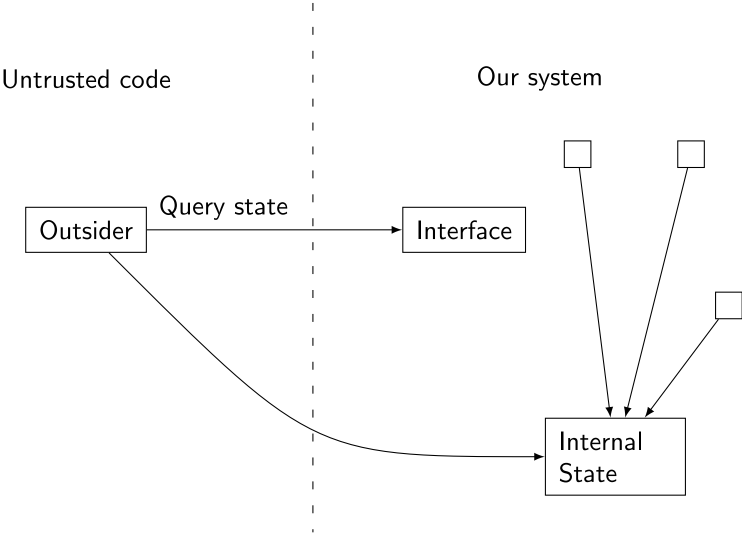
Types for Deep/Shallow Cloning

Ka Wai Cheng

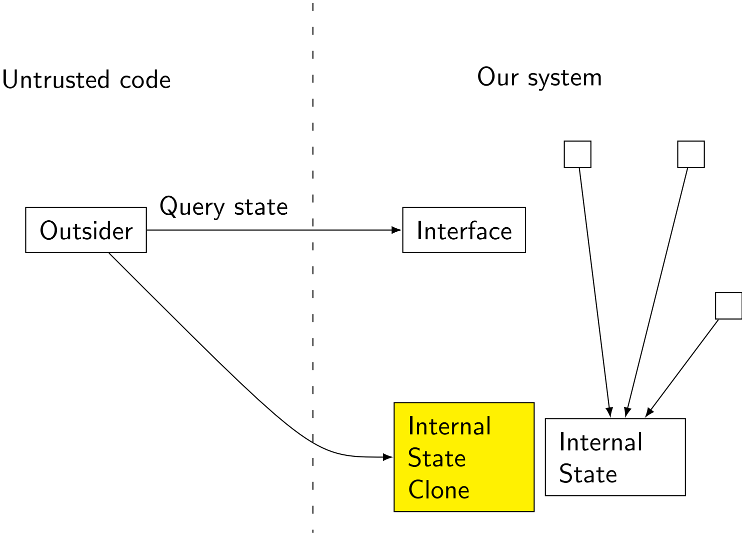
Imperial College London
Department of Computing

June 26, 2012

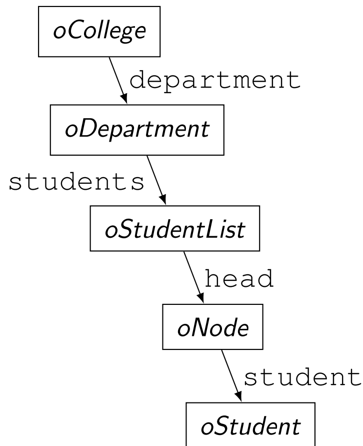
Motivation



Motivation



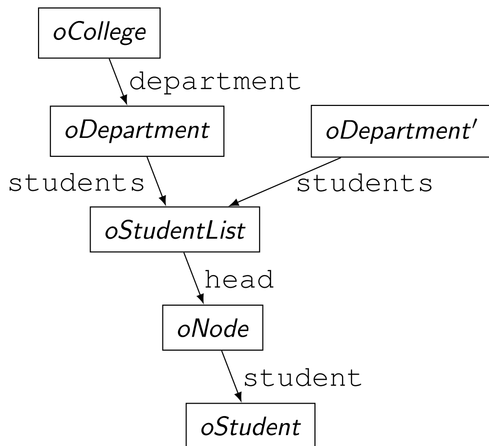
Shallow & Deep Cloning



Cloning in Java:

- Shallow cloning:
  default `clone()` method
- Deep cloning:
  serialization
- In between shallow & deep cloning:
  custom `clone()` implementation

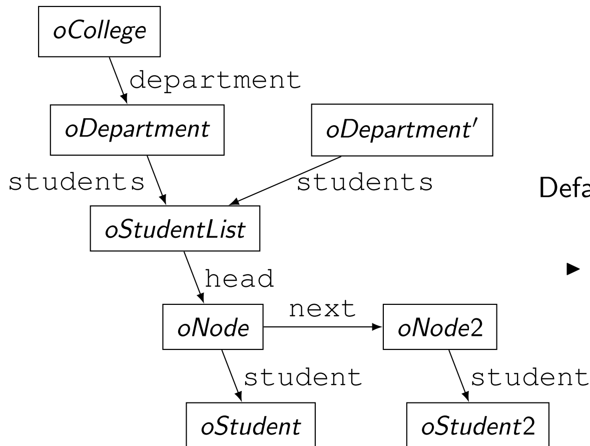
Shallow Clone of *oDepartment*



Default `clone()` method

- Clone too little

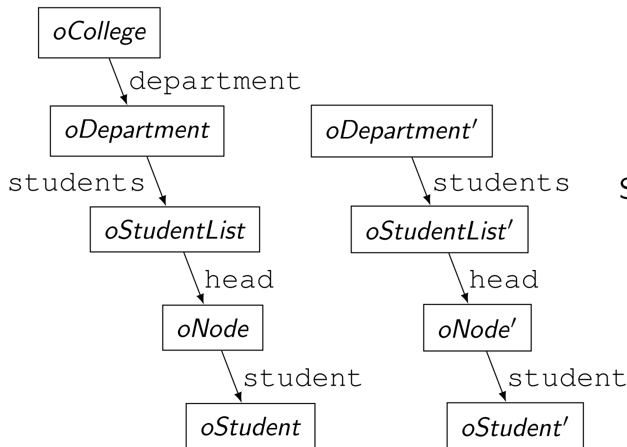
Shallow Clone of *oDepartment*



Default `clone()` method

- Clone too little

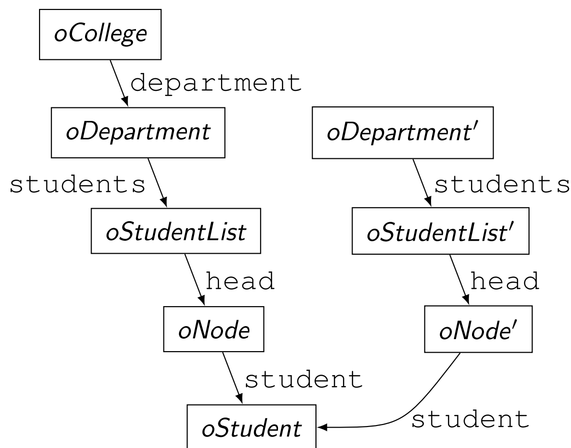
Deep Clone of *oDepartment*



Serialization

- Clone too much

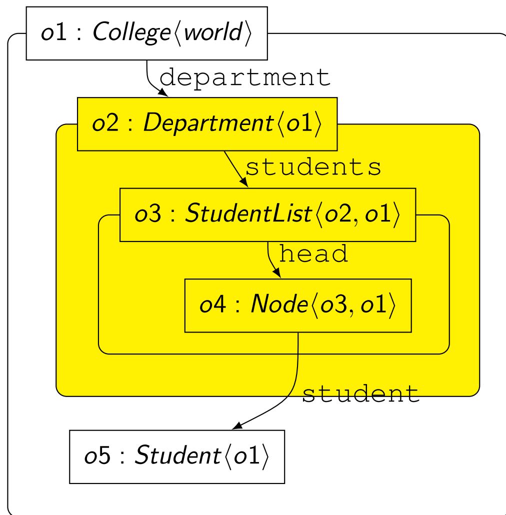
Custom Clone of *oDepartment*



Custom `clone()` method

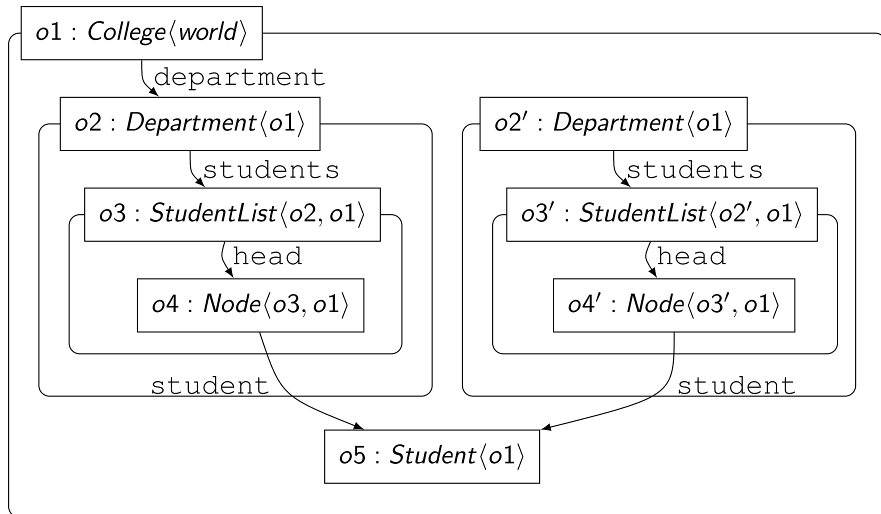
- Programmer's responsibility
- Tedious
- Error prone

Ownership Types




```
class College <c> {
  Department <this>
      department;
}

class Department <c> {
  StudentList <this,c>
      students;
}

class StudentList <c1,c2> {
