Ten Years of Ownership Types, or the benefits of Putting Objects into Boxes

Sophia Drossopoulou
Department of Computing, Imperial College London
We would like our surroundings* to be “tidy”

*surroundings = home, or desk, or code, or program heap, ....
This room is a mess!
No, it is not!
Everything is neatly categorised in its box!
A common problem in programming is that code/object topology is far too complex.
A common solution is to organize code/objects into “boxes”.

Over the last decade, several kinds of “boxes” have been suggested with different aims.

Some of this work has concentrated on static type systems.

We shall discuss:
- Survey some of the work on boxes
- The associated heap topology
- **MOJO**: the need for multiple boxes
Ten years of ownership types …


… and many, many others.
... it started, ten years ago, at ECOOP 1998 with a paper without implementation, without semantics.

Flexible Alias Protection

James Noble¹, Jan Vitek², and John Potter¹

¹ Microsoft Research Institute, Macquarie University, Sydney
   kjx.potter@nri.mq.edu.au
² Object Systems Group, Université de Genève, Geneva.
   Jan.Vitek@cui.unige.ch

Abstract. Aliasing is endemic in object oriented programming. Because an object can be modified via any alias, object oriented programs are hard to understand, maintain, and analyze. Flexible alias protection is a conceptual model of inter-object relationships which limits the visibility of changes via aliases, allowing objects to be aliased but mitigating the undesirable effects of aliasing. Flexible alias protection can be checked statically using programmer supplied aliasing modes and imposes no runtime overhead. Using flexible alias protection, programs can incorporate mutable objects, immutable values, and updatable collections of shared objects, in a natural object oriented programming style, while avoiding the problems caused by aliasing.

1 Introduction

I am who I am; I will be who I will be.

Object identity is the foundation of object oriented programming. Objects are useful for modelling application domain abstractions precisely because an object's identity always remains the same during the execution of a program —
... it started, ten years ago, at ECOOP 1998 with a paper without implementation, without semantics, but with very compelling diagrams.

Fig. 4. A Course uses a hashtable as part of its representation (dark grey) while Student and Lecturer objects are the course’s arguments (light grey). The hashtable also stores RawMark objects for each student, and these are arguments to the hashtable but part of the Course’s representation (mid grey), so cannot be accessed from outside the Course (dotted arrows).

tainer objects. Flexible alias encapsulation separates the objects within an ag-
... and then, at OOPSLA 1998

Ownership Types for Flexible Alias Protection

David G. Clarke, John M. Potter, James Noble

Microsoft Research Institute, Macquarie University, Sydney, Australia
{clad, potter, kjs}@nri.mq.edu.au

Abstract

Object-oriented programming languages allow inter-object

interactions that are not provided for in classical

programming models. The authors have proposed a

model of flexible alias protection, supported by illustrative

examples, and suggested incorporating aliasing modes into

programming languages. For flexible alias protection three

... a paper with a formal system

D. Clarke and A. Buckley then developed implementations ...
Survey - 1

Boxes for Package Encapsulation

Bokowski, Vitek, Grothof, Palsberg,...
Boxes for Package Encapsulation

- some classes declared confined within their package
- objects of confined type encapsulated within package

Therefore
- “box” is a package; static boxes
- **owner as dominator:** no incoming references to a box

Properties guaranteed statically
Boxes for Package Encapsulation

- some classes declared confined within their package
- objects of confined type encapsulated within package

Therefore
- “box” is a package; static boxes
- **owner as dominator**: no incoming references to a box

Properties guaranteed statically

```c
package P1 {
    class A{ ... }
    class B{ ... }
    confined class C{ ... }
}

package P2 {
    class D{ ... }
    confined class E{ ... }
}
```
Boxes for Package Encapsulation

- some classes declared confined within their package
- objects of confined type encapsulated within package

Therefore

- “box” is a package; static boxes
- **owner as dominator**: no incoming references to a box

Properties guaranteed statically

```java
package P1 {
    class A{ ... }
    class B{ ... }
    confined class C{ ... }
}
package P2 {
    class D{ ... }
    confined class E{ ... }
}
```

with a possible heap:
Boxes for Package Encapsulation

- some classes declared confined within their package
- objects of confined type encapsulated within package
Therefore
- “box” is a package; static boxes
- owner as dominator: no incoming references to a box

