Modular Reasoning about Object-Oriented Programs using FRL and Typestates

Yuyan Bao and **Gary T. Leavens**
University of Central Florida

IFIP WG 2.3, Mooloolaba Brech, QLD, Australia
July 20, 2017

www.cs.ucf.edu

A Whale of a Good Time: Thanks!

FSE 2018 in Orlando

ACM SIGSOFT Conference on Foundations of Software Engineering

- Papers due March 9, 2018
- Conference November 4-9, 2018
- See 2018.FSEconference.org for details

Modular Reasoning about Object-Oriented Programs using FRL and Typestates

Yuyan Bao and **Gary T. Leavens**
University of Central Florida

IFIP WG 2.3, Mooloolaba Brech, QLD, Australia
July 20, 2017

www.cs.ucf.edu

Background

- Modular reasoning means conclusions remain valid when code is placed in the context of a larger program.
- FRL = **F**ine-grained **R**egion **L**ogic (Bao and Leavens 2015) based on Region Logic (Banerjee, Nauman, and Rosenberg 2013)
- Typestates serve as abstract zero-argument predicates on a program's data representation; replace invariants in this work

Goals

- Modular treatment of framing in OOP with subtyping, including:
  - Local Reasoning, as in Separation Logic (SL)
    - Verified code should have no runtime errors
  - Patterns of sharing (e.g., Subject-Observer)
  - Flexible invariant-like properties (typestates)
    - Avoid problems with re-entrance (callbacks)
    - Summarize when properties hold
    - Value dependent on object's fields
  - Information Hiding

Approach: Methodology using FRL and TypeStates (1 of 2)

- For framing:
  - Specify method read (δ) and write effects (ε)
  - Extended state in subtypes must be encapsulated
  - Rules for overriding predicates, typestates
- For data abstraction:
  - Use model fields, predicates, typestates
- For information hiding:
  - Declared modules
  - Specify visibility to define a public, protected, private view

Methodology (2 of 2)

- For flexibility compared to invariants:
  - Use typestates
  - Specify when each typestate must hold (not implicit)

Related Work

- Leavens and Naumann (2015)
  - Considers frames as part of postconditions
  - So not a modular treatment of framing
- Müller (2001)
  - Universe type system (ownership)
  - Some sharing patterns not compatible with ownership (e.g., Subject-Observer pattern)
- Barnett et al. (2004)
  - Uses ghost fields for ownership
  - Specifications must discuss owner object's state, so not strictly local

Related Work

- Deline and Fähndrich (2004)
  - Linear types for alias control
  - Supports only restricted forms of sharing
  - Cannot express some patterns (e.g., Subject-Observer)

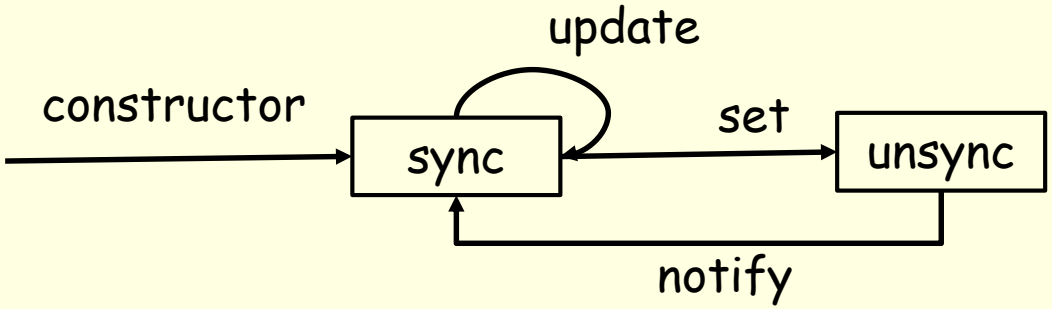
Subject Public View (1 of 2)

```
module SubObs {
public class Subject {
    public model a_val : int;
    public ghost repr : seq<Observer>;
    public pure fpt() : region reads fpt();
    public predicate valid_cache() reads /*... */
     { (∀ o:Observer : o in repr : o.sync) }
    public typestate sync exports valid_cache()
        reads fpt();
```

Subject Public View (2 of 2)

```
public class Subject {  /* … slide 1 of 2 …*/
   public Subject()
      modifies region{this.*};
      ensures a_val = 0 && repr = [] && sync;

   public update(v: int)
      requires sync;
      modifies region{this.a_val},
                  filter(fpt(), Observer, a_cache);
      ensures a_val = v && sync;

   public pure get() : int
      reads region{this.a_val};
      ensures ret = a_val;
```

