Software Verification (Autumn 2015)
Lecture 17: Separation Logic for Object-Orientation

Chris Poskitt

(adapted from material by Stephan van Staden, Matthew Parkinson, and Gavin Bierman)
Verifying object-oriented programs

- **object-oriented (O-O) languages** are popular and widely used
  - \(\Rightarrow\) objects combine data with operations
  - \(\Rightarrow\) clients don’t need to know about internal representation

- **encapsulation** facilitates modular thinking

- reasoning about O-O programs is challenging
  - \(\Rightarrow\) shared mutable state
  - \(\Rightarrow\) inheritance (i.e. subtyping and method overriding)
Shared mutable state

class CONNECTION_POOL

get_connection
Shared mutable state

class CONNECTION_POOL

get_connection

request
Shared mutable state

class CONNECTION_POOL

get_connection

request

connection
Shared mutable state

class CONNECTION_POOL

get_connection

request

connection

encapsulation
Shared mutable state

- **are there free connections?**
  - yes
  - no

- create connection

- encapsulation challenging to check statically
Inheritance

- inheritance allows specialisation and overriding

- determining what a method call actually does is difficult

- lookup scheme relies on dynamic information
  => but we are interested in static reasoning and verification
class CELL
{
    private int val;

    public virtual void set(int x)
    {
        this.val = x;
    }

    public virtual int get()
    {
        return this.val;
    }
}
Inheritance

```csharp
class CELL
{
    private int val;

    public virtual void set(int x)
    {
        this.val = x;
    }

    public virtual int get()
    {
        return this.val;
    }
}

class RECELL: CELL
{
    private int bak;
    public override void set(int x)
    {
        this.bak = base.get();
        base.set(x);
    }
}
```
class CELL
{
    private int val;

    public virtual void set(int x)
    {
        this.val = x;
    }

    public virtual int get()
    {
        return this.val;
    }
}

class RECELL: CELL
{
    private int bak;
    public override void set(int x)
    {
        this.bak = base.get();
        base.set(x);
    }
}

class DCELL: CELL
{
    public override void set(int x)
    {
        base.set(2*x);
    }
}
Inheritance

class CELL
{
    private int val;

    public virtual void set(int x)
    {
        this.val = x;
    }

    public virtual int get()
    {
        return this.val;
    }
}

class RECELL: CELL
{
    private int bak;
    public override void set(int x)
    {
        this.bak = base.get();
        base.set(x);
    }
}

class DCELL: CELL
{
    public override void set(int x)
    {
        base.set(2*x);
    }
}

inheritance is not subtyping!
Motivating separation logic

- let’s first see how far we can go with classical Hoare logic
- consider the method `java.awt.Rectangle.translate(int x, int y)`
Motivating separation logic

- let's first see how far we can go with classical Hoare logic

- consider the method `java.awt.Rectangle.translate(int x, int y)

\[
\begin{align*}
\{ & \text{this.x} = X \land \text{this.y} = Y \\
\text{Rect::translate}(x,y) \\
\{ & \text{this.x} = X + x \land \text{this.y} = Y + y \}
\end{align*}
\]
Motivating separation logic

\{this.x = X \land this.y = Y\}

Rect::translate(x,y)

\{this.x = X + x \land this.y = Y + y\}

d this.x += x;
d this.y += y;
Motivating separation logic

\{\text{this}.x = X \land \text{this}.y = Y\}

\text{Rect::translate}(x, y)

\{\text{this}.x = X + x \land \text{this}.y = Y + y\}

\text{this}.x += x;
\text{this}.y += y;

\text{this}.x += x;
\text{this}.y += y;
\text{this}.h = 0;

\text{this}.x += x;
\text{this}.y += y;
\text{if (this.parent != this) }
\text{this}.parent.x += x;
Motivating separation logic

\{ \text{this.x} = X \land \text{this.y} = Y \} \\
\text{Rect::translate(x,y)} \\
\{ \text{this.x} = X + x \land \text{this.y} = Y + y \} \\
\text{this.x} += x; \\
\text{this.y} += y; \\
\text{this.x} += x; \\
\text{this.y} += y; \\
\text{this.h} = 0; \\
\text{this.x} += x; \\
\text{this.y} += y; \\
\text{if (this.parent} \neq \text{this)} \\
\text{this.parent.x} += x; \\
framing?
Motivating separation logic