Properties guaranteed statically

```
package P1 {
    class A{ ... }
    class B{ ... }
    confined class C{ ... }
}
package P2 {
    class D{ ... }
    confined class E{ ... }
}
```

with a possible heap:
Boxes for Package Encapsulation

- some classes declared confined within their package
- objects of confined type encapsulated within package

Therefore

- “box” is a package; static boxes
- **owner as dominator**: no incoming references to a box

Properties guaranteed statically

Code from one package won’t run on confined objects from another.

```java
package P1 {
    class A { ... }
    class B { ... }
    confined class C { ... }
}
package P2 {
    class D { ... }
    confined class E { ... }
}
```

with a possible heap:
Survey - 2

Boxes for Object Encapsulation

Aldrich, Biddle, Boyapati, Chambers, Clarke, Drossopoulou, Khrishnaswami, Kostadinov, Liskov, Lu, Noble, Potanin, Potter, Vitek, Shrira, Wrigstad, ...
Boxes for Object Encapsulation

- Clarke, Noble, Potter, Vitek,..

• each object belongs in a box;
• each box is characterized by an object (its owner)
• objects may hold references to objects in enclosing boxes

Therefore
• tree hierarchy of objects
• owner as dominator: no incoming references to a box

Properties guaranteed statically

a possible heap:
Boxes for Object Encapsulation – An Example

An employee is responsible for a sequence of tasks. Each task has a duration and a due date.

When an employee is delayed, each of his tasks gets delayed accordingly.

An employee is OK, if all his tasks are within the due dates.

"Java" code

```java
class Employee {
    List tasks;
    void delay() { ... }
}

class List {
    Node first;
    void delay() { ... }
}

class Node {
    Node next;
    Task task;
    void delay() { ... }
}

class Task { ... 
    void delay() { ... } }
```
Boxes for Object Encapsulation - An Example

"Java" code

```java
class Employee {
    List tasks;
    void delay() { ... }
}
class List {
    Node first;
    void delay() { ... }
}
class Node {
    Node next;
    Task task;
    void delay() { ... }
}
class Task {
    ... 
    void delay() { ... }
}
```

possible heap

Boxes for Object Encapsulation – An Example

Employee “owns” his tasks, and the list.

The list “owns” its nodes.
Each object owned by another, eg 1 owns 2, 5, 6. Thus, classes have owner parameter, eg

```java
class List<o>{ ... }
```

and types mention owners, eg

```java
List<this>
```

Objects may have fields pointing to enclosing boxes, eg 3.

Classes have as many ownership parameters, as boxes involved

```java
class Node<o1,o2>{
    Node<o1,o2> next;
    Task<o2> task; .. }
```
Boxes for Object Encapsulation – An Example

“Java + OT” code

class Employee<o> {
    List<this> tasks;
    void delay( ){ ... } }
class List<o1>{
    Node<this,o1> first;
    void delay(){ ... } }
class Node<o1,o2>{
    Node<o1,o2> next;
    Task<o2> task;
    void delay(){ ... } }
class Task<o>{ ... 
    void delay(){ ... } }

with a possible heap:
Boxes for Object Encapsulation – An Example

class Employee<o> {
    List<this> tasks;
    void delay( ){ ... }
}
class List<o1>{
    Node<this,o1> first;
    void delay(){ ... }
}
class Node<o1,o2>{
    Node<o1,o2> next;
    Task<o2> task;
    void delay(){ ... }
}
class Task<o>{ ... void delay(){ ... } }

Employee “controls” its tasks; list controls its links.
Please turn the volume down.

This will not make my room any tidier!
radio.volumeDown() # room.TIDY()
We want to be able to argue for “different” employees $e_1, e_2$:
\[ e_1 \# e_2 \vdash e_1.\text{delay}() \# e_2.\text{OK}() \]

**Approach:** Boxes characterize the parts of heap affecting/ed by some execution/property.

For example:
- $1.\text{delay}() : 1.\text{under}$
- $7.\text{OK}() : 7.\text{under}$

Disjoint boxes $\Rightarrow$ independence
Boxes for Property Encapsulation – An Example