Subject's Typestates



public typestate sync **exports** valid_cache()
reads fpt();

protected typestate unsync **reads** fpt();

Protected Methods of Subject

```
protected notify()
  modifies filter{fpt(), Observer, cache};
  ensures sync;

protected set(v : int)
  requires sync;
  modifies region{this.val};
  ensures a_val = v && unsync;
}
```

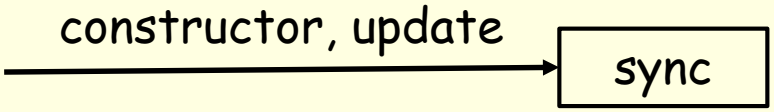
Observer Public View (1 of 2)

```
module SubObs { /* Subject as previously */
public class Observer {
    public model a_cache : int;
    public model a_sub : Subject;

    public pure fpt() : region reads fpt();

    public predicate valid() reads fpt()
     { this.a_cache = sub.a_val }

    public typestate sync = valid();
```

Observer Public View (2 of 2)

```
public class Observer {  /* slide 1 of 2 */
    public Observer(s: Subject)
      requires s ≠ null && this !in s.repr && s.sync;
      modifies region{this.*}, region{s.repr};
      ensures a_sub = s && s.repr = old(s.repr) + [this]
                  && s.sync && this.sync;

    public pure get() : int
      reads region{this.a_cache};
      ensures ret = a_cache;
} }  // end class and module
```

Observer's Typestate



public typestate sync **exports** valid()
reads fpt();

Protected Method of Observer

```
protected void update()
   requires sub ≠ null;
   modifies region{this.cache};
   ensures sync;
```

Why Typestates?

- Help describe intuition about state transitions
- Flexibility: need not hold in all (visible) states
- Re-entrance (callbacks) can be specified easily
- Avoids repeating specification expressions in pre- and postconditions
  - Typestates are named abstractions

For information hiding:

- Typestates export what clients can know

Supertype Abstraction: Local Reasoning and Framing

Suppose *m* is an instance method of *T* with specification:

requires P_T;

modifies ε_T;

ensures Q_T;

Let *R* be a predicate that does not depend on ε_T.

Then the following should be valid for all *o*:*T*

{ *o* != null && P_T && *R* } *o*.*m*() { Q_T && *R* }

This should be valid if *o* is an object of a subtype.

Notation: Framing Judgments, Separation, Quadruples

- $P \vdash \delta$ frm R means if P holds, then δ frames R
  i.e., every location that R depends on is in δ
  - δ denotes a set of locations (vars and fields)

- $\delta \div \varepsilon_T$ means the sets that δ and ε_T denote are disjoint
  - Banerjee et al. write this operator as $\not\cdot$

- $\{P\}\ C\ \{Q\}\ [\varepsilon]$ means $\{P\}\ C\ \{Q\}$ and all locations that C modifies are in ε

Local Reasoning by Framing

FRL uses frame rules to extend a local specification to a property of a larger program:

$$\frac{\vdash \{P\}\ C\ \{Q\}\ [\varepsilon]\ ,\quad P \vdash \delta\ \mathbf{frm}\ R}{\vdash \{P\ \&\&\ R\}\ C\ \{Q\ \&\&\ R\}\ [\varepsilon]}$$

where $P\ \&\&\ R \Longrightarrow \delta \div \varepsilon$

Supertype Abstraction

Consider a type T, a variable $o{:}T$, an assertion R.

If T's method m satisfies:

$$\{P_T\}\ o.m()\ \{Q_T\}\ [\varepsilon_T],$$

and $P_T \vdash \delta$ **frm** R,

and P_T && $R \Longrightarrow \delta \div \varepsilon_T$

then $\vdash \{P_T \text{ \&\& } R\}\ o.m()\ \{Q_T \text{ \&\& } R\}\ [\varepsilon_T]$

Suppose $S \leq T$ *and* o **instanceof** S

All of the above should still hold;

What does that tell us about the spec. of S?

Sufficient for Soundness: Restricting Subtype's Effects

Let $S \leq T$, and be an instance method of T, specified as:

$\{P_T\}\ m()\ \{Q_T\}\ [\varepsilon_T]$.

So S's specification of m,

$\{P_S\}\ m()\ \{Q_S\}\ [\varepsilon_S]$.

