The Benefits of Putting Objects into Boxes - using pencil and paper -

Sophia Drossopoulou
Department of Computing, Imperial College London
... 1972, Scientific Gymnasium, Athens, 20 boys, 4 girls

... trying to decide what to study,

... came across basic set theory, where

\[(P \times Q) + R = (P + R) \times (Q + R)\]

... found it intriguing, because, eg

\[(2 \times 1) + 5 \neq (2 + 5) \times (1 + 5)\]

... so, settled on “Informatik”
1973, Fakultätfuer Informatik, Univ. Karlsruhe
10 Professors (age 35-45), no women,
40 academic staff, 4 women
   very enthusiastic, none knew much computer science
600 students, 40% women

Professor Krueger:
“computer science will be golden opportunity for women
because
1) women are more precise,
2) they will be able to work from home”
The “Rechenzentrum” had
IBM 360/370
Burroughs 9000
PDP 9/11
Univac 1108

<table>
<thead>
<tr>
<th>used punched cards ...</th>
<th>it was very slow ....</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Image of punched card]</td>
<td>[Image of people working]</td>
</tr>
</tbody>
</table>

pencil and paper much faster...
1979 obtained Diploma
joined Prof. Goos group - 20 people and 2 women;
worked on Modula-2 and Ada compilers;
attribute grammars

1982 PhD, Karlsruhe
work on parser generators theory; theory lead
to new speedups
(30th PhD of Department,
10th foreigner, 1st woman)
1986 Department of Computing, Imperial College
roughly 30 academic staff,
of which 1 woman senior lecturer,
3 women lecturer

worked on the Flagship project on functional
programming; and discovered the joys of object
oriented programming
1986 Department of Computing, Imperial College roughly 30 academic staff, of which 1 woman senior lecturer, 3 women lecturers

and 20 years later,
  22 Professors (of which 1 woman)
    6 Readers (of which 3 women)
  15 Senior Lecturers (of which 6 women)
  10 Lecturers (of which 1 woman)
In those 20 years, I worked on

- Separate Compilation and Dynamic Linking
- Semantics of Java
- Dependent Classes
- Ownership Types
- Traits
- Generic Algorithms and Combinatorial Optimization
- Object Reclassification
- Types for Scripting Languages

and family Constantine, Sophia, Athena, Nicky ....
This room is a mess!
No, it is not! Everything is neatly categorised in its box!
A common problem in programming is that code structure/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 (4 strands),

• One further issue on boxes.
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

```java
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

```
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

```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
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:

```
1A  2A  3B  4D
\  /  \  /  \\
4C--5C--8E
  |         |
  |         |
P1  P2
```
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
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

![Heap Diagram]
Boxes for Object Encapsulation – An Example

Employee “owns” his tasks, and the list.

The list “owns” its nodes.

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

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()
Boxes for Object/Property Encapsulation
Clarke, Drossopoulou, Smith

We want to be able to argue for “different” employees $e_1, e_2$:

$$e_1 \not\equiv e_2 \vdash e_1\.delay() \# e_2\.OK()$$

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

For example:

1. delay() : 1.under
7. OK() : 7.under

Disjoint boxes $\Rightarrow$ independence
Boxes for Object/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,

- `e1.delay() : e1.under`
- `e2.OK() : e2.under`

Because `e1 ≠ e2 ⊢ e1.under # e2.under`

we have `e1 # e2 ⊢ e1.delay() # e2.OK()`
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.
Survey - 3

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

and also
Abadi, Flanagan, Freund, Qadeer, ...
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 - Summary
<table>
<thead>
<tr>
<th></th>
<th>owner is …</th>
<th>owner as dominator?</th>
<th>benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Confined types</strong></td>
<td>package - <em>static</em> number of owners</td>
<td>yes</td>
<td>object encapsulated in package</td>
</tr>
<tr>
<td><strong>Object Encapsulation</strong></td>
<td>an object</td>
<td>yes</td>
<td>object encapsulated in objects, scoped memory, visualization, independence</td>
</tr>
<tr>
<td><strong>Locking</strong></td>
<td>object or thread, holds “logic lock” to owned objects</td>
<td>no; no nesting</td>
<td>guarantee race-free, or atomicity</td>
</tr>
<tr>
<td><strong>Universes/Boogie</strong></td>
<td>an object; owner’s properties depend on owned objects’ state – owners may change</td>
<td>no; modifier owners may change</td>
<td>modular verification</td>
</tr>
</tbody>
</table>
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:

- nicky: Daughter
- sophia: Parent
- nano : iPod
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.