**Approach:** we add effects to methods:

```java
class Employee<o> { ...
    void delay() : this.under
}
class List<o1>{...
    void delay() : o1.under
}
class Node<o1,o2>{...
    void delay() : o2.under
}
class Task<o>{ ...  
    void delay() : o.under
}
```

Therefore, \( e_1 \text{.delay()} : e_1.\under \)
\( e_2 \text{.OK()} : e_2.\under \)

Because \( e_1 \parallel e_2 \models e_1.\under \# e_2.\under \)
we have \( e_1 \parallel e_2 \models e_1.\text{delay()} \# e_2.\text{OK()} \)
Boxes for Scoped Memory
Zhao, Noble, Vitek, Sacianu, Boyapati, Beebee, Rinard
Exploit owners as dominators property, to reclaim whole memory areas rather than individual objects, in presence of multithreading

Here, 2, 3, and 4 belong in one memory scope and reclaimed together. Then, 1, 5 and 6 belong to the parent memory scope.

Memory areas organized hierarchically. Threads enter/leave memory scopes consistent with the hierarchy.

Scoped memory used in unmanned airplanes (Vitek & Noble)
Survey - 3

Boxes for Concurrency
Boyapati, Lee, Liskov, Rinard, Salcianu, Shrira, Whaley, ...

and also
Abadi, Flanagan, Freund, Qadeer,

and also
Cunningham, Eisenbach, Drossopoulou
Boxes for Concurrency

To avoid races/guarantee atomicity, a thread must have acquired the lock to an object before accessing it. The owner of a box stands for the lock of all the contained objects.

A thread must lock 1 before accessing 1, 5, or 6 – ie **no need to lock objects individually**.

Threads must lock 2 before accessing 2, 3, or 4.

Note

- no nesting of boxes
- owners **not** dominators
- owners as locks.
Survey - 4

Boxes for Program Verification

Barnett, Bannerjee, Darvas, DeLine, Dietl, Faehndrich, Jacobs, Leavens, Leino, Logozzo, Mueller, Naumann, Parkinson, Piessens, Poetzsch-Heffter, Schulte ...
Boxes for Verification

An object “owns” other objects; the owner’s invariant depends on the properties of the owned object.

A company is OK, if all its employees are OK. An employee is OK, if all his tasks are on time.

Note:

- owners may change; (5 may move to 7)
- no owners as dominators; (3 may have reference to 9)
- owner as modifier (3 may not change 9)
Survey - 5

Boxes for Program Architecture

Chambers, Aldrich, Krisnaswami, ....
Boxes as Ownership Domains

Objects “own” boxes. Link statements allow references across boxes

A Bank has several branches and an archive. A branch has tellers and a vault. Customers are allowed access to the tellers, but not the vaults.

class Bank<b> {  
domain branches, archive;  
Branch<branches> br1, br2;  
Data<archive> d1, d2, d3;  
link b.customer & branches...  
}  
class Branch<b1>{  
domain vault, tellers;  
Safe<vault> s1, s2;  
Cahier<teller> c1, c3;  
link b1 & teller;  
}  
class Main<b> {  
domain customer, shop;  
Bank<shop> b1;  
Person<customer> p1, p2 } ;
class Bank<b> { 
    domain branches, archive;
    Branch<branches> br1, br2;
    Data<archive> d1, d2, d3;
    link b.customer & branches b1 & teller;
}

class Branch<b1> {
    domain vault, tellers;
    Safe<vault> s1, s2;
    Cashier<teller> c1, c2; 
    link
}

;
Aldrich et al. have developed tools with extract such architectural descriptions from the program code.
Survey – Summary
<table>
<thead>
<tr>
<th>Confined types</th>
<th>Box is a</th>
<th>M/D</th>
<th>nest dpth</th>
<th>TR</th>
<th>Obj has ? boxes</th>
<th>Obj in boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>package</td>
<td>OAD</td>
<td>1</td>
<td>no</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object Encapsulation</th>
<th>Box is a</th>
<th>M/D</th>
<th>nest dpth</th>
<th>TR</th>
<th>Obj has ? boxes</th>
<th>Obj in boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>object</td>
<td>OAD</td>
<td>n</td>
<td>some</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Locking</th>
<th>Box is a</th>
<th>M/D</th>
<th>nest dpth</th>
<th>TR</th>
<th>Obj has ? boxes</th>
<th>Obj in boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locking</td>
<td>object or thread</td>
<td>none</td>
<td>n</td>
<td>no</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Universes Boogie</th>
<th>Box is a</th>
<th>M/D</th>
<th>nest dpth</th>
<th>TR</th>
<th>Obj has ? boxes</th>
<th>Obj in boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universes Boogie</td>
<td>object</td>
<td>OAM</td>
<td>n</td>
<td>yes</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ownership domain</th>
<th>Box is a</th>
<th>M/D</th>
<th>nest dpth</th>
<th>TR</th>
<th>Obj has ? boxes</th>
<th>Obj in boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ownership domain</td>
<td>“object”</td>
<td>none</td>
<td>n</td>
<td>no</td>
<td>n</td>
<td>1</td>
</tr>
</tbody>
</table>

