Ownership Types After Ten Years

JOHN POTTER
YI LU

COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF NEW SOUTH WALES, SYDNEY

Outline

- The problem
  - Flexible Alias Protection
- Implicit structure in object graphs
  - Ins and Outs of Objects
- Imposing object structure in programs
  - Ownership Types
- Variations on the ownership theme
- Ownership and accessibility
- Ownership effect systems
- Object validity
  - Oval

The Problem

- Aliasing is endemic in OO programming
  - Objects have identity + mutable state
  - Knowing the object ID gives access to the object state
    - Either directly or indirectly
- Mutable state + sharing creates problems
  - To understand program behaviour:
    - An object's invariants may depend on other aliased objects
    - Need to understand the topology of the object graph
    - Loses modularity in program reasoning
  - When objects are updated, their clients may need to adapt
    - But there may be no local knowledge of this object dependency
    - Object notification is difficult

Ownership Prehistory: The Geneva Convention on the Treatment of Object Aliasing

- Formulated by 5 researchers at ECOOP'91
  - John Hogg, Bell-Northern Research &
    Doug Lea, SUNY Oswego &
    Alan Wills, University of Manchester &
    Dennis deChampeaux, Hewlett-Packard &
    Richard Holt, University of Toronto
- Will *port1 transferTo: port2 amount: $100.00* really decrease the amount of money in *port1*
  - Two ways to fail:
    - *port1 == port2* which is easy to check for (a direct alias)
    - Or the two portfolios share the internal account involved in the transfer which is not easy to check for (an indirect alias)

Ownership Prehistory: The Geneva Convention on the Treatment of Object Aliasing

- Categorised 4 approaches to aliasing:
  - Detection.
    - Static or dynamic (run-time) diagnosis of potential or actual aliasing.
  - Advertisement.
    - Annotations that help modularize detection by declaring aliasing properties of methods.
  - Prevention.
    - Constructs that disallow aliasing in a statically checkable fashion.
  - Control.
    - Methods that isolate the effects of aliasing.

Ownership Prehistory: Full Encapsulation: Islands and Balloons

- Islands (Hogg 91) and Balloons (Almeida 97) provide alias protection
- Full encapsulation => objects inside an island/balloon
  - Cannot be referenced from outside
  - Cannot refer to other objects outside
    - Internal aliasing is OK
- Tends to be overly restrictive
  - A container cannot share its elements with another container
  - To allow ease of use of encapsulated objects, both approaches allow dynamic aliases (via local variables)
- Enforcement of full encapsulation
  - Islands used annotations with run-time checks
  - Balloons advocated a complex static analysis
    - Unusable in practice

Ownership Conception: Flexible Alias Protection

- Noble, Vitek, Potter: ECOOP'98
- Language level access modifiers are too weak
  - An object referenced via a private field may be returned via a public method
    - Gave rise to security hole in Java 1 applet security model
  - Access modifiers do not control aliasing
- Full encapsulation techniques are too strong
- Flexible alias protection aims to allow benign forms of aliasing

Ownership Conception: Flexible Alias Protection

- *Aliasing modes* for object references
  - Rep
    - For internal representation
    - Allows internal aliasing but no export
  - Arg (with Role)
    - For “arguments” or shareable elements of a container
    - Only access immutable interface of referenced objects
  - Free
    - For new unbound objects
  - Val
    - Immutable objects
  - Var (with Role)
    - The escape hatch …

Ownership Conception: Flexible Alias Protection


```
class Course<arg s Student> {
    private rep Hashtable<arg s Student, rep RawMark>
    marks = new Hashtable();

    public void enrol (arg s Student s) {
        rep RawMark r = new RawMark();
        marks.put(s, r);
    }

    public void recordMarkFor(arg s Student s,
                              val String workUnit,
                              val int mark) {
        marks.get(s).recordMarkFor(workUnit, mark);
    }

    public void finalReport (arg s Student s) {
        marks.get(s).finalReport();
    }
}
```

Ownership Conception: Flexible Alias Protection

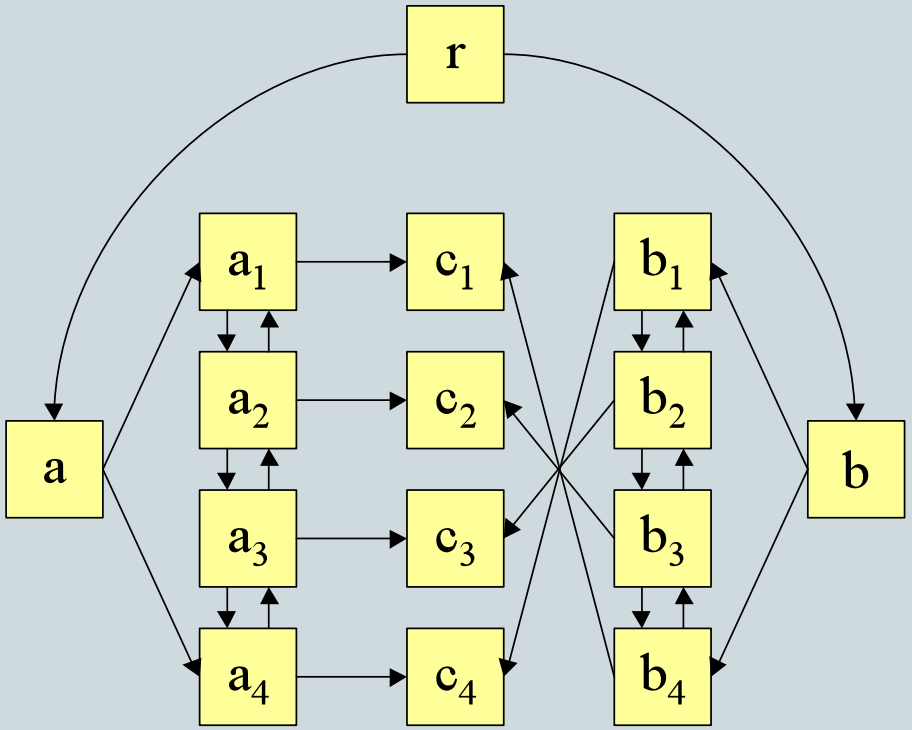
- No formal model developed
- Implementation attempted (by Dave Clarke) in Pizza
  - Martin Odersky's experiment with generics in Java
  - Provided a vehicle with type parametric classes
  - Pizza type checking code hard to modify
  - Unspecified type rules to implement!
- Inspirations from FLAP
  - Need to be able to partition object graphs somehow
  - Need to develop a formal type system
  - Issues with various code idioms and design patterns
  - Potential applications such as memory management and concurrency control

Prenatal Ownership: Implicit Structure in Object Graphs

- The Ins and Outs of Objects
  - J. Potter, J. Noble, and D. Clarke.
    - In Australian Software Engineering Conference (ASWEC), 1998
    - Most Valuable Paper awarded in 2008
- Partitioning of object graph
  - Lattice structure for sets of separating objects
  - James told John it's too complex
  - Attempt to focus on simplest separators led to rediscovery of graph dominator concept
    - If I'd known more about compilers I would have known about dominators!

An Object Graph

- an application object r
- list header objects a, b
  - a and b are doubly linked lists
  - they share data content
    - data objects c_1, c_2, c_3, c_4
  - their link objects are not shared
    - link objects a_1, a_2, a_3, a_4
    - link objects b_1, b_2, b_3, b_4



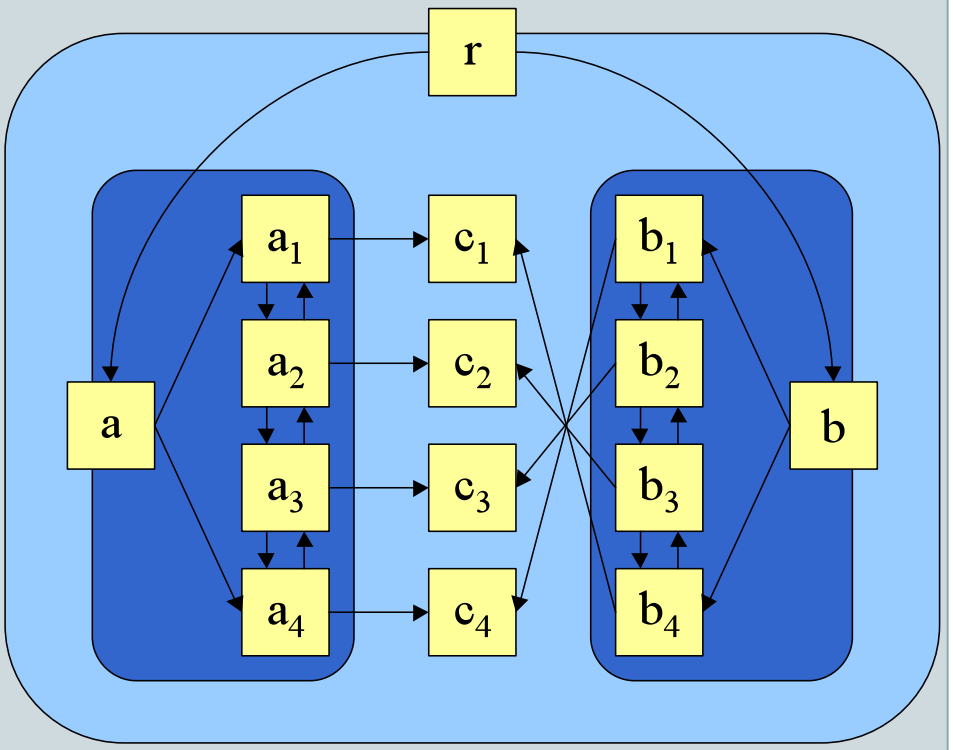
- list b is the reverse of list a

The Ins and Outs of Objects

- all reference paths to an object from a root object may share
  - in graph theory, these are called articulation points, or *dominators*
- the dominators form a tree structure
- our idea: the dominator tree (often) captures the intended object encapsulation structure