• specifying what isn’t modified is tedious:

\[
\{ (z /= \text{this} \lor f \not\in \{x,y\}) \land z.f = \top \} \\
\text{Rect::translate}(x,y) \\
\{ z.f = \top \}
\]

• can we just use modifies clauses?
Motivating separation logic

• specifying what isn’t modified is tedious:

\[
\{(z \neq \text{this} \lor f \notin \{x,y\}) \land z.f = \forall\}
\]

\[
\text{Rect::translate}(x,y)
\]

\[
\{z.f = \forall\}
\]

• can we just use modifies clauses?

\[
\text{Rect::translate}(x,y) \text{ modifies this.x, this.y}
\]
Motivating separation logic

• not when we have complex shapes in memory

• consider the method `System.Collection.SortedList.Clear()`
Motivating separation logic

- not when we have complex shapes in memory
- consider the method `System.Collection.SortedList.Clear()

`SortedList::Clear()` modifies `*.next`
Motivating separation logic

• not when we have complex shapes in memory

• consider the method System.Collection.SortedList.Clear()

  SortedList::Clear() modifies * . next

  - not precise
  - breaks abstraction
  - doesn’t work at all for interfaces
Motivating separation logic

- not when we have complex shapes in memory

- consider the method `System.Collections.SortedList.Clear()`

`SortedList::Clear()` modifies `* .next`

- not precise
- breaks abstraction
- doesn’t work at all for interfaces

if not modifies clauses, then what? => ownership, SL, ...
Motivating separation logic

• **separation logic** makes modifications implicit in the specification
  => “anything not mentioned isn’t changed”

• supports assertions describing only the part of the memory being modified

• “natural” reasoning for O-O programs
  => but need a new memory model
  => need to address encapsulation
  => and need to accommodate and control inheritance
Next on the agenda

(1) motivation and challenges

(2) extending the memory model

(3) simple statements and proof rules

(4) tackling inheritance: abstract predicate families

(5) method specification and verification
Recap: the heaplet model

- **the store**: state of the local variables
  
  Variables $\rightarrow$ Integers

- **the heap**: state of dynamically-allocated objects
  
  Locations $\rightarrow$ Integers

**where**: Locations $\subseteq$ Naturals
Recap: separating conjunction

$s, h \models p \land q$

• informally: the heap $h$ can be divided in two so that $p$ is true of one partition and $q$ of the other
Recap: separating conjunction

\[
\begin{align*}
  s, h & \models p \cdot q \\
  \text{informally: the heap } h \text{ can be divided in two so that } \quad p \text{ is true of one partition and } q \text{ of the other}
\end{align*}
\]

\[s, h \models p \cdot q \text{ if } \exists h_1, h_2. (h_1 \perp h_2), (h_1 \circ h_2 = h), s, h_1 \models p \text{ and } s, h_2 \models q\]
Recap: example store and heap

- **Store**
  - $x:$
  - $y:$
  - $z:$
  - $\text{temp}: 3$

- **Heap**
  - $1$
  - $16$
Recap: example store and heap

\[ x \mapsto 1 * y \mapsto z * z \mapsto 16 \land temp = 3 \]
Extending the memory model

\[ s, h, d \models p \]

- must now accommodate objects and dynamic types
Extending the memory model

\[ s, \ h, \ d \models p \]

- must now accommodate objects and dynamic types

\[ s : \text{Variables} \rightarrow \text{ObjectIDs} \cup \text{Integers} \]
Extending the memory model

\[ s, h, d \models p \]

- must now accommodate objects and dynamic types

\[ s : \text{Variables} \rightarrow \text{ObjectIDs} \cup \text{Integers} \]

\[ h : \text{ObjectIDs} \times \text{FieldNames} \rightarrow \text{ObjectIDs} \cup \text{Integers} \]
Extending the memory model

\[ s, h, d \models \rho \]