Where OAM = owners as modifier; OAD = owner as dominator; TR = ownership transfer
However ...
The nano is mine

No, it is mine

OK, let us share it!
Common Ownership - The Classic Way

Put the nano in the most enclosing inner box.

class Family{o} {...
iPod<this> nano;
Daughter<this> nicky;
Parent<this> sophia;
...}

then:
Common Ownership - The Classic Way - Limitations

However, the family also includes athena and constantine. Therefore, they too will get their hands on the nano....
Give sophia a readonly reference to the nano.

class Daughter {...
    rep iPod nano;
    ...
}

class Parent {...
    readonly iPod nano;
    ...
}

then, sophia can listen to the nano.
Common Ownership - the Universes Way - Limitations

However, then, sophia cannot switch the nano on or off!

nicky: Daughter

nano: iPod

readonly

sophia: Parent
**Common Ownership – Ownership Domains Way**

Put *sophia* and *nicky* in the same ownership domain, with access to the domain containing *nano*.

```java
class Daughter { ... }  
class Parent { ... }  
class Together {  
    public domain people;  
    domain music;  
    link people->music;  
    people Daughter nicky;  
    people Parent sophia;  
    music iPod nano;  }  
```

then, only *sophia* and *nicky* can manipulate *nano*.
However, what if sophia wanted to
• share the nano with nicky, and also
• share the walkman with constantine?
We developed

**MOJO, Multiple Ownership for Java Objects**
Multiple Ownership -- OOPSLA 2007

- allow more than one hierarchy
- allow more than one owner

Multiple Ownership

Nicholas R Cameron,
Sophia Dressopoulou
Department of Computing,
Imperial College London, UK
{ncameron, sd}@doc.ic.ac.uk

James Noble *
School of Mathematics, Statistics
& Computer Science,
Victoria University of Wellington,
New Zealand
kjp@mcs.vuw.ac.nz

Matthew J Smith
Department of Computing,
Imperial College London, UK
mjs198@doc.ic.ac.uk

Abstract
Existing ownership type systems require objects to have precisely one primary owner, organizing the heap into an ownership tree. Unfortunately, a tree structure is too restrictive for many programs, and prevents many common design patterns where multiple objects interact.

1. Introduction

We're tired of trees... We should stop believing in trees, roots, and radicles.

Deleuze and Guattari, A Thousand Plateaus [17]
Multiple Ownership – An Example

Tasks and employees as before.

A project consists of a sequence of tasks.

When a project is delayed, its tasks get delayed accordingly.

A project is OK, if all its tasks are within their due dates.

In the code we omit Node class.

“Java” code

```java
class Employee {
    EList tasks;
    void delay( ) { ... } }

class Project {
    List tasks;
    void delay( ) { ... } }

class List {
    List next;
    Task task;
    void delay( ) { ... } }

class Task {
    ...
    void delay() { ... }; }
```
Multiple Ownership – An Example

```java
class Employee {
    List tasks;
    void delay() { ... }
}
class Project {
    List tasks;
    void delay() { ... }
}
class List {
    List next;
    Task task;
    void delay() { ... }
}
class Task {
    ...
    void delay() { ... }
}
```

We want:

```
e1 # e2 ⊨ e1.delay() # e2.OK()
p1 # p2 ⊨ p1.delay() # p2.OK()
```
Need to express that a task belongs to an employee and a project, e.g.

```
1:E
5: T
11: P
```