Object Dominator Tree

- the blocks in the diagram are associated with an *owner* object
- the blocks contain the objects dominated by the owner
  - e.g. a_1 is dominated by a
  - c_1 is not dominated by a
    - there is an alternative path from r to c_1 via b



Ownership Invariant

- the object reference structure induces the dominator (or ownership) tree
- think of the objects dominated by an owner as being *inside* the owner
- object references can only cross ownership boundaries from the inside to the outside
- the ownership invariant: given objects x, y

  if x refers to y

  then *owner(y)* dominates x

Ownership Monitoring

- track dominator tree for all objects on the heap at run-time
- ownership will need to be updated if the ownership invariant is violated
  - this can only happen with object field assignment
- in practice for Java, the stack plays the role of a root object, and we further exploit the stack structure to yield a stack of dominator trees
  - *dominator update* is a challenging algorithm
- version 1: hacked the source code of a JVM
- version 2: instrumented bytecode

Object Visualisation

- Idea: display object graph at run-time
  - problem: how to do graph layout?
  - solution: use a tree structure
  - problem: what tree?
- Creation tree: creator as parent
  - advantage: creator is fixed
  - problem: objects often out-live their creators
- Ownership tree: owner as parent
  - advantage: relatively stable, owners out-live their objects, references do not cross into encapsulations
  - problem: ownership needs to be updated dynamically

Object Visualisation

- OTOG was first attempt: same example as above

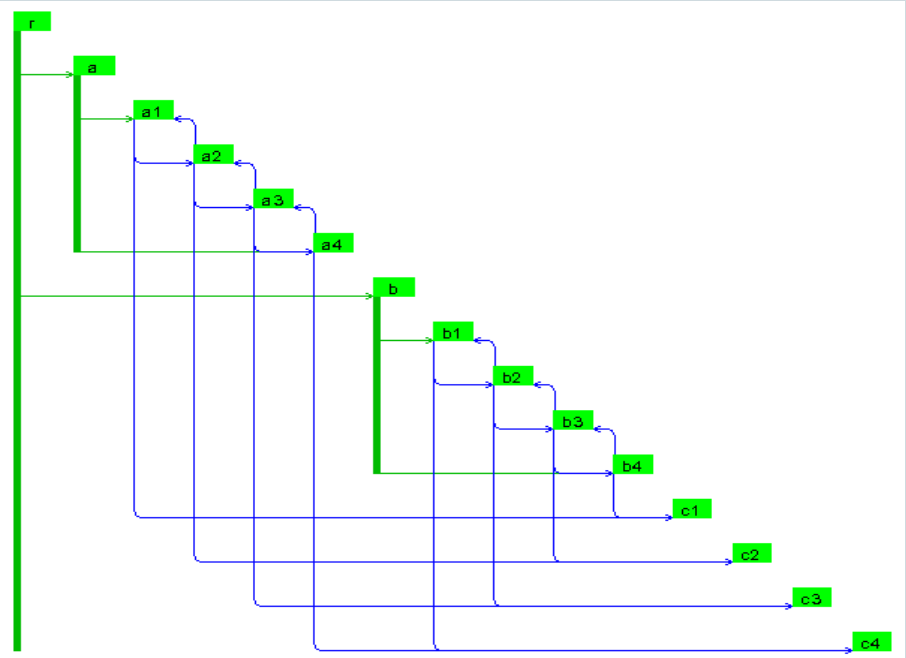


Figure 5. OTOG Graph Layout

Object Visualisation

- Dino was greatly improved second attempt
  - Moral: student slaves produce better work than paid lackeys
- Trent Hill, 4th year project at Macquarie

