client’s reading of “wr as” is that \([xs.add(v)](\sigma) = \tau\) implies 
\(\tau(p.f) = \sigma(p.f)\) for every object/field \(p.f\) except \(xs.as\).

But \(as\) isn’t in the program state.
And the code writes \(xs.head\) and \(t.nxt\).
Framing and abstraction – internal view 2

class List {
   private head: Node; private j: int;
   model as: seq<int>  constraint as = f(head);
   meth add(v: int)
      ensures as = v::old(as);
      effects wr as;     // write of j not allowed
   {
      t: Node := new Node;
      t.item:= v;     t.nxt:= head;     head:= t;     j:=1;  }
   }

“wr as” must allow write of head but disallow write of j if it can be indirectly observed by client, like head is.

Revised client interpretation: \([\text{xs.add(v)}](\sigma) = \tau\) implies \(\tau(p.f) = \sigma(p.f)\) for every allocated \(p\) and every \(f\) in scope for the client, except \(\text{xs.as}\) and whatever depends on \(as\).

[Leino,Nelson,Müller]
Framing and field abstraction: more problems

Idea: a public field $f$ may be declared to depend on another field $g$. If $g$ is in scope then so is the dependency. Permission to write $f$ implies writing $g$ too.

- **Mis-match** between interpretation of effect at call site and in VCs for implementation (e.g., using axiomatic semantic style).

- **Interference via sharing**: Suppose client calls method on $xs$ that returns $xs.%head$ and then invokes $xs.%incrAll()$ with effect $wr$ $xs.%as$. (Next lecture)

- **Framing for query methods**: Suppose client calls $xs.%member(3)$ and then some other method on $xs$; does the latter affect the value of $xs.%member(3)$?
Framing and field abstraction: more problems

Idea: a public field $f$ may be declared to depend on another field $g$. If $g$ is in scope then so is the dependency. Permission to write $f$ implies writing $g$ too.

- **Mis-match** between interpretation of effect at call site and in VCs for implementation (e.g., using axiomatic semantic style).

- **Interference via sharing**: Suppose client calls method on $xs$ that returns $xs\.head$ and then invokes $xs\.incrAll()$ with effect $wr \ xs\.as$. (Next lecture)

- **Framing for query methods**: Suppose client calls $xs\.member(3)$ and then some other method on $xs$; does the latter affect the value of $xs\.member(3)$?
Framing and field abstraction: more problems

Idea: a public field $f$ may be declared to depend on another field $g$. If $g$ is in scope then so is the dependency. Permission to write $f$ implies writing $g$ too.

- **Mis-match** between interpretation of effect at call site and in VCs for implementation (e.g., using axiomatic semantic style).
- **Interference via sharing**: Suppose client calls method on $xs$ that returns $xs\.head$ and then invokes $xs\.incrAll()$ with effect $wr \ xs\.as$. (Next lecture)
- **Framing for query methods**: Suppose client calls $xs\.member(3)$ and then some other method on $xs$; does the latter affect the value of $xs\.member(3)$?
Framing for query methods

Example using method calls in assertions. (If this seems dubious, replace with other form of opaque predicate, e.g., model field.)

Client of List:

```
x: List;
x.add(5);
assert x.size() = 1;
x.incrAll();
assert x.size() = 1;
```

Rule of invariance:

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

Can we avoid costly functional reasoning (e.g., defn of \textit{as})?

\textit{size()} reads the \textit{nxt} fields but not the \textit{items}, and \textit{incrAll()} writes \textit{items} but not \textit{nxt} fields.
Meaning and effects

Clause \( \text{wr } y, \text{wr } x.f \) means final value of \( y \) and of \( x.f \) may differ from initial, but nothing else may. (What if \( x \) also writable?)

Need read effects of formulas and query methods for framing. \( \text{rd } z \) means dependence on initial value of \( z \).

Obvious syntactic checks conservative; consider
\[
\{ \text{x:int; x:= y; y:= z; y:= x; y:= z - z; } \}
\]

Meaning of write clause checkable using 2-state postcondition.

Of course with concurrency, intermediate effects matter.
Meaning and effects

Clause \([\text{wr } y, \text{wr } x.f]\) means final value of \(y\) and of \(x.f\) may differ from initial, but nothing else may. (What if \(x\) also writable?)

Need read effects of formulas and query methods for framing. \(\text{rd } z\) means dependence on initial value of \(z\).

Obvious syntactic checks conservative; consider
\[
\{ \ x: \text{int}; \ x := y; \ y := z; \ y := x; \ y := z - z; \ \}
\]

Meaning of write clause checkable using 2-state postcondition.

Of course with concurrency, intermediate effects matter.
Summary

Programs composed of modules containing classes. One or more instances present an abstraction that is implemented using internal representation that may involve other instances.

Reason about clients using pre/post/effect specifications of methods in the module. Clients need to know which method invocations alter the abstraction.

Reason about implementation in terms of the representation.

Class-based program language (simple denotational semantics).

Verifier based on weakest-precondition calculation. First-order assertion language. Heap reachability conditions natural but costly.
Summary

Programs composed of modules containing classes. One or more instances present an *abstraction* that is implemented using *internal representation* that may involve other instances.

Reason about *clients* using pre/post/effect specifications of methods in the module. Clients need to know which method invocations alter the abstraction.

Reason about *implementation* in terms of the representation.

Class-based program language (simple denotational semantics).

Verifier based on weakest-precondition calculation. First-order assertion language. Heap reachability conditions natural but costly.
Summary

Programs composed of modules containing classes. One or more instances present an abstraction that is implemented using internal representation that may involve other instances.

Reason about clients using pre/post/effect specifications of methods in the module. Clients need to know which method invocations alter the abstraction.

Reason about implementation in terms of the representation.

Class-based program language (simple denotational semantics).

Verifier based on weakest-precondition calculation. First-order assertion language. Heap reachability conditions natural but costly.
Summary

Programs composed of modules containing classes. One or more instances present an abstraction that is implemented using internal representation that may involve other instances.

Reason about clients using pre/post/effect specifications of methods in the module. Clients need to know which method invocations alter the abstraction.

Reason about implementation in terms of the representation.

Class-based program language (simple denotational semantics).

Verifier based on weakest-precondition calculation. First-order assertion language. Heap reachability conditions natural but costly.
Summary

Programs composed of modules containing classes. One or more instances present an abstraction that is implemented using internal representation that may involve other instances.

Reason about clients using pre/post/effect specifications of methods in the module. Clients need to know which method invocations alter the abstraction.

Reason about implementation in terms of the representation.

Class-based program language (simple denotational semantics).

Verifier based on weakest-precondition calculation. First-order assertion language. Heap reachability conditions natural but costly.
Selected references


[Leavens et al] JML (Includes model fields and dependencies, ghost fields, method calls in assertions, Universe Types for ownership encapsulation.)

[Leavens,Leino,Müller ’07] Specification and verification challenges for sequential OO programs. (Says read effects are most promising for framing and encapsulation.)

[Leavens,Naumann,Rosenberg ’06] CoreJML (Denotational semantics encoded in PVS theorem prover.)
Plan of lectures

0. Background, framing and abstraction
1. Invariants, hiding, and ownership
2. Cluster invariants and dynamic framing
3. Hoare’s mismatch, region logic and second order framing.
4. Behavioral subtyping: static reasoning for dynamic dispatch
5. Model programs: first order reasoning for higher order methods
6. Relational properties, opportunistic additions and wrap-up