Task 5 is owned by Employee 1, and Project 11. Here, Task< 1&11 >

In general, we allow types like

```
A<o1&o2, o3, o5&o6>
```

or

```
A<o1&any, o3, any>
```

In a type, we say any, when actual owner unknown (cf readonly).
Multiple Ownership

class Employee<o> {
    List<this, this> tasks;
    void delay() { ... } }

class Project<o> {
    List<this, this> tasks;
    void delay() { ... } }

class List<o1, o2> {
    List<o1, o2> next;
    Task<o1, o2 & any> task;
    void delay() { ... } }

class Task<o1> { ... 
    void delay() { ... } ; }

NOTE: 😊 Task class unaware of number of owners. 😊
NOTE: 😞 List class aware of number of owners. 😞
**BEGIN ASIDE:** the meaning of *any*

*any* = corresponding owner is unknown, but fixed;

We have to distinguish the *don’t know* from the *don’t care* use of any.

```java
class List<o1,o2> {
    ...
    List<o1,o2> next;
}
...
List<o4,o5&any> l1;       // any as don’t care
```
BEGIN ASIDE: the meaning of any
any = corresponding owner is unknown, but fixed;
We have to distinguish the don’t know from the don’t care use of any.

class List<o1,o2> {

  ...
  List<o1,o2> next;
}

...
List<o4,o5&any> l1; // any as don’t care
l1 = new List<o4,o5&o6>; // OK
l1 = new List<o4,o5&o7&o8>; // OK

l1.next; // any as don’t know

l1.next:= new List<o4,o5&o6>; : ERROR
l1.next:= l1; : ERROR
To distinguish the don’t care from the don’t know, we employ different field lookup functions upon field read and upon field write

For field read

\[
\begin{align*}
\Gamma &\vdash e : t \\
fType(t, f, e, \Gamma) & = t' \\
\Gamma &\vdash e.f : t'
\end{align*}
\]

\[fType(c<\overline{Q}>, f, e, \Gamma) = [\overline{Q/p}](t^{\Gamma,e}) \quad \text{where } t = fType^{aux}(c<\overline{p}>, f)\]

gives \[\texttt{ll.next;} : \texttt{List<04,05&any>}\]
To distinguish the don’t care from the don’t know, we employ different field lookup functions upon field read and upon field write,

For field read

\[
\text{....}
gives \quad l1.\text{next}; : \text{List}<\text{o4},\text{o5}&\text{any}>\]

On the other hand, for field write

\[
\Gamma \vdash e : t \\
f\text{Type}^{\text{strict}}(t, f, e, \Gamma) = t' \\
\Gamma \vdash e' : t' \\
\Gamma \vdash e.f = e' : t'
\]

\[
f\text{Type}^{\text{strict}}(c<\overline{Q}>, f, e, \Gamma) = [\overline{Q}/p]^{\text{strict}}t', \quad \text{where} \quad t = f\text{Type}^{\text{aux}}(c<\overline{P}>, f) \quad \text{and} \quad t' = t^e
\]

gives \quad l1.\text{next} := \text{new List}<\text{o4},\text{o5}&\text{o6}>; : \text{ERROR}

\text{END ASIDE}
We want to be able to argue:

\( e_1 \parallel e_2 \vdash e_1 \text{delay}() \# e_2 \text{OK}() \)

We first define when an object is “inside” another object, i.e. \( \iota \ll \iota' \) as the minimal reflexive, transitive relation, such that if one of the owners of \( \iota \) is \( \iota' \) then \( \iota \ll \iota' \)

Therefore

\[
\begin{align*}
1: \text{E} & \ll 6: \text{E} \\
2: \text{EL} & \ll 3: \text{EL} \\
4: \text{T} & \ll 5: \text{T} \\
9: \text{PL} & \ll 10: \text{PL} \\
11: \text{P} & \ll 5 \\
5 & \ll 5 \\
5 & \ll 1 \\
5 & \ll 11
\end{align*}
\]
Define run-time effects: $\text{χ ::= } \iota \mid \text{χ.undr} \mid \text{χ & χ}$

meaning:

$[[\iota]] = \{\iota\}$

$[[\text{χ.undr}]] = \{\iota \mid \iota \in [[\text{χ}]]\}$

$[[\text{χ & χ'}]] = [[\text{χ}]] \cap [[\text{χ'}]]$

$[[1]] = \{1\}$

$[[1.\text{under}]] = \{1, 2, 3, 4, 5\}$

$[[1.\text{under} \& 11.\text{under}]] = \{5\}$

$[[1.\text{under} \& 6.\text{under}]] = \emptyset$
Define also an effects annotation system, which gives

class Employee<o> {  
    List<this>this tasks; void  
    void delay()this&any.undr {...}  
}
class Project<o> {  
    ...  
    void delay() this&any.undr {...}  
}
class List<o1,o2> {  
    ...  
    void delay() o2.undr {...}  
}
class Task<o>{ ...  
    void delay() this.undr{..}  
}

For stack s and heap h, define \( [[\phi]]_{s,h} \) the obvious way.
Define the judgement $\Gamma \vdash \phi \ # \ \phi'$ to denote disjointness of effects.

**Lemma:**

$$\Gamma \vdash s, h \quad \Gamma \vdash \phi \ # \ \phi' \quad \Rightarrow \quad [[\phi]]_{s,h} \cap [[\phi']]_{s,h} = \emptyset$$

Execution of an expression does not require/modify more than what is described by the read/write effects:

**Theorem:**

$$\Gamma \vdash_{rd} e : \phi_1 \quad \Gamma \vdash_{wr} e : \phi_2$$

$$\Gamma \vdash s, h$$

$$e, s, h \sim v, h'$$

$$\begin{cases}
    h = [[\phi_1]]_{s,h} \ast h_2 \\
    [[\phi_1]]_{s,h} = [[\phi_2]]_{s,h} \ast h_3 \\
    h' = h'' \ast h_3 \ast h_2 \\
    e, s, [[\phi_2]]_{s,h} \ast h_3 \sim v, h'' \ast h_3 \\
    \text{for some } h_2, h_3, h''
\end{cases}$$
Thus, \[ e_1.\text{delay}() : \ (e_1\&\text{any}).\text{under} \]
\[ e_2.\text{OK}() : \ (e_2\&\text{any}).\text{under} \]

Because \[ e_1 \neq e_2 \vdash (e_1\&\text{any}).\text{under} \neq (e_2\&\text{any}).\text{under} \]

we have \[ e_1 \neq e_2 \vdash e_1.\text{delay}() \neq e_2.\text{OK}() \]

Similarly, \[ p_1 \neq p_2 \vdash p_1.\text{delay}() \neq p_2.\text{OK}() \]

😊
Can I preserve owners as dominators?

Yes, in a way, if we

- require that in each type definition the actual owner parameters are “within” the actual context parameters,
- define a program “slice”, $P_i$, where each class as a “selected” ownership parameter out of the may ownership parameters.
- For each slice, we filter the heap, by dropping any field whose selected owner is not “outside” the selected owner parameter of the defining class.
Can I preserve owners as dominators? yes, partly

Yes, in a way, if we

• require that in each type definition the actual owner parameters are “within” the actual context parameters,

• define a program “slice”, $Pi$, where each class as a “selected” ownership parameter out of the may ownership parameters.

• For each slice, we filter the heap, by dropping any field whose selected owner is not “outside” the selected owner parameter of the defining class.

Then

• For each of the slices, the selected owners are dominators in the correspondingly filtered heap.
Preserving owners as dominators – partly – P1 slice

Selected owner highlighted,

class Task<o1,o2> { ... }  
class Employee<o> {   
    EList<this> tasks;  
.. }  
class EList<o> {   
    EList<o> next;   
    Task<o,any> task;   
.. }  
class Project<o> {   
    PList<this> tasks; ... }  
class PList<o> {   
    PList<o> next;   
    Task<any,o> task;   
.. }
Preserving owners as dominators - partly - P1 slice

Selected owner highlighted, // and fields filtered out

class Task<o1,o2>{ ... }
class Employee<o> { 
  EList<this> tasks;
  .. } 
class EList<o> { 
  EList<o> next;
  Task<o,any> task;
  ... } 
class Project<o> { 
  PList<this> tasks; ... } 
class PList<o> { 
  PList<o> next; 
  // Task<any,o> task;
  ... }
Preserving owners as dominators – partly – P2 slice