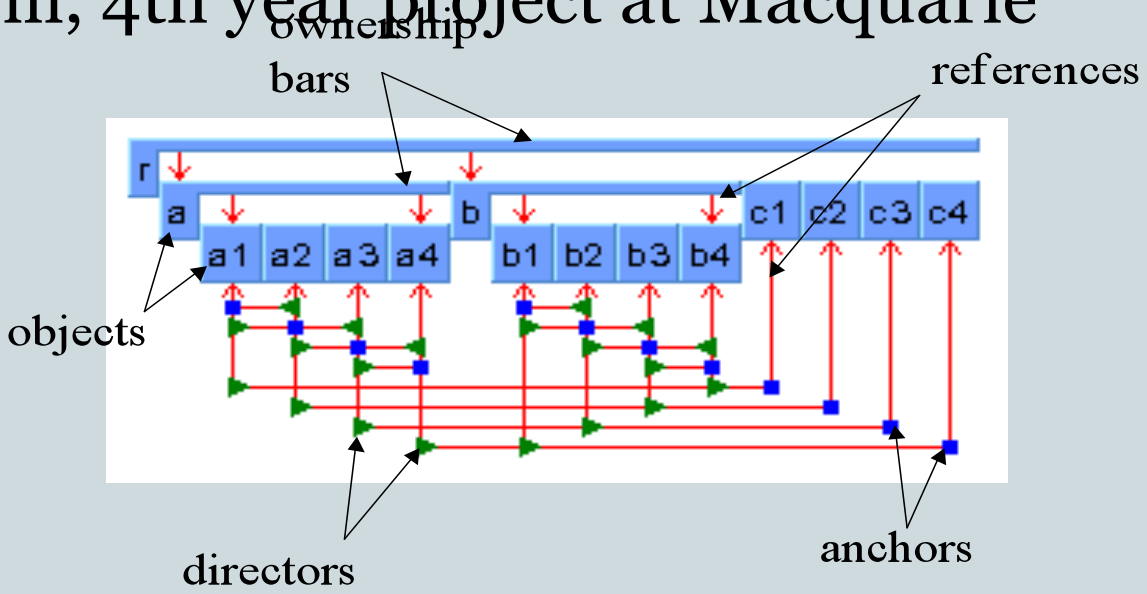


Figure 7. Arma-Dino Ownership Tree

More Than One Thread

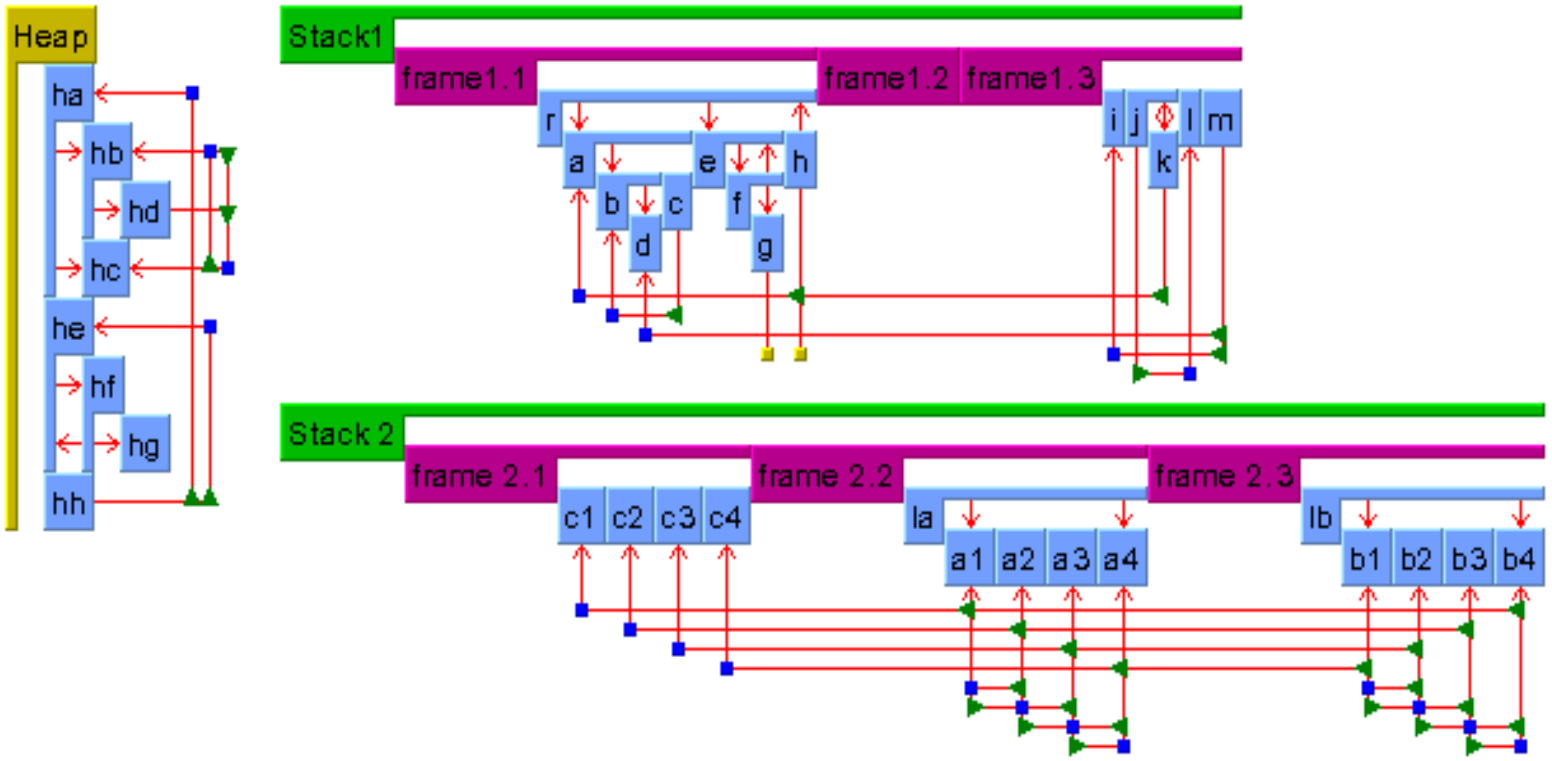


Figure 10. DINO Layout

Displaying Class Names

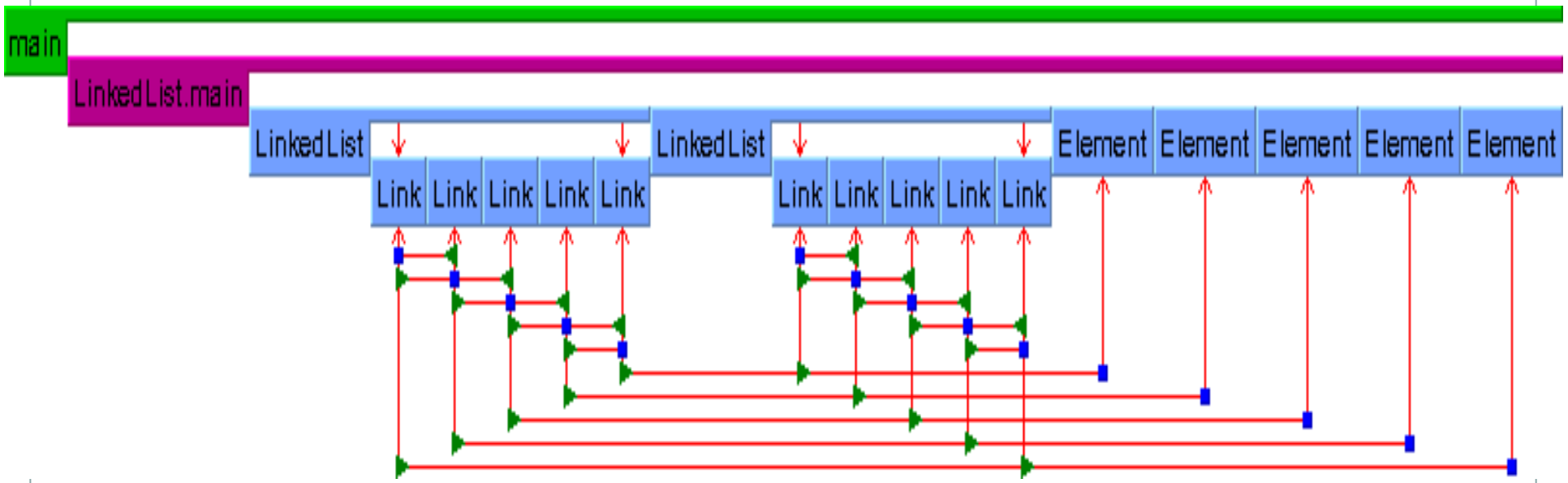


Figure 13. Example Visualisation (cont'd)