- must now accommodate objects and dynamic types

\[ s : \text{Variables} \rightarrow \text{ObjectIDs} \cup \text{Integers} \]

\[ h : \text{ObjectIDs} \times \text{FieldNames} \rightarrow \text{ObjectIDs} \cup \text{Integers} \]

\[ d : \text{ObjectIDs} \rightarrow \text{ClassNames} \]
A partial semantics

- Let $se$ denote a “simple expression”, i.e. one that does not access any fields or methods.

\[
s, h, d \models se_1.f \mapsto se_2
\]

\[
s, h, d \models se : C
\]

\[
s, h, d \models se_1 = se_2
\]

\[
s, h, d \models p \ast q
\]
A partial semantics

- let $se$ denote a “simple expression”, i.e. one that does not access any fields or methods

\[
s, h, d \models se_1. f \mapsto se_2 \quad \text{if} \quad h([|se_1|]s, f) = [|se_2|]s
\]

\[
s, h, d \models se : C
\]

\[
s, h, d \models se_1 = se_2
\]

\[
s, h, d \models p \ast q
\]
A partial semantics

- let \( \text{se} \) denote a “simple expression”, i.e. one that does not access any fields or methods

\[
\begin{align*}
  s, h, d & \models \text{se}_1.f \mapsto \text{se}_2 \\
  s, h, d & \models \text{se} : C \\
  s, h, d & \models \text{se}_1 = \text{se}_2 \\
  s, h, d & \models p * q
\end{align*}
\]

\[
\text{if } h([|\text{se}_1|]s, f) = [|\text{se}_2|]s
\]

- different to last week!
- contains “at least” (not “exactly”)
A partial semantics

- let \( se \) denote a “simple expression”, i.e. one that does not access any fields or methods

\[
\begin{align*}
& s, h, d \models se_1.f \mapsto se_2 \quad \text{if} \quad h([se_1]s, f) = [se_2]s \\
& s, h, d \models se : C \quad \text{if} \quad d([se]s) = C \\
& s, h, d \models se_1 = se_2 \\
& s, h, d \models p * q
\end{align*}
\]
A partial semantics

- let $se$ denote a “simple expression”, i.e. one that does not access any fields or methods

$$s, h, d \models se_1.f \mapsto se_2 \quad \text{if} \quad h([|se_1|]s, f) = [|se_2|]s$$

$$s, h, d \models se : C \quad \text{if} \quad d([|se|]s) = C$$

$$s, h, d \models se_1 = se_2 \quad \text{if} \quad [|se_1|]s = [|se_2|]s$$

$$s, h, d \models p \ast q$$
A partial semantics

- let \( se \) denote a “simple expression”, i.e. one that does not access any fields or methods

\[
\begin{align*}
    s, h, d & \models se_1.f \mapsto se_2 & \text{if } h([|se_1|]s, f) = [|se_2|]s \\
    s, h, d & \models se : C & \text{if } d([|se|]s) = C \\
    s, h, d & \models se_1 = se_2 & \text{if } [|se_1|]s = [|se_2|]s \\
    s, h, d & \models p * q & \text{if } \exists h_1, h_2. \ h_1 \perp h_2 \land h = h_1 \cup h_2 \\
    & & \land s, h_1, d \models p \land s, h_2, d \models q
\end{align*}
\]
Separating conjunction example

• what kind of heap would satisfy the following?

\[ s, h, d \models x_1.f \leftrightarrow y \ast x_2.f \leftrightarrow y \]
Separating conjunction example

• what kind of heap would satisfy the following?

\[ s, h, d \models x_1.f \leftrightarrow y \wedge x_2.f \leftrightarrow y \]

• ...and what about this?

\[ s, h, d \models x_1.f \leftrightarrow y \land x_2.f \leftrightarrow y \]
Next on the agenda

(1) motivation and challenges

(2) extending the memory model

(3) simple statements and proof rules

(4) tackling inheritance: abstract predicate families

(5) method specification and verification
Simple instructions and proof rules

• we start by building a separation logic for a simple object-oriented language
  => field mutation, field lookup, ...

