Separation logic for OO

Stephan van Staden

Introduction

OO languages are popular and widely used

We need to reason about OO programs

Some (problematic) features of OO languages:

- Shared mutable state
- Inheritance (subtyping and overriding)
  - Determining what a method call does is difficult! Complicated method lookup scheme that relies on dynamic info. We are interested in static verification...

Reasoning about OO programs

We can use separation logic to reason about shared mutable state

But it is not enough! We need to accommodate and control inheritance

(Many published OO proof systems cannot reason about simple programming patterns, or support them in a very complicated way)

We will look at a state of the art separation logic for OO by Parkinson and Bierman ("Separation Logic, Abstraction and Inheritance", proceedings POPL 2008)

Outline

1. Shared mutable state
    1. Memory model
    2. Simple statements & proof rules

2. Inheritance
    1. Abstract predicate families
    2. Method specification and verification

1. Shared mutable state

1.1 OO memory model

State =def Stack x DType x Heap
Stack =def Var -> Value
Value =def Ref ∪ Int ∪ ...
DType =def Ref $\to_{fin}$ Type (dynamic type info)
Heap =def Field $\to_{fin}$ Value (location granularity = field)
Field =def Ref x Attributename

$(S, D, H) \vDash e.f \mapsto e'$ =def $H([e]_S, f) = [e']_S$ (different!)
$(S, D, H) \vDash e : C$ =def $D([e]_S) = C$
$(S, D, H) \vDash e = e'$ =def $[e]_S = [e']_S$
$(S, D, H) \vDash P * Q$ =def $\exists H_1, H_2 .\ H_1 \perp H_2,\ H = H_1 \cup H_2,$
$(S, D, H_1) \vDash P,\ (S, D, H_2) \vDash Q$

1.2 Simple instructions & proof rules

- Field mutation
  $\{x.f \mapsto \_\}\ x.f := y\ \{x.f \mapsto y\}$
- Field lookup
  $\{x.f \mapsto e\}\ y := x.f\ \{x.f \mapsto e * y = e\}$, provided y is not the same as x, and y does not occur free in e

Rules for variable assignment, sequential composition, conditional, loop, etc. are the familiar ones

Later: method calls (a bit complicated due to inheritance)

Structural rules

- Frame:

$$\frac{\{P\}\ s\ \{Q\}}{\{P * R\}\ s\ \{Q * R\}}$$

provided $\text{modifies}(s) \cap \text{FV}(R) = \{\}$

Note: $\text{modifies}(x.f := y) = \{\}$

- Auxiliary Variable Elimination:

$$\frac{\{P\}\ s\ \{Q\}}{\{\exists v.\ P\}\ s\ \{\exists v.\ Q\}}$$

provided v does not occur free in s

- Consequence, ...

2. Inheritance

2.1 Abstract predicate families (apfs)

OO programming is based on encapsulated data-centered abstractions

Parkinson and Bierman's system embraces this, resulting in simple reasoning that accommodates OO (esp. inheritance) well

Apfs are heavily used in method specs

Abstract predicates

Abstract predicate: name, definition and scope.

Within the scope, one can freely change between the name and definition. Outside the scope, one can use the name only atomically

Examples: list, tree, etc.

Since sep logic predicates describe data, abstract predicates are encapsulated data abstractions. Fit OO remarkably well!

For simplicity: single class as the scope. Think “interface” and “implementation” of a predicate. Clients use interfaces

The abstraction boundary is a class, but the abstractions themselves are not dictated by classes

Examples:

- An abstract predicate List can be implemented with Node objects
- class List can also implement a Stack abstract predicate
- class Student can implement a Person predicate

The “family” part of apfs

Different classes (and in particular subclasses) can implement abstractions differently

They can provide different definitions, or *entries*, for the same predicate, hence predicate *family.*

The definition that applies is based on the dynamic type of the first predicate argument. Apfs as “dynamically dispatched predicates”

Example: class Cell defines $x.Val_{Cell}(n)$ as $x.val \mapsto n$.