Collapsing Tree Nodes

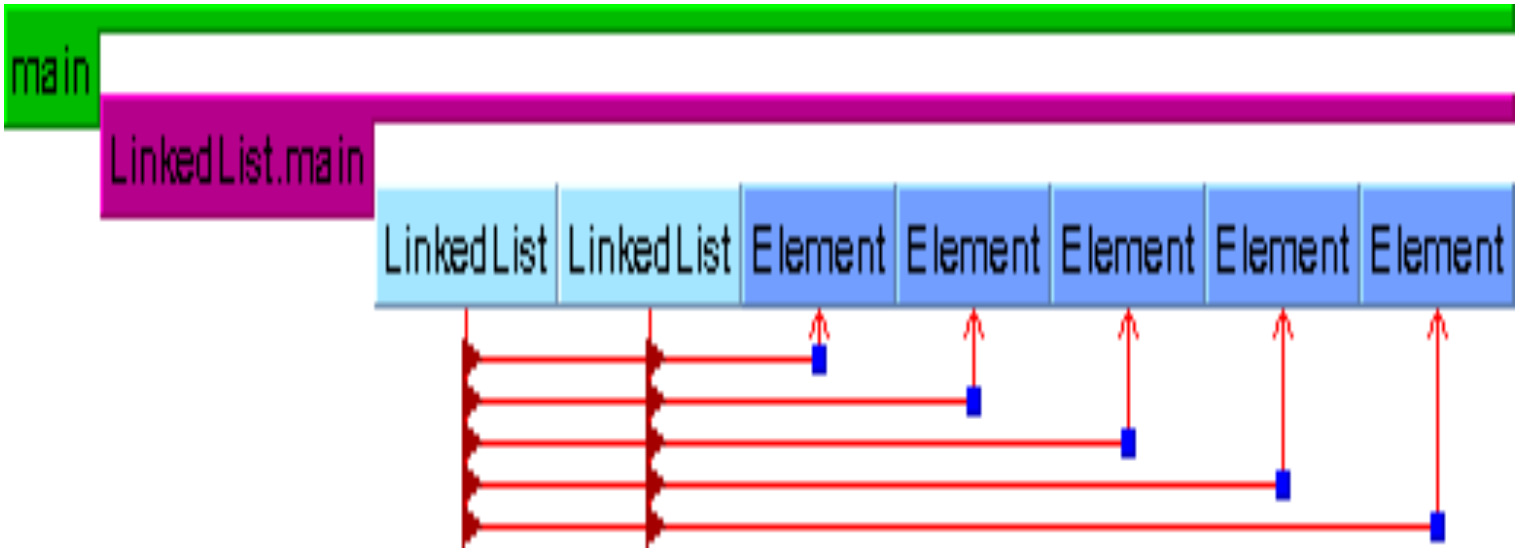


Figure 14. Collapsed Nodes

Alternative Views

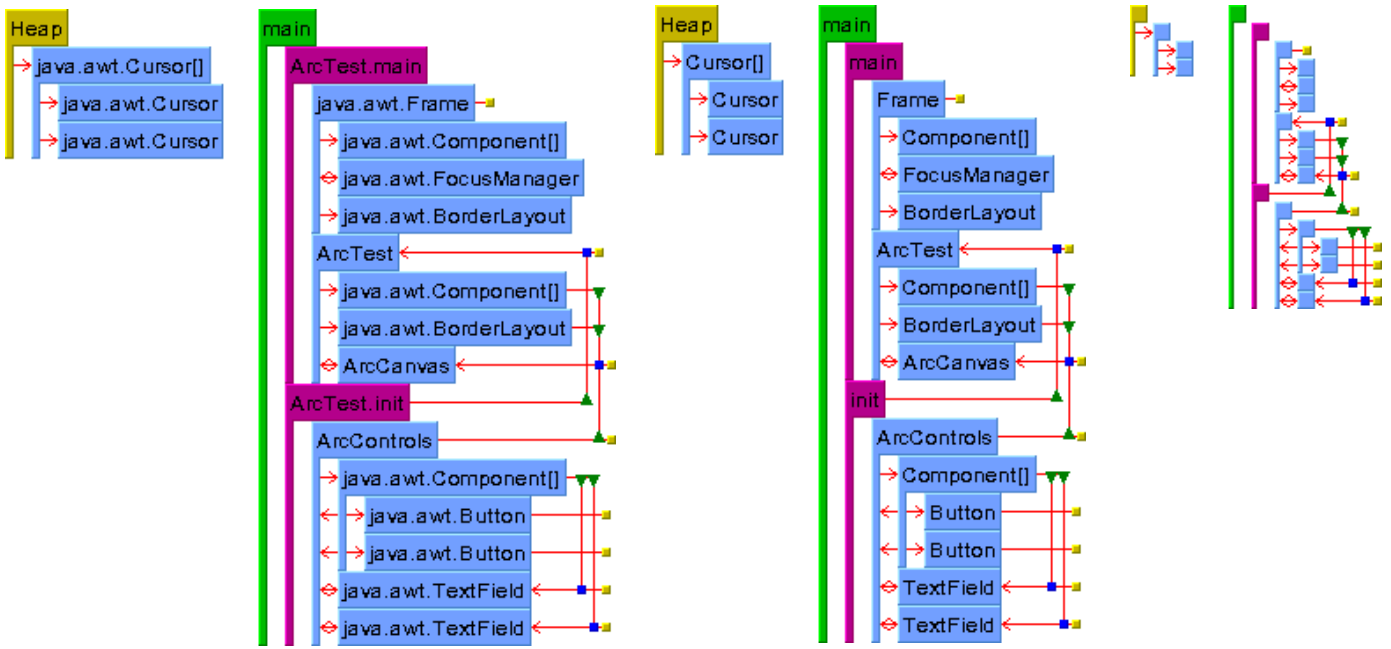


Figure 15. Verbose, Brief and Compressed Modes

Views of Ownership

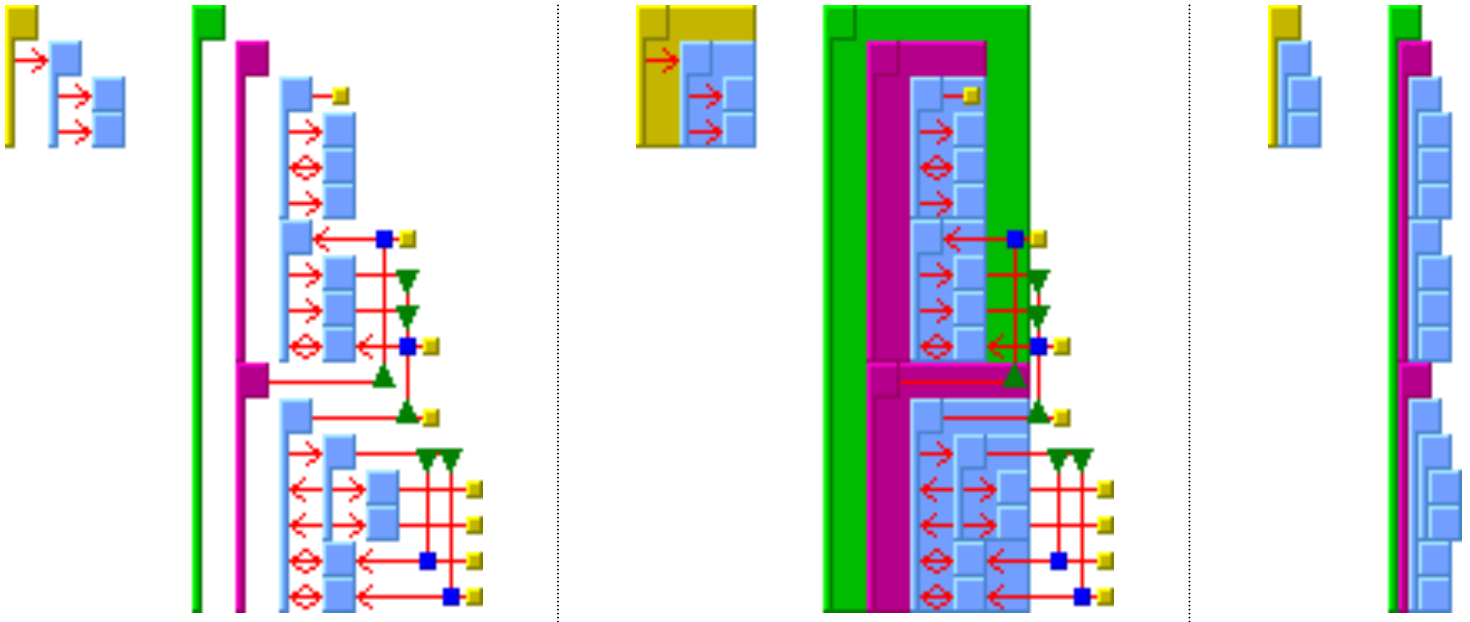


Figure 16. Normal, Set and Cell Views

The Birth of Ownership Types

- Dynamic monitoring extracts intended object encapsulation
- Why not allow programmers to document their intentions?
  - Then perhaps a compiler could check for unintended breaches of encapsulation
- First publication on ownership types
  - Clarke, Potter and Noble
    - *Ownership Types for Flexible Alias Protection*
    - OOPSLA 1998
    - Awarded Most Valuable Paper in 2008

Ownership Types

- Every class has an *owner* parameter
  - when a new object of the class is created, the owner must be specified
    - either using an existing owner, or as *this*
    - all existing owners are accessed via type parameters
- Objects owned by *this* are internal objects
  - their type cannot be accessed by any other external object
  - inability to name *this* is how we statically enforce the ownership invariant
  - now called the owners-as-dominators model
- The owner is part of the type of an object
  - dynamically, ownership forms a tree which is extended with each new object creation
  - ownership types are a simple kind of dynamic type
  - syntactically, this can work nicely with generic types

Example for Ownership Types


```
class Stack<X> {

        this::Link<X> head;

        void add(X x) {
                temp = head;
                head = new this::Link<X>;
                head.next = temp;
                head.element = x;
        }

        X get() {
                return head.element;
        }
}
```

```
class Link<X> {
        owner::Link<X> next;
        X element;
}
```

Warning on Syntax

- If you read our papers, you will find the syntax much heavier than this
  - We use explicit ownership parameters, and do not marry with generic types
  - This syntax allows us to focus on the key theoretical points
- Alex Potanin's Ownership Generic Java
  - Blends ownership type parameters with
  - Requires minimal change to Java 5+ type checker
  - Uses sensible defaults
    - objects with unspecified owner are in the top level ownership context (i.e. the root level)
    - such objects are not encapsulated and can be accessed from anywhere

Dave Clarke

- PhD thesis
  - *Object Ownership and Containment*
  - completed at UNSW in 2001 (Dave's now at Leuven)
- Formal model
  - Presented in Cardelli's *Object Calculus*
- Recognised distinction between
  - *rep* defining reference capability for an object
  - *owner* defining accessibility
    - In Dave's model this may be an ancestor of *rep* rather than just a parent
- Extends owners-as-dominators
  - X can reference Y ⇔ X.rep is inside Y.owner
- Many other issues and extensions addressed informally in his thesis
  - Required reading for anyone working in ownership related areas

Related Work on Ownership

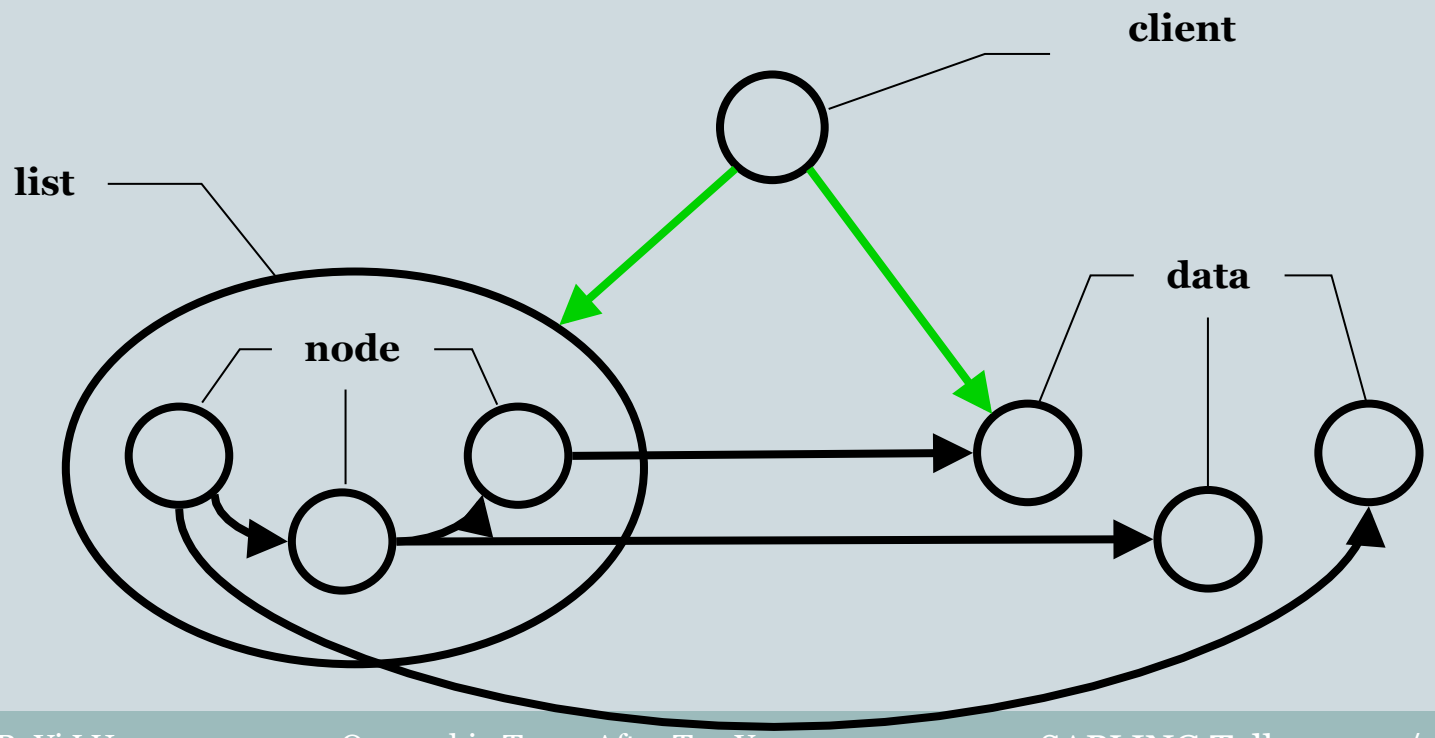
- Boyapati: uses ownership for separating between per thread objects, and shared objects
  - synchronisation control only needed on shared objects
- Other related models:
  - Boskowski and Vitek: confined types
  - Aldrich and Chambers: ownership domains and ArchJava
  - Muller: Universes
  - Clarke and Wrigstad: external uniqueness
  - Boyapati and Liskov: uses inner classes to provide limited form of exposure e.g. for iterators