Selected owner highlighted

class Task<o1,o2:>{ ... }  
class Employee:o: {    
  EList<this:> tasks;  
.. }  
class EList:o: {    
  EList:o: next;    
  Task<o,any:> task;  
... }  
class Project:o: {    
  PList<this:> tasks; ... }  
class PList:o: {    
  PList:o: next;    
  Task<any,o:> task;  
... }
Preserving owners as dominators – partly – P2 slice

Selected owner highlighted,

// and fields filtered out

class Task<o1,o2> { ... } 

class Employee<o> { 
    EList<this> tasks;
    .. } 

class EList<o> { 
    EList<o> next;
    // Task<o,any> task;
    .. } 

class Project<o> { 
    PList<this> tasks; ... } 

class PList<o> { 
    PList<o> next;
    Task<any,o> task;
    .. }
Multiple Owners and Aspects

Aside: I started tackling this problem (independence of actions and assertions in the presence of “overlapping topologies”) unsuccessfully by filtering out fields in and off for the three years. Multiple owners was the missing link, and in particular the idea of intersection - remember basic set theory?

Looking for an AOP view, where
the program is
\[ P = P_1 \oplus P_2 \oplus \ldots \oplus P_n \]
the heap is
\[ h = h_1 \oplus \ldots \oplus h_n \]
and execution of \( P \) consists of the combination of execution of \( P_1, P_2, \ldots, P_n \), and preserves some of the properties established in the context of \( P_i \).

\[ h_1 \oplus h_2 = h_0 \ast h_3 \ast h_4 \quad \text{where} \quad h_1 = h_0 \ast h_3 \quad \text{and} \quad h_2 = h_0 \ast h_4 \]
Multiple Ownership - Conclusions

- multiple owners are possible,
- multiple owners describe realistic object topologies, and thus document programmer’s intuitions,
- multiple owners can be used to argue disjointness.

Multiple Ownership - Further Work

- refine type system (*any* as existential, refine scope),
- apply to concurrency and verification,
- AOP: combine two programs into one program with multiple ownership hierarchies.
Putting MOJO into Context
<table>
<thead>
<tr>
<th>Confined types</th>
<th>M/D</th>
<th>nest dpth</th>
<th>TR</th>
<th>Obj has boxes</th>
<th>Obj in boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>package</td>
<td>OAD</td>
<td>1</td>
<td>no</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>object</td>
<td>OAD</td>
<td>n</td>
<td>some</td>
<td>1</td>
</tr>
<tr>
<td>Object Encapsulation</td>
<td>object</td>
<td>OAD</td>
<td>n</td>
<td>no</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>object or thread</td>
<td>none</td>
<td>n</td>
<td>no</td>
<td>1</td>
</tr>
<tr>
<td>Universe Boogie</td>
<td>object</td>
<td>OAM</td>
<td>n</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>Ownership domains</td>
<td>“object”</td>
<td>none</td>
<td>n</td>
<td>no</td>
<td>n</td>
</tr>
<tr>
<td>MOJO</td>
<td>“object”</td>
<td>none</td>
<td>n</td>
<td>no</td>
<td>1</td>
</tr>
</tbody>
</table>

where OAM = owners as modifier; OAD = owner as dominator; TR = ownership transfer
Conclusions

- “boxes” express and preserve a topology in the object heap;
- topology exploited for different goals, eg encapsulation, memory management, program verification, concurrency
- different goals impose slightly different constraints and notations - a unification would be nice (pluggable types).
- notation heavy in some cases; some nice simplifications exist, more are currently being developed.
- type inference exists for some systems, more would be good.
... not bad for a paper without implementation, without semantics, but with compelling diagrams,

Flexible Alias Protection

James Noble\textsuperscript{1}, Jan Vitek\textsuperscript{2}, and John Potter\textsuperscript{1}

\textsuperscript{1} Microsoft Research Institute, Macquarie University, Sydney
kjr.potter@mri.mq.edu.au
\textsuperscript{2} Object Systems Group, Université de Genève, Geneva.
Jan.Vitek@cui.unige.ch

Abstract. Aliasing is endemic in object oriented programming. Because an object can be modified via any alias, object oriented programs are

Ten years, later, we have many implementations, many semantics, vibrant research, diverse application areas, many further publications (eg 3 in ECOOP 07, 3 in OOPSLA 07), and recognition (J. Aldrich awarded the Junior Dahl/Nygaard Prize in 2007).
Thank you!