

On Ownership and Accessibility

Yi Lu

Joint work with John Potter

University of New South Wales
Sydney, Australia

Overview

Object-oriented programming

- Imperative
- Class level encapsulation

Overview

Object-oriented programming

- Imperative
- Class level encapsulation

Class level encapsulation is insufficient to localize/protect object instances

- References in private fields can be easily leaked
- **Representation exposure**

Overview

Object-oriented programming

- Imperative
- Class level encapsulation

Class level encapsulation is insufficient to localize/protect object instances

- References in private fields can be easily leaked
- **Representation exposure**

The problem: references are not constrained by lexical scope

Object encapsulation and ownership types

Object encapsulation

- An object **owns** (encapsulates) its representation
- Representation must not be referenced from outside its encapsulation

Ownership types enforce object encapsulation statically

- A dominator-tree-based encapsulation structure
 - Root is 'world'

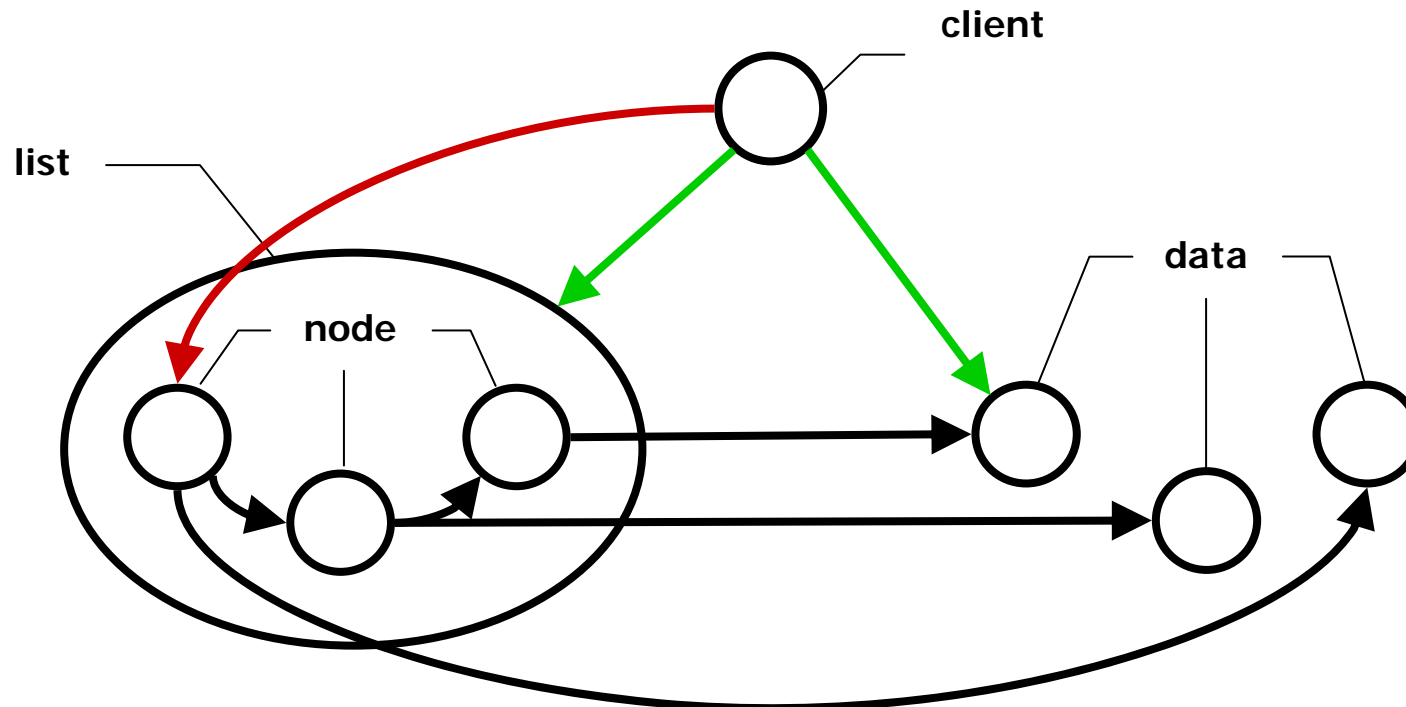
Containment Invariant:

- If $x \rightarrow y$ then $x \leq y.\text{owner}$
- A client must use the owner's interface to operate on its representation