Ownership and Accessibility

- Lu and Potter
  - *On Ownership and Accessibility*
  - ECOOP 2006
- Similar to Dave Clarke's separation of capability and accessibility
  - But Clarke's model specifies both capability and accessibility as part of object type
  - Lu and Potter define accessibility for reference types, rather than for object types
    - And provide a Java-like notation instead of the Object Calculus
    - New expressions ignore accessibility (object creation)
    - Type declarations require accessibility (use of a reference)

Ownership and Accessibility: Example

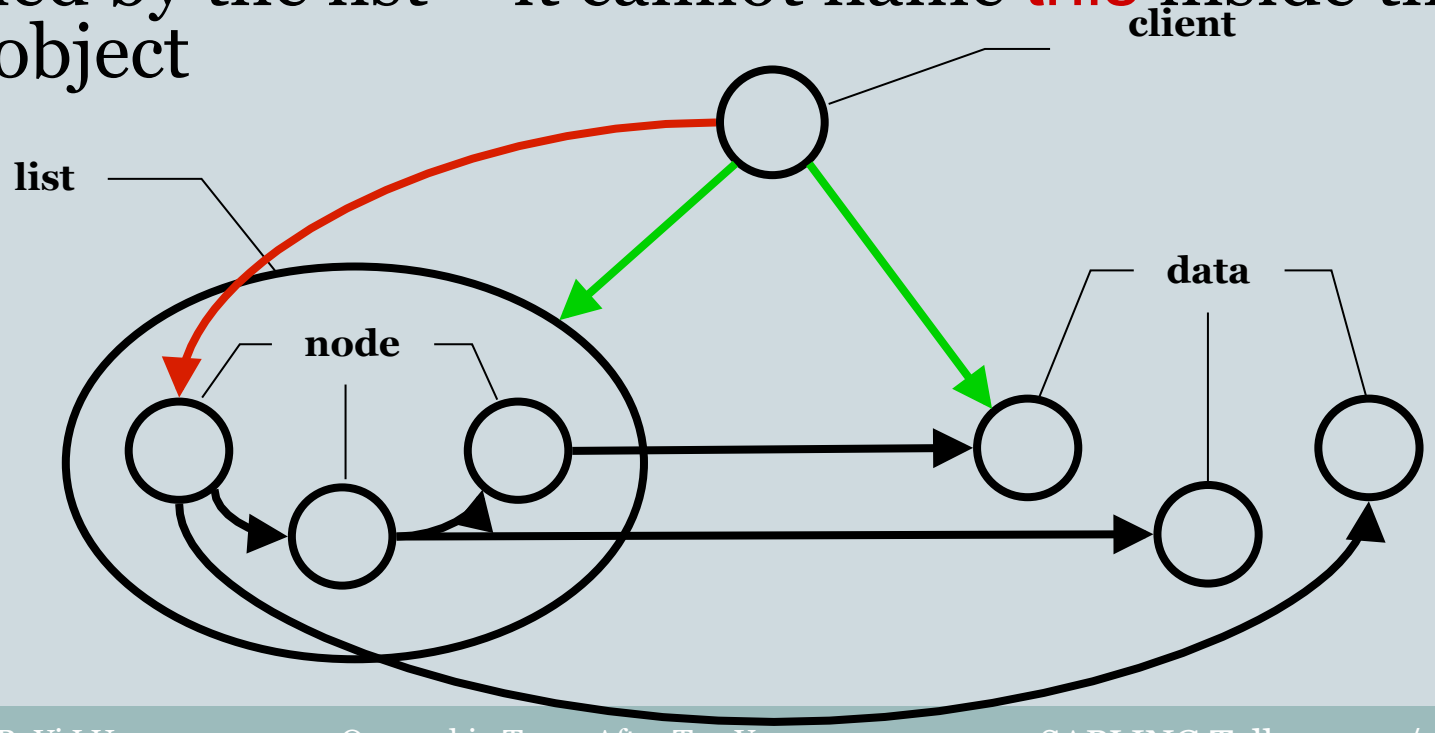
- class List<o, d> { Node<this, d> head; … }
- The client can reference both list and its elements



Ownership and Accessibility: Example

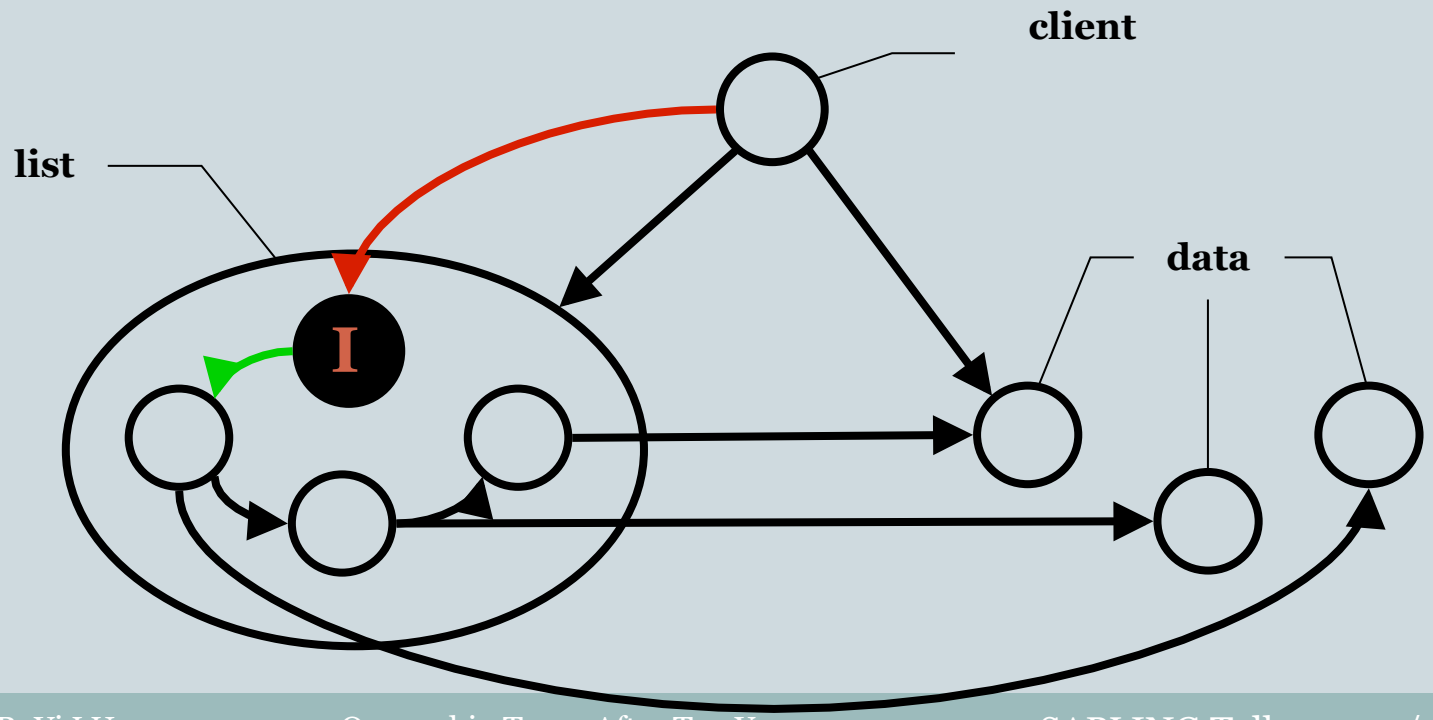
- class List<o, d> { Node<this, d> head; … }

- The client can NOT reference the node objects owned by the list – it cannot name this inside the list object