- Val is an apf
- Val_{Cell} is class Cell’s entry of the apf Val

class Cell {
  // apf definitions
  define x.Val_{Cell}(n) as x.val ↦ n
  // field declarations
  val: int
  // methods e.g. get, set
}

Within the scope of class Cell *only*, we can use the assumptions (in the rule of consequence when verifying the methods of Cell):

- FtoE: ∀x,n . x : Cell => [x.Val(n) <=> x.Val_{Cell}(n)]
- EtoD: ∀x,n . x.Val_{Cell}(n) <=> x.val ↦ n
- Arity reduction: ∀x . x.Val() <=> x.Val(_)

Arity reduction allows subclasses to add arguments to apfs

```
class Cell {
   // apf definitions
   define x.Val_Cell(n) as x.val ↦ n
   // field declarations
   val: int
   // methods   e.g. get, set
}
```

```
class ReCell  inherit Cell {
   // apf definitions
   define x.Val_ReCell(n, b) as x.Val_Cell(n) * x.bak ↦ b
   // field declarations: a backup value
   bak: int
   // methods: new, overridden, ...
}
```

- ReCell does not know the definition of $x.Val_{Cell}(n)$, yet it defines its entry of apf Val in terms of it
- ReCell adds an argument to the apf Val. In the scope of ReCell:
  $\forall x$. x.Val() <=> x.Val(_)
  $\forall x,n$. x.Val(n) <=> x.Val(n, _)

2.2 Method specification & verification

Static & dynamic specs

Two types of method calls in OO languages:

- Statically dispatched/bound calls
  Examples: *super* calls in Java, x.C::m(a) in C++
- Dynamically dispatched/bound calls
  Example: x.m(a)

Specify each method with a static and a dynamic spec

- Use static spec to verify statically dispatched calls.
  Static spec describes in detail what the body does
- Use dynamic spec to verify dynamically dispatched calls.
  Dynamic spec is more abstract – it gives the idea behind the method that all subclasses must respect

Example

```
class Cell {
   // apf definitions
   define x.Val_Cell(n) as x.val ↦ n
   // field declarations
   val: int

   // methods
   introduce set(x: int)
   dynamic {this.Val(_)}_{this.Val(x)}
   static {this.Val_Cell(_)}_{this.Val_Cell(x)}
      { this.val := x }

   introduce get(): int
   dynamic {this.Val(v)}_{this.Val(v) * Res = v}
   static {this.Val_Cell(v)}_{this.Val_Cell(v) * Res = v}
      { Res := this.val }
}
```

Client reasoning

```
class Cell {
   // apf definitions
   define x.Val_Cell(n) as x.val ↦ n
   // field declarations
   val: int

   // methods
   introduce set(x: int)
   dynamic {this.Val(_)}_{this.Val(x)}
   static {this.Val_Cell(_)}_{this.Val_Cell(x)}
      { this.val := x }

   introduce get(): int
   dynamic {this.Val(v)}_{this.Val(v) * Res = v}
   static {this.Val_Cell(v)}_{this.Val_Cell(v) * Res = v}
      { Res := this.val }
}
```

Assume the constructor Cell(x: int)
has dynamic spec:
{true}_{this.Val(x)}

{true}
x := new Cell(3)
{x.Val(3)}
y := new Cell(4)
{x.Val(3) * y.Val(4)}
x.set(5)
{x.Val(5) * y.Val(4)}
n := y.get()
{x.Val(5) * y.Val(4) * n=4}
m := x.val
{???}

Verifying a newly introduced method

Two proof obligations:

1. Body verification
   Verify that the body satisfies the static spec, using the apf assumptions of the containing class and all method specs
   {this.Val_{Cell}(_)}
   {this.val ↦ _}
   this.val := x
   {this.val ↦ x}
   {this.Val_{Cell}(x)}

define x.Val_{Cell}(n) **as** x.val ↦ n
// methods
introduce set(x: int)
dynamic {this.Val(_)}_{this.Val(x)}
static {this.Val_{Cell}(_)}_{this.Val_{Cell}(x)}
{ this.val := x }