• postpone method specification and verification
  => avoid the complexity of method dispatch until later

• reminder: tight interpretation of triples

\[ \models \{ pre \} \ P \ \{ post \} \]
Field mutation and field lookup

\[\vdash \{ x.f \mapsto _\_ \} \ x.f := y \ \{ \ \} \]

\[\vdash \{ x.f \mapsto e \} \ y := x.f \ \{ \ \} \]
Field mutation and field lookup

\[ \vdash \{ x.f \mapsto _- \} x.f := y \ {\{ x.f \mapsto \ y \} } \]

\[ \vdash \{ x.f \mapsto e \} y := x.f \ {\} \]
Field mutation and field lookup

\[
\vdash \{x.f \rightarrow _\} \ x.f := y \ \{x.f \mid\rightarrow y\} \\
\vdash \{x.f \rightarrow e\} \ y := x.f \ \{x.f \mid\rightarrow e \land y = e\}
\]

provided \(y \not= x\) and \(y\) not free in \(e\)
Field mutation and field lookup

\[ \vdash \{ x.f \mapsto \_ \} \ x.f := y \ \{ x.f \mid \mapsto y \} \]

\[ \vdash \{ x.f \mapsto e \} \ y := x.f \ \{ x.f \mid \mapsto e \land y = e \} \]

provided \( y \neq x \) and \( y \) not free in \( e \)

and if not...?
Structural rules

\[
\frac{\{p\} \quad C \quad \{q\}}{\{p \ast r\} \quad C \quad \{q \ast r\}}
\]

provided \( \text{modifies}(C) \cap \text{fv}(r) = \{\}\)

\[
\frac{\{p\} \quad C \quad \{q\}}{\{\exists v. p\} \quad C \quad \{\exists v. q\}}
\]

provided \( v \) not free in \( C \)
Simple proof example

• consider the statement $x := x.next$
  $\Rightarrow$ e.g. from a linked list class

• verify the following triple:

$$\{x.next \mapsto \_ \ast x = y\} \ x := x.next \ \{y.next \mapsto x\}$$
Simple proof example

\{ x.next \mid\rightarrow \_ \ast x = y \}

\[
x := x.next;
\]
Simple proof example

\{x.next |-> _ * x = y\}

\{\exists n, x^{old}. x.next |-> n * x = x^{old} /\ y = x^{old}\}

\[x := x.next;\]
Simple proof example

\{ x.next |-> _ * x = y \}
\{ \exists n, x^{old}. x.next |-> n * x = x^{old} /\ y = x^{old} \}
\{ x.next |-> n * x = x^{old} /\ y = x^{old} \}

x := x.next;
Simple proof example

\{ x.\text{next} \mid\rightarrow \_ \ast x = y \} \\
\{ \exists n, x^{\text{old}}. x.\text{next} \mid\rightarrow n \ast x = x^{\text{old}} \land y = x^{\text{old}} \} \\
\{ x.\text{next} \mid\rightarrow n \ast x = x^{\text{old}} \land y = x^{\text{old}} \} \\
\{ x := x.\text{next}; \} \\
\{ x^{\text{old}}.\text{next} \mid\rightarrow n \ast x = n \land y = x^{\text{old}} \}
Simple proof example

\{x'.next |-> _ \ast x = y\}
\{\exists n, x^{old}. x'.next |-> n \ast x = x^{old} \land y = x^{old}\}
\{x'.next |-> n \ast x = x^{old} \land y = x^{old}\}

x := x'.next;

\{x^{old}.next |-> n \ast x = n \land y = x^{old}\}
\{\exists n, x^{old}. x^{old}.next |-> n \ast x = n \land y = x^{old}\}
Simple proof example

\[
\begin{align*}
\{&x.next |\rightarrow \_ \ast x = y \} \\
\exists n, x^{\text{old}}. &x.next |\rightarrow n \ast x = x^{\text{old}} \land y = x^{\text{old}} \\
\{&x.next |\rightarrow n \ast x = x^{\text{old}} \land y = x^{\text{old}} \} \\
&x := x.next; \\
\{&x^{\text{old}}.next |\rightarrow n \ast x = n \land y = x^{\text{old}} \} \\
\exists n, x^{\text{old}}. &x^{\text{old}}.next |\rightarrow n \ast x = n \land y = x^{\text{old}} \\
\{&y.next |\rightarrow x \}
\end{align*}
\]
Next on the agenda