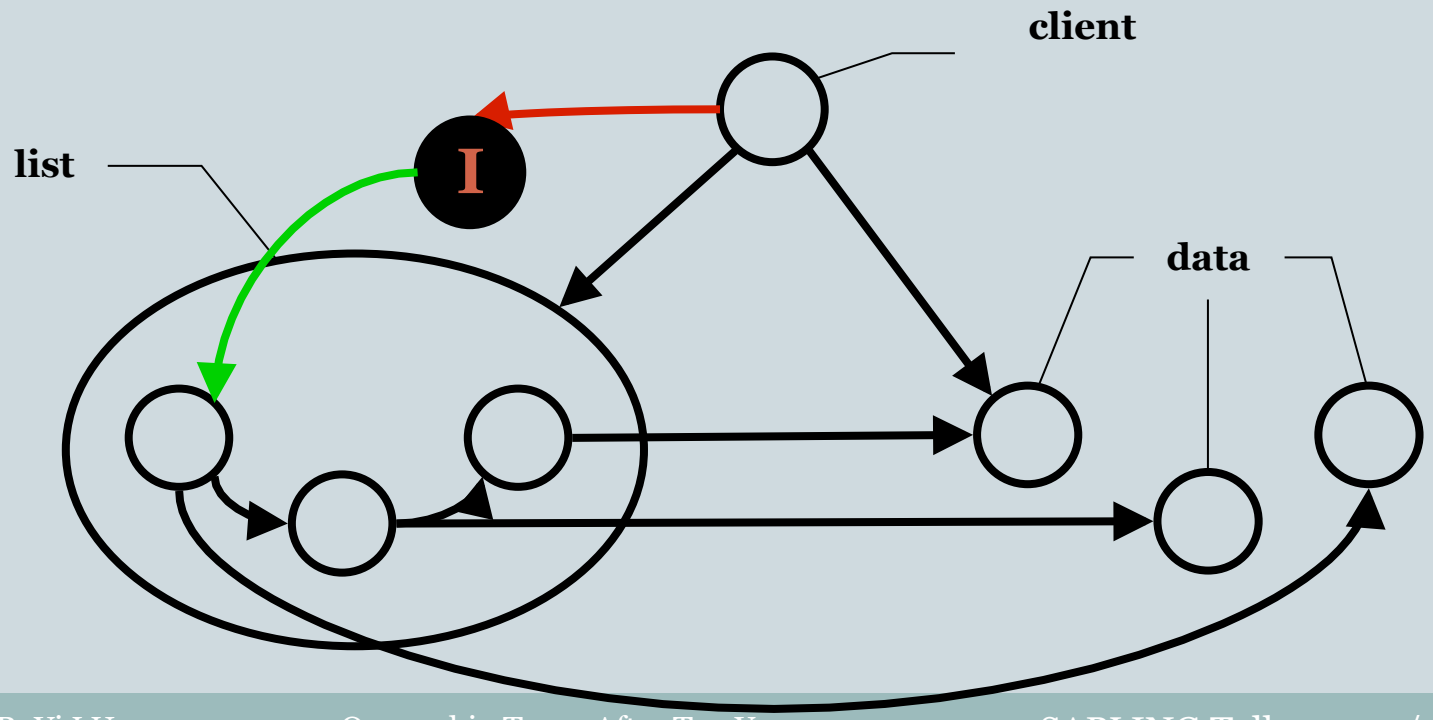
Ownership and Accessibility: Example

- class List<o, d> { Node<this, d> head; … }
- A problem: where should we put an iterator?



Ownership and Accessibility: Example

- class List<o, d> { Node<this, d> head; … }
- A problem: where should we put an iterator?



Ownership and Accessibility

- The challenges and forces:

- Iterators must reference the list's representation (nodes)
- Iterators must be used by the client
- Iterators must NOT expose nodes to the client

- Reference type:
  - [access] C<capability list>
  - access is a *single* owner context
    - Determines the object's accessibility
- **accessibility invariant**:
    - If x→y then x ≤ y.access
  - Allows much more flexible reference structures

A list example with iterator


```
class List<o, d> {
    [this] Node<this, d> head;
    [o] Iterator<this, d> getIter() { return new [o]Iterator<this, d>(head); } }

class Iterator<o, d> {
    [o] Node<o, d> current;
    [d] Data element() { return current.data; } }

// client code
List<this, world> list = new List<this, world>();
[this] Iterator<*, world> iter = list.getIter(); // OK
... = iter.current // ERROR, type is [?] Node<?, d>
iter.element().useMe(); // OK, type is [world] Data
```

A list example with iterator

```
class List<o, d> {
    [this] Node<this, d> head;
    [o] Iterator<this, d> getIter() { return new [o]Iterator<this, d>(head); } }

class Iterator<o, d> {
    [o] Node<o, d> current;
    [d] Data element() { return current.data; } }

// client code
List<this, world> list = new List<this, world>();
[this] Iterator<*, world> iter = list.getIter(); // OK
… = iter.current // ERROR, type is [?] Node<?, d>
iter.element().useMe(); // OK, type is [world] Data
```

client

list

world

A list example with iterator

```
class List<o, d> {
    [this] Node<this, d> head;
    [o] Iterator<this, d> getIter() { return new [o]Iterator<this, d>(head); } }

class Iterator<o, d> {
    [o] Node<o, d> current;
    [d] Data element() { return current.data; } }

// client code
List<this, world> list = new List<this, world>();
[this] Iterator<*, world> iter = list.getIter(); // OK
… = iter.current // ERROR, type is [?] Node<?, d>
iter.element().useMe(); // OK, type is [world] Data
```

client

list

world

A list example with iterator

```
class List<o, d> {
    [this] Node<this, d> head;
    [o] Iterator<this, d> getIter() { return new [o]Iterator<this, d>(head); } }

class Iterator<o, d> {
    [o] Node<o, d> current;
    [d] Data element() { return current.data; } }

// client code
List<this, world> list = new List<this, world>();
[this] Iterator<*, world> iter = list.getIter(); // OK
… = iter.current // ERROR, type is [?] Node<?, d>
iter.element().useMe(); // OK, type is [world] Data
```


client

list

world

A list example with iterator

```
class List<o, d> {
    [this] Node<this, d> head;
    [o] Iterator<this, d> getIter() { return new [o]Iterator<this, d>(head); } }

class Iterator<o, d> {
    [o] Node<o, d> current;
    [d] Data element() { return current.data; } }

// client code
List<this, world> list = new List<this, world>();
[this] Iterator<*, world> iter = list.getIter(); // OK
… = iter.current // ERROR, type is [?] Node<?, d>
iter.element().useMe(); // OK, type is [world] Data
```

client

list

world

Ownership Effect Systems

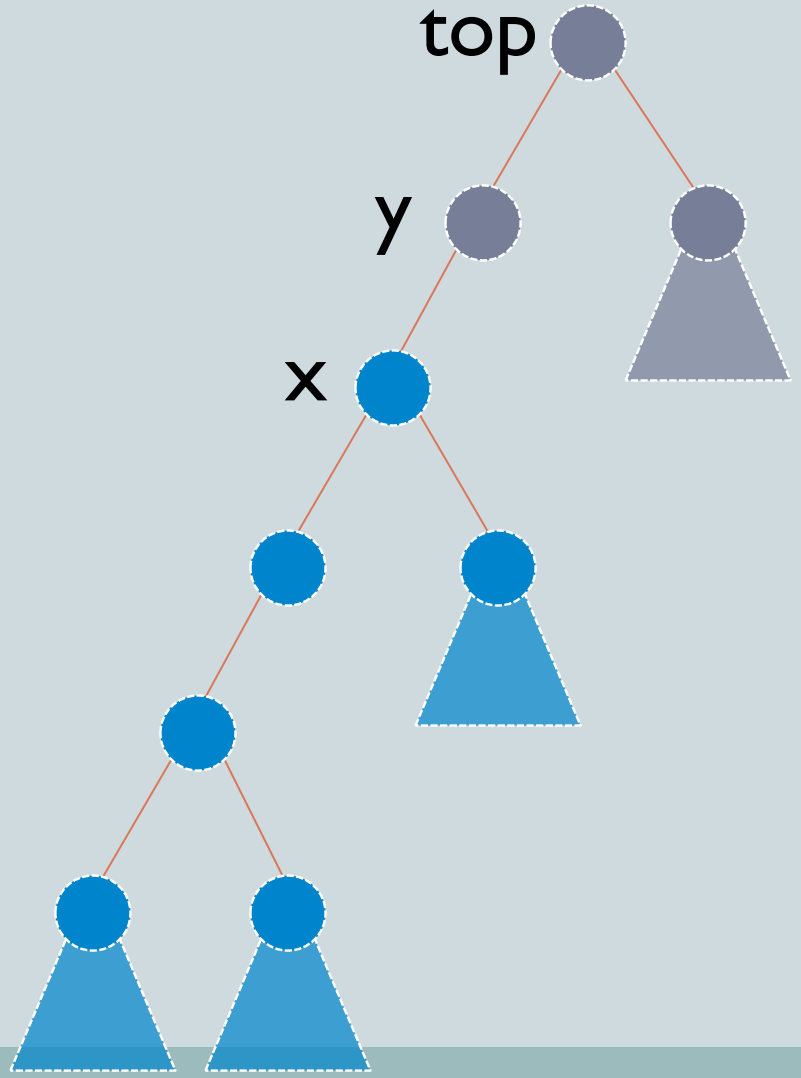
- Greenhouse and Boyland
  - An object-oriented effects system ECOOP 1999
  - Later work on fractional permissions by Boyland
- Clarke and Drossopolou
  - Ownership, encapsulation and disjointness of type and effect. OOPSLA 2002
  - "JOE"
- N. Cameron, S. Drossopoulou, J. Noble, and M. Smith
  - Multiple Ownership OOPSLA 2007
  - "MOJO"

- Read-only and immutability
  - Muller and various others 99+
    - Universes
  - Birka and Ernst 02
    - Javari