A linked list 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

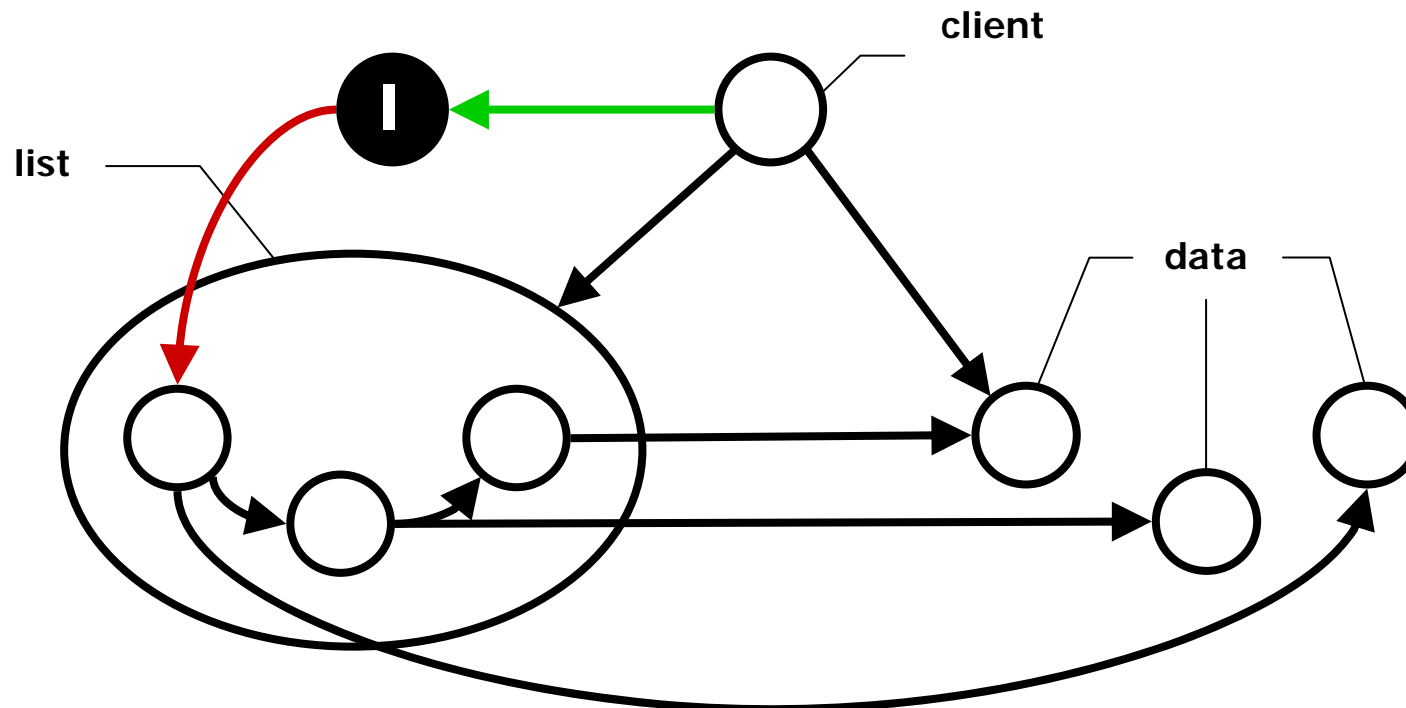


A linked list example

```
class List<o, d> { Node<this, d> head; ... }
```

A problem

- Where should we put an iterator?



Limitations of ownership types

Owners-as-dominators containment property is overly strong

- Cannot express some common design patterns such as iterators, callback objects (Aldrich and Chambers, ECOOP'04)

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**

Another limitation - inability to express owner variance

- Elements stored in list must all have same owner
- Clients must access all elements via that owner

Attempts

Previous attempts have various limitations:

- Universes (Müller and Poetzsch-Heffter, PLFP'99)
- Owner-rep model (Clarke, Potter and Noble, ECOOP'01)
- JOE (Clarke and Drossopoulou, OOPSLA'02)
- Inner classes (Boyapati, Liskov and Shriram, POPL'03)
- Ownership domains (Aldrich and Chambers, ECOOP'04)
- Immutable references (Birka and Ernst, OOPSLA'04'05)
- Effect encapsulation (Lu and Potter, POPL'06)

Our new proposal generalizes ownership types with an novel access control system

- More flexible and expressive

Our observation

Ownership types: $\text{Iterator}\langle o, d \rangle$

- o and d are capability of iterator objects
 - o is the capability to reference list's private nodes
 - d is the capability to reference the elements stored on the list

Accessibility = capability?