(1) motivation and challenges

(2) extending the memory model

(3) simple statements and proof rules

(4) tackling inheritance: abstract predicate families

(5) method specification and verification
Recap: Cell, ReCell, and DCell

class CELL
{
    private int val;

    public virtual void set(int x)
    {
        this.val = x;
    }

    public virtual int get()
    {
        return this.val;
    }
}

class RECELL: CELL
{
    private int bak;
    public override void set(int x)
    {
        this.bak = base.get();
        base.set(x);
    }
}

class DCELL: CELL
{
    public override void set(int x)
    {
        base.set(2*x);
    }
}
```plaintext
{ ??? }
x.set(5)
{ ??? }
y := x.get + 5
{ ??? }
```
```plaintext
{ ??? }
x.set(5)
{ ??? }
y := x.get + 5
{ ??? }
```

```plaintext
def set
```
```plaintext
def get
```
```plaintext
val 5
```
{ x.val |-> _ }
x.set(5)
{ x.val |-> 5 }
y := x.get + 5
{ ??? }

breaks abstraction!
\[
\{ \text{x.val} \mapsto _\}\ 
\text{x.set(5)}
\{ \text{x.val} \mapsto 5 \}
\text{y := x.get + 5}
\{ ??? \}
\]

*breaks abstraction!*
\{
  \texttt{x.val} \rightarrow \_ \}
\texttt{x.set}(5)
\{
  \texttt{x.val} \rightarrow 5
\}
\texttt{y := x.get + 5}
\{
  \\ ???? \\
\}

\textit{breaks abstraction!}

\textit{need to be able to reason abstractly on this side...}

\ldots \textit{and concretely (e.g. x.val \rightarrow 5) on this side}
The boundary of abstraction

- there is a need for data-centred abstractions in our reasoning system

- reason, on the client side, about encapsulated state abstractly

- need to cope with inheritance and dynamic dispatch
Abstract predicates (Ap)

- annotate classes with **abstract predicate (Ap)** definitions

- an Ap consists of a **name**, **definition**, and **scope**
  => *for simplicity, scope here is a single class*

- within the scope, can freely change between the name and definition

- outside the scope, can **only use the name**
class CELL {

// Ap definitions
define x.\text{Val}_\text{Cell}(n) \text{ as } x.\text{val} \rightarrow n

// field declarations
private \text{int} \text{ val;}

// methods (i.e. set, get)
...
Abstract predicate example

class CELL
{
    // Ap definitions
    define x.\text{Val}_{\text{Cell}}(n) \text{ as } x.\text{val} \mapsto n

    // field declarations
    private int val;

    // methods (i.e. set, get)
    ...
}

name?
definition?
scope?
x.\text{Val}_{\text{Cell}}(n)
x.\text{val} \mapsto n
CELL
Abstract predicate example

class CELL
{
  // Ap definitions
  define x.ValCELL(n) as x.val |-> n

  // field declarations
  private int val;

  // methods (i.e. set, get)
  ...
}

name?  x.ValCELL(n)

definition?  x.val |-> n

scope?  CELL

{ ??? }

x.set(5)

{ ??? }

y := x.get + 5

{ ??? }
class CELL
{
    // Ap definitions
    define x.\text{Val}_{\text{Cell}}(n) \textbf{as} x.\text{val} \mapsto n

    // field declarations
    private int val;

    // methods (i.e. set, get)
    ...
}

- how do we prove \{ x.\text{Val}_{\text{Cell}}(\_)) \} x.set(5) \{ x.\text{Val}_{\text{Cell}}(5) \} ?
- what if \text{d}(x) = \text{ReCell} ?
Abstract predicate families (Apfs)

• different (sub)classes can have different Ap definitions

• abstract predicate families (Apfs) provide different definitions, or “entries”, based on dynamic type information => “dynamically dispatched predicates”

• annotate classes with different Apf entries
Abstract predicate families (Apfs)

- different (sub)classes can have different Ap definitions

- abstract predicate families (Apfs) provide different definitions, or “entries”, based on dynamic type information
  => “dynamically dispatched predicates”