Ownership and Object Validity

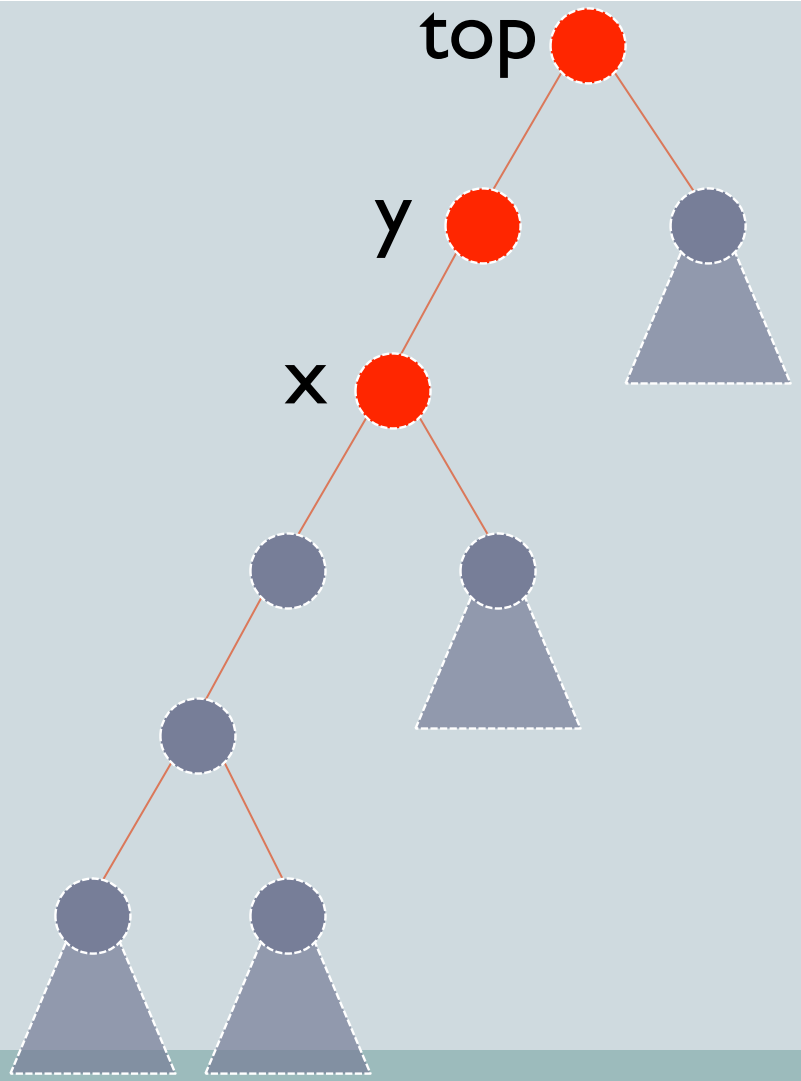
- Lu and Potter
  - Effective Ownership POPL 2007
- Lu, Potter and Xue
  - Validity invariants and effects ECOOP 2007
  - "Oval"
  - Key ideas:
    - Ownership confined dependency
    - Validity contracts for methods
      - Specifies what objects are valid before and after
        - The Validity Invariant
      - and what may be invalidated
        - The Validity Effect

Ownership-confined Dependency



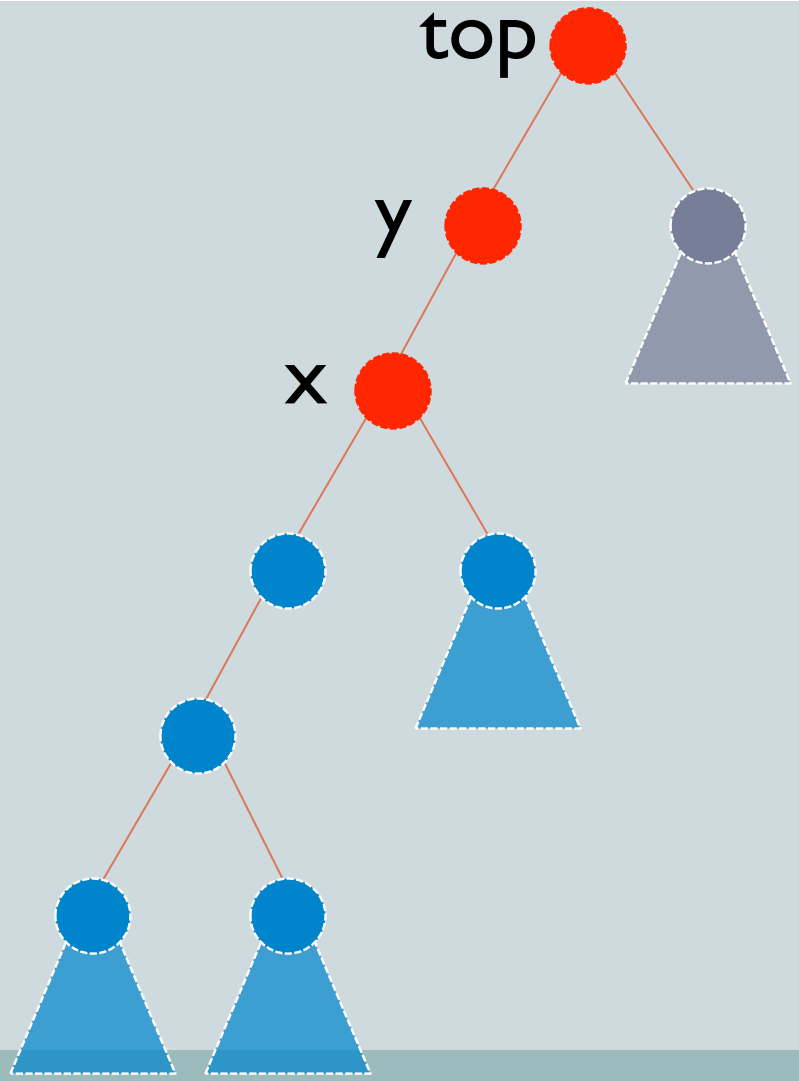
- An object's invariant can only depend on its state and states of its owned objects.
- Dependency is reflexive and transitive
- If x is valid, then all objects x depends on must be valid too

Ownership-confined Dependency



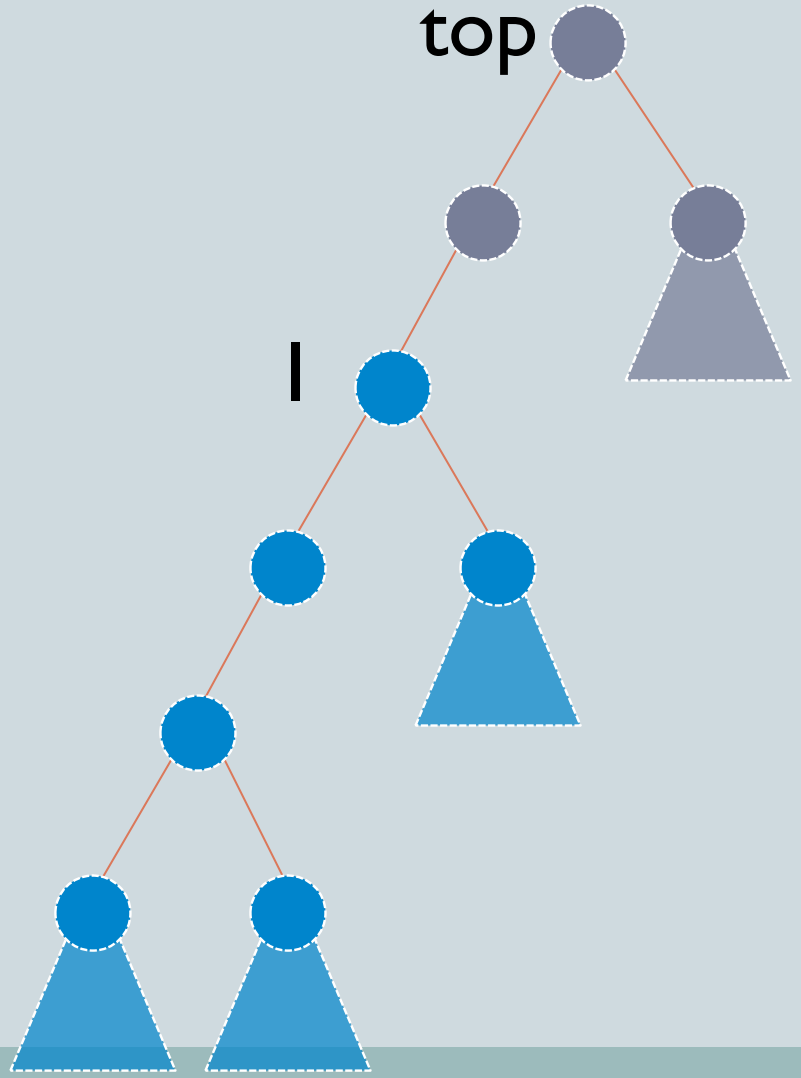
- An object's invariant can only depend on its state and states of its owned objects.
- Dependency is reflexive and transitive
- If x is updated, then all objects depending on x become invalid