2. Dynamic dispatch
   Verify the consistency of the static and dynamic specs.
   In particular, check under the apf assumptions that the dynamic spec with "this : Cell" added to the precondition follows from the static spec.
   {this.Val_{Cell}(_)}_{this.Val_{Cell}(x)} ==> {this : Cell * this.Val(_)}_{this.Val(x)}

Specification refinement

{P}_{Q} ==> {P'}_{Q'} also means that:

- the specification {P}_{Q} is stronger than {P'}_{Q'}
- whenever a statement satisfies {P}_{Q}, it must also satisfy {P'}_{Q'}

A proof of {P}_{Q} ==> {P'}_{Q'} uses the structural rules of sep logic (Frame, AuxVarElim, Consequence, ...) to establish {P'}_{Q'} under the assumption {P}_{Q}

Remember the apf assumption of class Cell:
$\forall x,n \,.\, x : Cell \Rightarrow [x.Val(n) <=> x.Val_{Cell}(n)]$

For example,

$\{this.Val_{Cell}(\_)\}\_\{this.Val_{Cell}(x)\}$ ==> {this : Cell * this.Val(_)}_{this.Val(x)}

Proof:

Assumption	$\{this.Val_{Cell}(\_)\}\_\{this.Val_{Cell}(x)\}$
Frame rule	$\{this : Cell * this.Val_{Cell}(\_)\}\_\{this : Cell * this.Val_{Cell}(x)\}$
Consequence	{this : Cell * this.Val(_)}_{this : Cell * this.Val(x)}
Consequence	{this : Cell * this.Val(_)}_{this.Val(x)}

Subclassing

```
class Cell {
   // apf definitions
   define x.Val_Cell(n) as x.val ↦ n
   // field declarations
   val: int

   // methods
   introduce set(x: int)
   dynamic {this.Val(_)}_{this.Val(x)}
   static {this.Val_Cell(_)}_{this.Val_Cell(x)}
      { this.val := x }

   introduce get(): int
   dynamic {this.Val(v)}_{this.Val(v) * Res = v}
   static {this.Val_Cell(v)}_{this.Val_Cell(v) * Res = v}
      { Res := this.val }
}
```

```
class ReCell inherit Cell {
   // apf definitions
   define x.Val_ReCell(n, b) as x.Val_Cell(n) * x.bak ↦ b
   // field declarations: a backup value
   bak: int

   // methods
   override set(x: int)
   dynamic {this.Val(v, _)}_{this.Val(x, v)}
   static {this.Val_ReCell(v, _)}_{this.Val_ReCell(x, v)}
      { local t: int
        t := this.Cell::get();  this.Cell::set(x); this.bak := t }

   inherit get(): int
   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

The three proof obligations use the apf assumptions of the child class:

```
class ReCell inherit Cell {
...
   override set(x: int)
   dynamic {this.Val(v, _)}_{this.Val(x, v)}
   static {this.Val_ReCell(v, _)}_{this.Val_ReCell(x, v)}
     { local t: int
       t := this.Cell::get();  this.Cell::set(x); this.bak := t }
...
}
```

1. Body verification

2. Dynamic dispatch

3. Behavioural subtyping
   Verify that the dynamic spec of the method in the child class is stronger than the one in the parent class

   Example: {this.Val(v, _)}_{this.Val(x, v)} ==> {this.Val(_)}_{this.Val(x)}
   Proof:
   Assumption: {this.Val(v, _)}_{this.Val(x, v)}
   AuxVarElim: {∃v. this.Val(v, _)}_{∃v. this.Val(x, v)}
   Consequence: {this.Val(_, _)}_{this.Val(x, _)}
   Consequence: {this.Val(_)}_{this.Val(x)}

Remember the apf assumption of class ReCell:
∀x,n . x.Val(n) <=> x.Val(n, _)

Verifying an inherited method

The three proof obligations use the apf assumptions of the child class:

```
class ReCell inherit Cell {
 define x.Val_ReCell(n, b) as x.Val_Cell(n) * x.bak ↦ b
...
   inherit get(): int
   dynamic {this.Val(v, b)}_{this.Val(v, b) * Res = v}
   static {this.Val_ReCell(v, b)}_{this.Val_ReCell(v, b) * Res = v}
}
```

1. Behavioural subtyping

2. Dynamic dispatch

3. Inheritance
   Verify that the static specification of the method in the child class follows from the one in the parent class

   Example: $\{this.Val_{Cell}(v)\}\_\{this.Val_{Cell}(v) * Res = v\}$ ==>
   $\{this.Val_{ReCell}(v, b)\}\_\{this.Val_{ReCell}(v, b) * Res = v\}$

   Proof:
   Assumption: $\{this.Val_{Cell}(v)\}\_\{this.Val_{Cell}(v) * Res = v\}$
   Frame: $\{this.Val_{Cell}(v) * this.bak \mapsto b\}\_\{this.Val_{Cell}(v) * Res = v * this.bak \mapsto b\}$
   Consequence: $\{this.Val_{ReCell}(v, b)\}\_\{this.Val_{ReCell}(v, b) * Res = v\}$

Static/dynamic specs - reflection

Only dynamic specs are involved in behavioural subtyping

Apfs are a great enabler of behavioural subtypes

With static specs, child classes never need to see the code of parents. Good for modularity

Copy-and-paste inheritance

```
class Cell {
   // apf definitions
   define x.Val_Cell(n) as x.val ↦ n
   // field declarations
   val: int

   // methods
   introduce set(x: int)
   dynamic {this.Val(_)}_{this.Val(x)}
   static {this.Val_Cell(_)}_{this.Val_Cell(x)}
      { this.val := x }

   introduce get(): int
   dynamic {this.Val(v)}_{this.Val(v) * Res = v}
   static {this.Val_Cell(v)}_{this.Val_Cell(v) * Res = v}
      { Res := this.val }
}
```

```
class DCell inherit Cell {

   override set(x: int)
     { this.Cell::set(2*x) }

}
```

- Is this a “proper” use of inheritance?
- Can one ever hope to verify such code?

Surprise! No problem for verification

```
class Cell {
   // apf definitions
   define x.Val_Cell(n) as x.val ↦ n
   // field declarations
   val: int

   // methods
   introduce set(x: int)
   dynamic {this.Val(_)}_{this.Val(x)}
   static {this.Val_Cell(_)}_{this.Val_Cell(x)}
      { this.val := x }

   introduce get(): int
   dynamic {this.Val(v)}_{this.Val(v) * Res = v}
   static {this.Val_Cell(v)}_{this.Val_Cell(v) * Res = v}
      { Res := this.val }
}
```

```
class DCell inherit Cell {
   // apf definitions
   define x.Val_DCell(n) as false
   define x.DVal_DCell(n) as x.Val_Cell(n)

   // methods
   override set(x: int)
   dynamic {this.Val(_)}_{this.Val(x)}
         also {this.DVal(_)}_{this.DVal(2*x)}
   static {this.DVal_DCell(_)}_{this.DVal_DCell(2*x)}
     { this.Cell::set(2*x) }

   inherit get(): int
   dynamic {this.Val(v)}_{this.Val(v) * Res = v}
         also {this.DVal(v)}_{this.DVal(v) * Res = v}
   static {this.DVal_DCell(v)}_{this.DVal_DCell(v) * Res = v}
}
```

The key insight: **define** x.$\text{Val}_{\text{DCell}}$(n) **as** false

The proof obligations (e.g. behavioural subtyping) are trivialized

DCell ensures that no client will ever have a Val predicate for a Dcell object. Therefore, in the “Val-world”, DCell is not a subtype of Cell (that is, a variable of static type Cell that satisfies Val will not point to a DCell object)

Apfs specify logical inheritance

Conclusion

Separation logic for reasoning about shared mutable state

Apfs and static/dynamic method specs allow flexible handling of inheritance

The combination (Parkinson & Bierman's system) suits the OO paradigm well. Modular and intuitive. Can verify common design patterns

Implemented in tools: jStar, VeriFast

This was just the basics – there is much more in the paper, and several extensions also exist