- annotate classes with different Apf entries

\[
x \cdot \text{Val} \xrightarrow{d(x) = \text{Cell}} x \cdot \text{Val}_{\text{Cell}} \\
x \cdot \text{Val} \xrightarrow{d(x) = \text{ReCell}} x \cdot \text{Val}_{\text{ReCell}} \\
x \cdot \text{Val} \xrightarrow{d(x) = \text{DCell}} x \cdot \text{Val}_{\text{DCell}}
\]
Abstract predicate family example

class CELL
{
    // Apf definitions
    define x.\text{Val}_{\text{Cell}}(n) \text{ as } x.\text{val} |-> n

    // field declarations
    private int val;

    // methods (i.e. set, get)
    ...
}

class RECELL: CELL
{
    // Apf definitions
    ???

    // field declarations
    private int bak;

    // methods (i.e. override set)
    ...
}
Abstract predicate family example

```java
class CELL {
    // Apf definitions
    define x.Val\_Cell(n) as x.val |-> n

    // field declarations
    private int val;

    // methods (i.e. set, get)
    ...
}

class RECELL: CELL {
    // Apf definitions
    define x.Val\_Recell(n,b)
    as x.Val\_Cell(n) * x.bak |-> b

    // field declarations
    private int bak;

    // methods (i.e. override set)
    ...
}
```
ReCell adds an argument to the Apf Val

=> in the scope of ReCell, \( \forall x,n: x.Val(n) \iff x.Val(n,\_ ) \)
Next on the agenda

(1) motivation and challenges

(2) extending the memory model

(3) simple statements and proof rules

(4) tackling inheritance: abstract predicate families

(5) method specification and verification
Static vs. dynamic specifications

- two types of method calls in O-O languages
  => *statically* dispatched, e.g. `y.Cell::get()`
  => *dynamically* dispatched, e.g. `x.m(a)`

- annotate methods with *static and dynamic specifications*

  describes in detail what the body does

  more abstract: "idea" behind the method that subclasses must respect
Example

class CELL
{
  define x.\text{Val}_{\text{Cell}}(n) \ as \ x.\text{val} \ |\to \ n

  private int val;

  public virtual void set(int x)
  dynamic \{
    \_ \{
      ???
    \}
  \}
  static \{
    ???
  \}
  \{
    this.val = x;
  \}

  public virtual int get()
  dynamic \{
    \_ \{
      ???
    \}
  \}
  static \{
    ???
  \}
  \{
    return this.val;
  \}
}
class CELL
{
    define x.ValCell(n) as x.val |-> n

    private int val;

    public virtual void set(int x)
    dynamic { this.Val(_) } _ { this.Val(x) }
    static { ??? } _ { ??? }
    { this.val = x; }

    public virtual int get()
    dynamic { ??? } _ { ??? }
    static { ??? } _ { ??? }
    { return this.val; }
}
Example

class CELL
{
  define x.ValCell(n) as x.val |-> n

  private int val;

  public virtual void set(int x)
  dynamic { this.Val(_) } _ { this.Val(x) }
  static { this.ValCell(_) } _ { this.ValCell(x) }
  { this.val = x; }

  public virtual int get()
  dynamic { ??? } _ { ??? }
  static { ??? } _ { ??? }
  { return this.val; }
}
class CELL
{
    define x.Val\_Cell(n) as x.val |-> n

    private int val;

    public virtual void set(int x)
    dynamic { this.Val(_) } _ { this.Val(x) }  
    static { this.Val\_Cell(_) } _ { this.Val\_Cell(x) }  
    { this.val = x; }

    public virtual int get()
    dynamic { this.Val(v) } _ { this.Val(v) \&\& Res = v } 
    static { ??? } _ { ??? }  
    { return this.val; }
}
Example

```csharp
class CELL
{
    define x.Val_Cell(n) as x.val |-> n

    private int val;

    public virtual void set(int x)
    dynamic { this.Val(_) } _ { this.Val(x) }
    static { this.Val_Cell(_) } _ { this.ValCell(x) }
    { this.val = x; }

    public virtual int get()
    dynamic { this.Val(v) } _ { this.Val(v) \ Res = v }
    static { this.Val_Cell(v) } _ { this.ValCell(v) \ Res = v }
    { return this.val; }
}

prove!
{
    true
    x := new Cell(3)
    y := new Cell(4)
    x.set(5)
    n := y.get()
    { x.Val(5) * y.Val(4) \ n=4 }
```
Verifying a newly introduced method