Ownership-confined Dependency



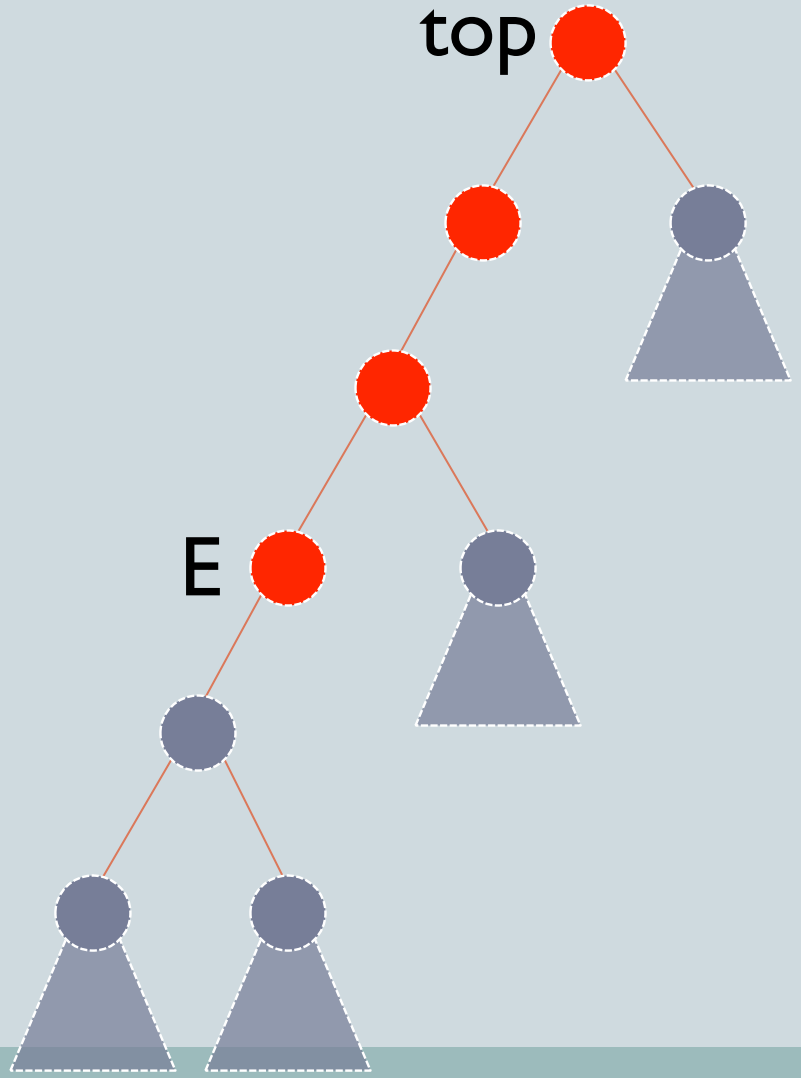
- If x is updated, then all objects depending on x become invalid
- If x was originally valid before update, then all objects owned by x are still valid

Validity Contract in Oval <I, E>



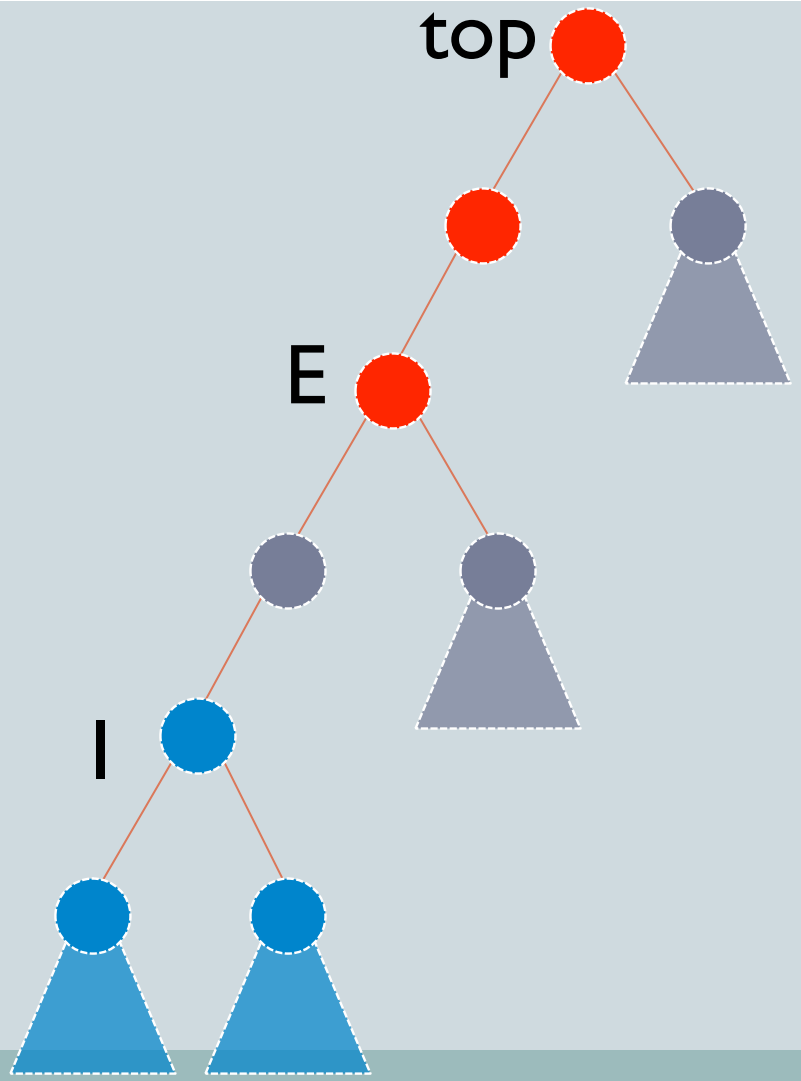
- m(…) <I, E> { … }
- I is the top of the sub-tree
- It abstracts the validity invariant sub-tree

Validity Contract in Oval <I, E>



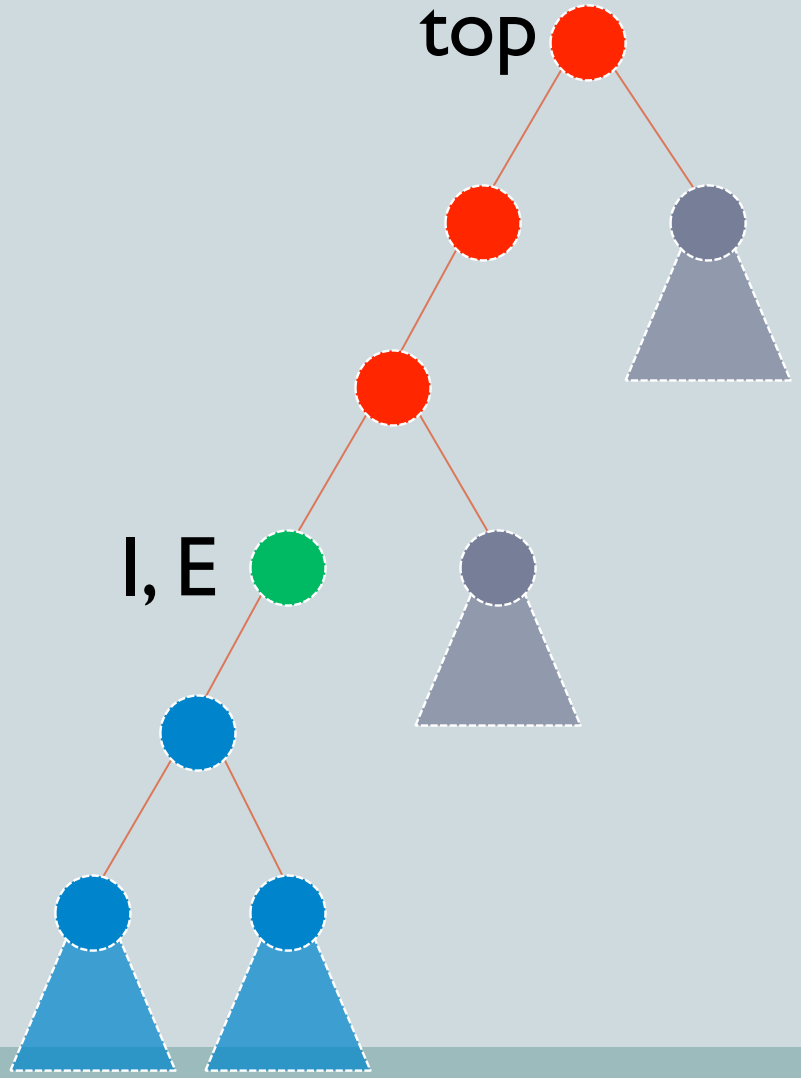
- m(...) <I, E> { ... }
- E is the bottom of the branch from top
- It abstracts the validity effect branch

Validity Contract in Oval <I, E>



- If I < E
- No overlap between validity invariant and effect
- No validation is required

Validity Contract in Oval <I, E>



- If I = E
- The only overlap is the local object
  - I = E = this
- Validation is required for the local object this

Our Current Work

- Extending the Oval model
  - Pre and postconditions for validity contracts
  - Yields a flow sensitive type system
  - Introduce an explicit validity assumption statement to cover lack of reasoning about actual program states
  - System reasons with 2 states per object: valid and invalid
  - More subtle than it looks!
- Ownership-based effects and interference
  - Synchronisation requirements inference
  - Automatic lock generation and allocation

In Retrospect

- Ownership types have gained a lot of attention
  - Even though no real popular uptake in PLs
    - Annotation burden
    - Overly restrictive type rules
  - Experimental language features should not be rushed into production
- We continue to learn more about how ownership concepts can be usefully deployed
- Need to combine ownership concepts with other related ideas
  - Separation logic
  - Regions

Key Ideas for Ownership

- Object ownership is determined at creation time
  - Just like object identity, but is programmer specified
  - Imposing object structure is a sensible thing to do
- Parametric ownership types gives reasonable flexibility
  - Need to integrate with parametric types better than OGJ
  - Need expressive constraint language for assumptions on type/ownership parameters
  - Want good choice of defaults and good inference algorithms to minimise annotation burden
- Different type rules can be used to achieve different kinds of ownership policies
  - Separation of object capability and reference accessibility
  - Need to be able to parametrise the type system for different policies for different types of objects
  - Ownership based effect systems offer the promise of more precise reasoning about effects than other kinds of systems