- The client has to state the *full capability* in order to type (then access) the iterators
- The client cannot type the iterators
 - Because the client cannot state the capability to reference list's internal

Our solution

Separate object **accessibility** from its reference capability

- By adding an **object accessibility modifier** to its type

Rationale:

- Client only needs to hold the capability of the access modifier for an object to use it
 - To type an object, a client must name its access modifier
- Reference capability can be abstracted
 - Via use-site type argument abstraction
 - To type an object, a client need only give an abstracted version of its type
 - Client does NOT have to name the object's owner or other capabilities

Object access modifiers

A typical type :

[**access**] C<capability list>

access is a *single* owner context

- Determines the object's accessibility
- NOT bound to the definition of the object

Old ownership containment invariant:

- If $x \rightarrow y$ then $x \leq y.\text{owner}$

New ownership **accessibility** invariant:

- If $x \rightarrow y$ then $x \leq y.\text{access}$
- Allow much more flexible reference structures

Field access modifiers v.s. object access modifiers

Traditional class-level field access modifiers

- Keywords (private/protected/public) restrict field access
- No protection on object instance

Instance-level object access modifiers

- Are runtime objects
- Determine the accessibility of object instances

Object access modifiers

```
class C<o> {  
    [world] D x;  
    [o] D y;  
    [this] D z; }
```

Objects in **x** are **public** to anyone

- Global accessibility

Objects in **y** are **protected** by owner of current object

- Only accessible from within its owner **o**

Objects in **z** are **private** to current object

- Only accessible from within the current object, i.e., **this**

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

```
➔ // 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
```

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

Implicit open/close

```
class Iterator<o, d> {  
    [o] Node<o, d> current;
```

```
// client code
```

```
[this] Iterator<*, world> iter;
```

```
iter.current // type is [?] Node<?, d>
```

To look up the type of a member, substitute * with ?

- this **opens** an object definition
- * - any possible owners (abstract)
- ? - one unknown owner (existential)

Type system avoids introducing new names for existential owners into environments

- By keeping them anonymous
- No need to **close** them
- Restricted kind of existential type

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

// 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
```

list

world



Other features

Bounded abstraction and existential owners

- o^+ : any owners inside o ($o^{+?}$: one unknown owner inside o)
- o^- : any owners outside o ($o^{-?}$: one unknown owner outside o)

Object-level read-only field

- Instead of traditional class-level read-only (final) field
- $[d]$ Node<this, d > head
- head field is readable from client (by abstracting this)
- head field is NOT updatable from client
- head field is both readable and updatable from inside the list

Allow accessibility to be reduced as computation proceeds

- $[a]$ C< o > $x = \text{new } [b]$ C< o >() is allowed given $a \leq b$
- Adds flexibility

Conclusion

Ownership types support information hiding by providing statically enforceable object encapsulation

Our type system generalizes ownership types by separating object accessibility from its reference capability

- Reference capability can be abstracted

It can significantly improve the expressiveness and utility of ownership types

Ownership types

Type systems:

- Flexible alias protection (Noble, Vitex and Potter, ECOOP'98)
- Ownership types (Clarke, Potter and Noble, OOPSLA'98, ECOOP'01)
- JOE (Clarke and Drossopoulou, OOPSLA'02)
- Inner classes (Boyapati, Liskov and Shriram, POPL'03)
- Ownership domains (Aldrich and Chambers, ECOOP'04)
- Permission-based ownership domains (Krishnaswami and Aldrich, PLDI'05)

Effect systems:

- Universes (Müller and Poetzsch-Heffter, PLFP'99)
- Javari (Birka and Ernst, OOPSLA'04, OOPSLA'05)
- Effect encapsulation (Lu and Potter, POPL'06)

Applications:

- Data race (Boyapati and Rinard, OOPSLA'01)
- Deadlock (Boyapati, Lee and Rinard, OOPSLA'02)
- Protect Javabeans (Clarke, Richmond and Noble, OOPSLA'03)
- Lazy modular upgrades (Boyapati et al., OOPSLA'03)
- Memory management (Boyapati et al., PLDI'03)