- two proof obligations

- first, verify the method body against the static specification
  \[ \Rightarrow \text{e.g. } \{\text{this}.\text{ValCell}(\_)} \text{ this}.\text{val} := x \{\text{this}.\text{ValCell}(x)} \]
  \[ \Rightarrow \text{we are now “in scope” and can use the definition of } \text{ValCell} \]

- second, check the consistency of the static and dynamic specifications
  \[ \Rightarrow \text{e.g. } \{\text{this}.\text{ValCell}(\_)} _ {\{\text{this}.\text{ValCell}(x)} \]
  \[ \text{implies} \]
  \[ \{\text{this:Cell} * \text{this}.\text{Val}(\_)} _ {\{\text{this}.\text{Val}(x)} \]
class RECELL: CELL
{
    // Apf definitions
    define x.ValReCell(n,b) as x.ValCell(n) * x.bak |-> b

    private int bak;

    public override void set(int x)
    dynamic { ??? } _ { ??? }
    static { ??? } _ { ??? }
    { this.bak = base.get(); base.set(x); }

    inherit get()
    dynamic { ??? } _ { ??? }
    static { ??? } _ { ??? }
}
class RECELL: CELL
{
    // Apf definitions
    define x.Val_{Recell}(n,b) as x.Val_{Cell}(n) * x.bak |-> b

    private int bak;

    public override void set(int x)
    dynamic { this.Val(v,_) } _ { this.Val(x,v) }
    static { ??? } _ { ??? }
    { this.bak = base.get(); base.set(x); }

    inherit get()
    dynamic { ??? } _ { ??? }
    static { ??? } _ { ??? }
}
Subclassing

class RECELL: CELL
{
    // Apf definitions
    define x.Val_{Recell}(n,b) as x.Val_{Cell}(n) * x.bak |-> b

    private int bak;

    public override void set(int x)
    dynamic { this.Val(v,__) } __ { this.Val(x,v) }
    static { this.Val_{ReCell}(v,__) } __ { this.Val_{ReCell}(x,v) }
    { this.bak = base.get(); base.set(x); }

    inherit get()
    dynamic { ??? } __ { ??? }
    static { ??? } __ { ??? }
}
Subclassing

class RECELL: CELL
{
    // Apf definitions
    define x.Val_{Recell}(n,b) as x.Val_{Cell}(n) * x.bak |-> b

    private int bak;

    public override void set(int x)
    dynamic { this.Val(v,__) } __ { this.Val(x,v) }
    static { this.Val_{Recell}(v,__) } __ { this.Val_{Recell}(x,v) }
    { this.bak = base.get(); base.set(x); }

    inherit get()
    dynamic { this.Val(v,b) } __ { this.Val(v,b) \ Res = v }
    static { ??? } __ { ??? }
}
class RECELL: CELL
{
    // Apf definitions
    define x.Val_{ReCell}(n,b) as x.Val_{Cell}(n) * x.bak |-> b

    private int bak;

    public override void set(int x)
    dynamic { this.Val(v,__) } __ { this.Val(x,v) }
    static { this.Val_{ReCell}(v,__) } __ { this.Val_{ReCell}(x,v) }
    { this.bak = base.get(); base.set(x); }

    inherit get()
    dynamic { this.Val(v,b) } __ { this.Val(v,b) \ Res = v }
    static { this.Val_{ReCell}(v,b) } __ { this.Val_{ReCell}(v,b) \ Res = v }
}
Verifying an **overridden** method (e.g. set)

- three proof obligations

- (1) body verification; (2) consistency checking; and

- (3) verify that the **dynamic** specification is **stronger** than the one in the **parent** class
Verifying an \textit{inherited} method (e.g. get)

- three proof obligations
- (1) body verification; (2) consistency checking; and
- (3) verify that the \textit{static} specification follows from the one in the \textit{parent} class
And what about this?

class DCELL: CELL
{
    public override void set(int x)
    {
        base.set(2*x);
    }
}
And what about this?