```java
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!
Common Ownership - Ownership Domains Way

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

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.
Common Ownership - Ownership Domains Way - Limitations

However, what if sophia wanted to
• share the nano with nicky, and also
• share the walkman with constantine?

nicky: Daughter
sophia: Parent
constantine: Parent
nano : iPod
walkman: Sony
people : Together
music
We explore

**Multiple Ownership**

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

Cameron, Drossopoulou, Smith, Noble
OOPSLA 2007
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 \parallel e2 \vdash e1.delay() \# e2.OK()
p1 \parallel p2 \vdash p1.delay() \# p2.OK()
```
Need to express that a task belongs to an employee \textit{and} a project, e.g.

![Diagram showing task 5 owned by Employee 1 and Project 11]

Task 5 is owned by Employee 1, \textit{and} Project 11.
Here, Task\(< 1\&11 >\)

In general, we allow types like

\[
A<01\&02,03,05\&06>
\]

or

\[
A<01\&\text{any},03,\text{any}>
\]

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

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

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

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

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

😊 NOTE: List class unaware of number of owners. 😊
The meaning of \texttt{any}: the corresponding owner is unknown, but fixed.

\begin{verbatim}
class List\langle o1, o2 \rangle {
    ...
    List\langle o1, o2 \rangle next;
}
...
List\langle o4, o5 & any \rangle l1;

l1 = new List\langle o4, o5 & o6 \rangle; : OK
l1 = new List\langle o4, o5 & o7 \rangle; : OK
l1 = new List\langle o4, o5 & o7 & o8 \rangle; : OK
l1.next; : is a List\langle o4, o5 & any \rangle
l1.next.next.next; : is a List\langle o4, o5 & any \rangle
l1.next := new List\langle o4, o5 & o6 \rangle; : ERROR
l1.next := l1; : ERROR
\end{verbatim}
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

\[
\text{if one of the owners of } \iota \text{ is } \iota' \text{ then } \iota \ll \iota'
\]

Therefore

\[
\begin{align*}
1 &: E \\
2 &: EL \\
3 &: EL \\
4 &: T \\
5 &: T \\
6 &: E \\
7 &: EL \\
8 &: T \\
9 &: PL \\
10 &: PL \\
11 &: P
\end{align*}
\]

\[
5 \ll 5 \\
5 \ll 1 \\
5 \ll 11
\]
Define run-time effects: $X ::= \iota \mid X.\text{undr} \mid X \& X$

meaning:

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

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

$$[[X \& X']] = [[X]] \cap [[X']]$$

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

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

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

class Employee<o> {
    List<this,this&any> tasks;
    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 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:

$\Gamma \vdash_{rd} e : \phi_1 \quad \Gamma \vdash_{wr} e : \phi_2$
$\Gamma \vdash s, h$
$e, s, h \sim v, h'$

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

Because \[ e_1 \not\parallel e_2 \vdash (e_1 \& \text{any}).\text{under} \# (e_2 \& \text{any}).\text{under} \]

we have \[ e_1 \not\parallel e_2 \vdash e_1 \text{delay()} \# e_2 \text{OK()} \]

Similarly, \[ p_1 \not\parallel p_2 \vdash p_1 \text{delay()} \# p_2 \text{OK()} \]
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", $P_i$, 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.

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
\[ P = P_1 \oplus P_2 \oplus ... \oplus P_n \]
the heap is
\[ h = h_1 \oplus ... \oplus h_n \]
and execution of \( P \) consists of the combination of execution of \( P_1, P_2, ..., P_n \), and preserves some of the properties established in the context of \( P_i \).

\[ h_1 \oplus h_2 = h_0 \ast h_3 \ast h_4 \]
where \( h_1 = h_0 \ast h_3 \) and \( h_2 = h_0 \ast h_4 \)
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.

Watch [http://slurp.doc.ic.ac.uk/](http://slurp.doc.ic.ac.uk/) and OOPSLA 2007 for the paper
The Benefits of Putting Objects into Boxes

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.
Things that have worked for me

• enjoying the work,
• choose what to work on,
• naiveté,
• work on several things in parallel,
• opportunity to revisit a question,
• collaborations,
• teaching,

and also

• family,
• two “blue eyed” decisions
Thank you!