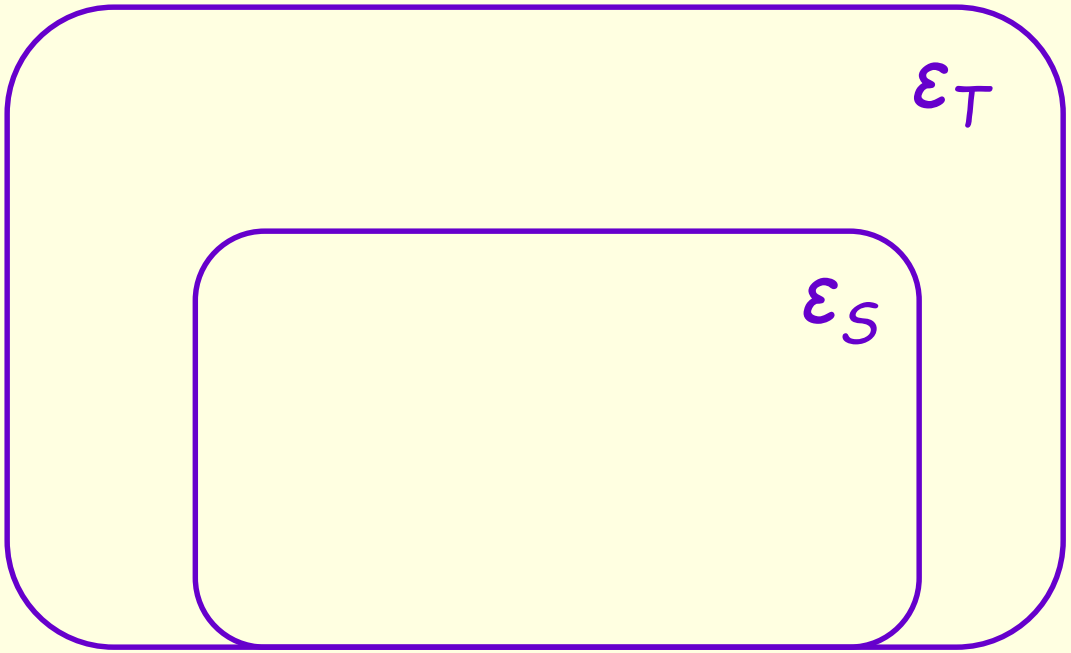
should satisfy:

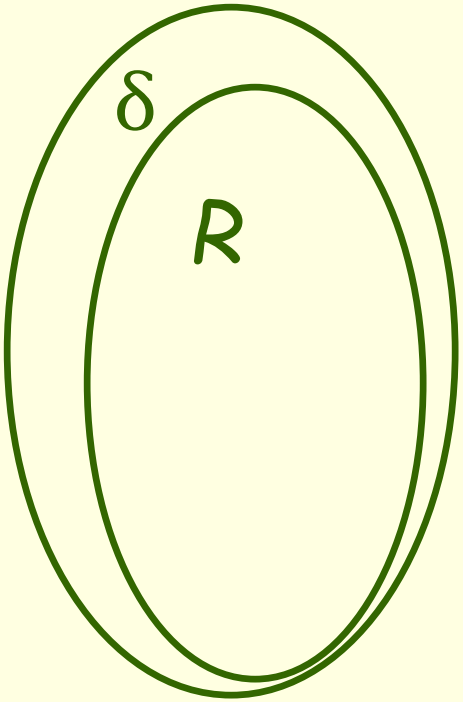
$P_T \Rightarrow P_S$,

$\mathbf{old}(P_T) \Rightarrow (Q_S \Rightarrow Q_T)$,

and $\varepsilon_S \leq \varepsilon_T$

Visualizing the Effects

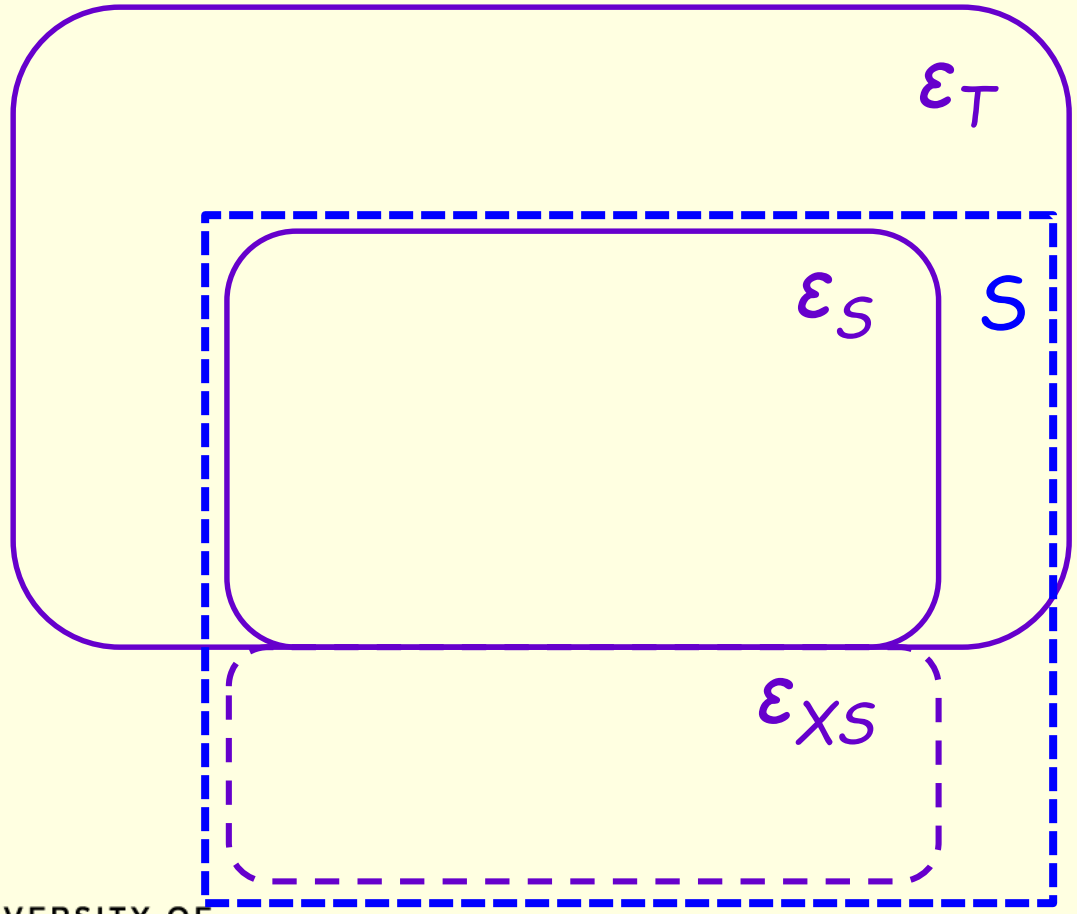


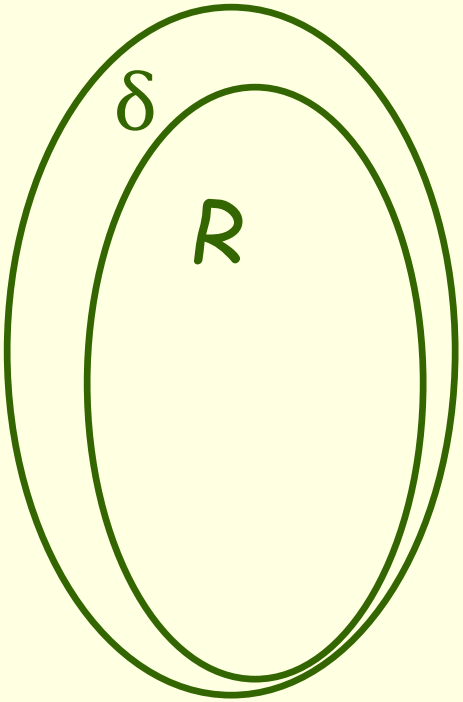


The Extended State Problem (Leino '98)

- Subtypes may have more fields than supertypes
  - Subtype methods often change those fields
  - How to know the subtype's fields are disjoint from those framing preserved assertions (δ)?

Allowing Extended State in Subtype (S)





Extended State in Subtype

Write effects:

- Parts per supertype (T):
  - Inherited, in overridden methods $\varepsilon_S \leq \varepsilon_T$
  - Extended
    - Not modified in overridden method, may be shared
    - Modified in overridden method and encapsulated, ε_{XS}
- Requirements for soundness:
  - Locations in ε_{XS} are encapsulated by S
  - Footprint of T and locations in ε_{XS} are disjoint

Encapsulated Locations

- *Encapsulated locations* are locations that cannot be aliased outside the object's footprint.
- Effectively need a notion of ownership to prevent such aliasing