class DCELL: CELL
{
    define ??? as ???

    public override void set(int x)
    dynamic { ??? } _ { ??? }
    static { ??? } _ { ??? }
    { base.set(2*x); }

    public inherit get()
    dynamic { ??? } _ { ??? }
    static { ??? } _ { ??? }
}
class DCELL: CELL
{
    define x.Val_DCell(n) as false
    define x.DVal_DCell(n) as x.Val_Cell(n)

    public override void set(int x)
    dynamic { this.Val(_)} _ { ??? }
    also { this.DVal(_)} _ { ??? }
    static { ??? } _ { ??? }
{ base.set(2*x); }

    public inherit get()
    dynamic { this.Val(v)} _ { ??? }
    also { this.DVal(v)} _ { ??? }
    static { ??? } _ { ??? }
}
And what about this?

```java
class DCELL: CELL
{
    define x.ValDCell(n) as false
    define x.DValDCell(n) as x.ValCell(n)

    public override void set(int x)
    dynamic { this.Val(_) } _ { this.Val(x) }
    also { this.DVal(_) } _ { this.DVal(2*x) }
    static { ??? } _ { ??? }
    { base.set(2*x); }

    public inherit get()
    dynamic { this.Val(v) } _ { ??? }
    also { this.DVal(v) } _ { ??? }
    static { ??? } _ { ??? }
}
```

idea: ensure that no client will ever have a Val predicate for a DCell object
class DCELL: CELL
{
    define x.ValDCell(n) as false
    define x.DValDCell(n) as x.ValCell(n)

    public override void set(int x)
    dynamic { this.Val(_} _ { this.Val(x) }
    also { this.DVal(_} _ { this.DVal(2*x) }
    static { this.DValDCell(_} _ { this.DValDCell(2*x) }
    { base.set(2*x); }

    public inherit get()
    dynamic { this.Val(v} _ { ??? }
    also { this.DVal(v} _ { ??? }
    static { ??? } _ { ??? }
}
class DCELL: CELL
{
    define x.Val_{DCELL}(n) as false
    define x.DVal_{DCELL}(n) as x.Val_{CELL}(n)

    public override void set(int x)
    dynamic { this.Val(_) } _ { this.Val(x) }
    also { this.DVal(_) } _ { this.DVal(2*x) }
    static { this.DVal_{DCELL}(_) } _ { this.DVal_{DCELL}(2*x) }
    { base.set(2*x); }

    public inherit get()
    dynamic { this.Val(v) } _ { this.Val(v) \&\& Res = v }
    also { this.DVal(v) } _ { this.DVal(v) \&\& Res = v }
    static { ??? } _ { ??? }
}

idea: ensure that no client will ever have a Val predicate for a DCell object
And what about this?

```csharp
class DCELL: CELL
{
    define x.ValDCell(n) as false
    define x.DValDCell(n) as x.ValCell(n)

    public override void set(int x)
    dynamic { this.Val(__) } __ { this.Val(x) }
    also { this.DVal(__) } __ { this.DVal(2*x) }
    static { this.DValDCell(__) } __ { this.DValDCell(2*x) }
    { base.set(2*x); }

    public inherit get()
    dynamic { this.Val(v) } __ { this.Val(v) \ Res = v }
    also { this.DVal(v) } __ { this.DVal(v) \ Res = v }
    static { this.DValDCell(v) } __ { this.DValDCell(v) \ Res = v }
}
```

idea: ensure that no client will ever have a Val predicate for a DCell object
Next on the agenda

(1) motivation and challenges

(2) extending the memory model

(3) simple statements and proof rules

(4) tackling inheritance: abstract predicate families

(5) method specification and verification
Conclusion

• separation logic, for reasoning about shared mutable state, can be extended to object-oriented programs

• memory model extended to support objects and dynamic type information

• inheritance tackled with Apfs and static/dynamic specs

• implemented (e.g. jStar, VeriFast); can verify common design patterns

• only just the basics! See the papers for the full story
Main sources for these lectures

Parkinson and Bierman: *Separation Logic, Abstraction and Inheritance*. In: POPL 2008

Thank you! Questions?

Next unit:

- testing (with Bertrand Meyer and guest lecturers)