An Example Supertype

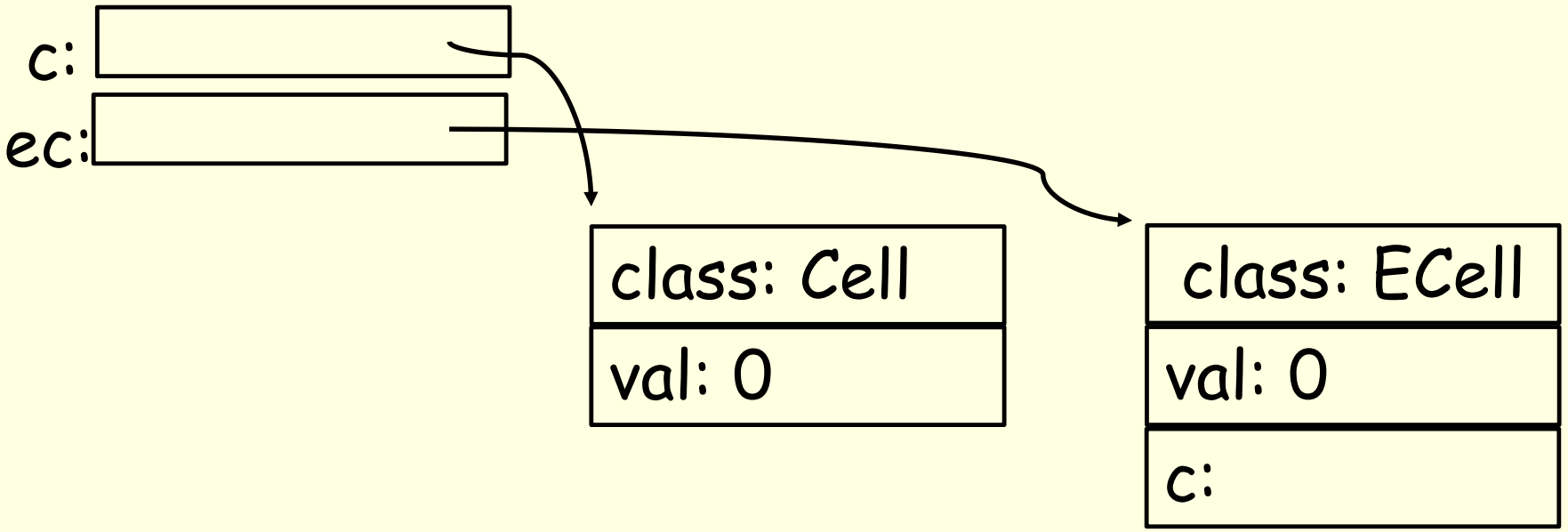
```
protected class Cell {
   protected  val : int;

   public pure  fpt() : region reads fpt()
   { ret := region{this.val}; }

   public predicate  Val(v: int) reads fpt(); { val = v }

   public set (v : int) modifies fpt(); ensures Val(v);

   public pure get() : int reads  fpt() { ret := val; }
}
```

Subtype to Show need for Encapsulated State

```
protected class ECell : Cell {  // val is inherited
   protected c : Cell;

   public pure fpt() : region reads fpt();
   { ret := super.fpt() + region{this.c} + c.fpt(); }

   public predicate Val(v : int)
   { super.Val(v) && (c ≠ null ⇒ c.Val(v)) }

   public typestate valid = Val(get());

   public ECell(c : Cell)
     modifies fpt(); ensures this.c = c && valid;
   { super(c.get()); this.c := c; }    // violates encapsulation!

   public set(v : int) modifies fpt(); ensures valid;
   {  super.set(v);  c.set(v); }}
```

Argument Exposure

```
c : Cell;        c := new Cell();
ec : ECell;      ec := new ECell(c);
```



Supertype Abstraction doesn't work for Non-Encapsulated Locations

```
public useCell(c1 : Cell, c2 : Cell)
    requires c1 ≠ c2;
{ tmp : int;   tmp := c2.get();
  assert c2.get() = tmp;
  c1.set(1);
  assert c2.get() = tmp;
}

public client()
{ c : Cell;        c := new Cell();
  ec : ECell;     ec := new ECell(c);
  useCell(ec, c);  }
```

What happens

```
assume c1 ≠ c2 && c1.get() = 0 && c2.get() = 0;
tmp : int;   tmp := c2.get();
assert tmp = 0;
assert c2.get() = tmp;
c1.set(1);
assume c1.get() = 1 && tmp = 0;
// here c2.get() is 1 due to aliasing!
assert c2.get() = tmp;   // fails!
```

Future Work

- Settle details of the methodology
  - How to enforce encapsulation?
  - What are the rules for client reasoning, and reasoning for subclasses/module clients
  - More flexible ways to deal with sharing/aliasing?
- Soundness proof
- Merge into JML?

Contributions

- Another perspective on modular framing
  - More complete than SL or ownership?
- Another perspective on invariants
  - How useful in practice?

Questions?