  Node <this,c2> head;
}

class Node <c1,c2> {
  Student <c2> student;
  Node <c1,c2> next;
}

class Student <c> {}
```

Ownership Types & Sheep Cloning

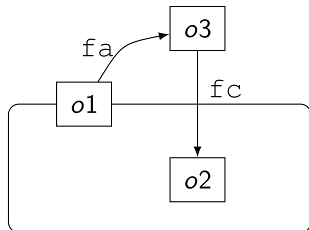


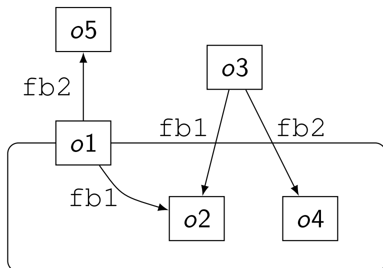
Deriving Cloning Methods (Proposed in *Trust the Clones*)

```
class Node/*<c1,c2>*/ {
  Student/*<c2>*/ student;
  Node/*<c1,c2>*/ next;

  Node clone(){
    this.clone(false, false, new IdentityHashMap());
  }

  Node clone(Boolean s1,Boolean s2,Map m){
    Object n = m.get(this);
    if (n != null) {
      return (Node)n;
    }
    Node clone = new Node();
    m.put(this,clone);
    clone.next= s1
      ? this.next.clone(s1,s2,m) : this.next;
    clone.student= s2
      ? this.student.clone(s2,m) : this.student;
    return clone;
  }
}
```

Contributions

- Implementation
- Problem of & solution to cloning without Owners-as-Dominators

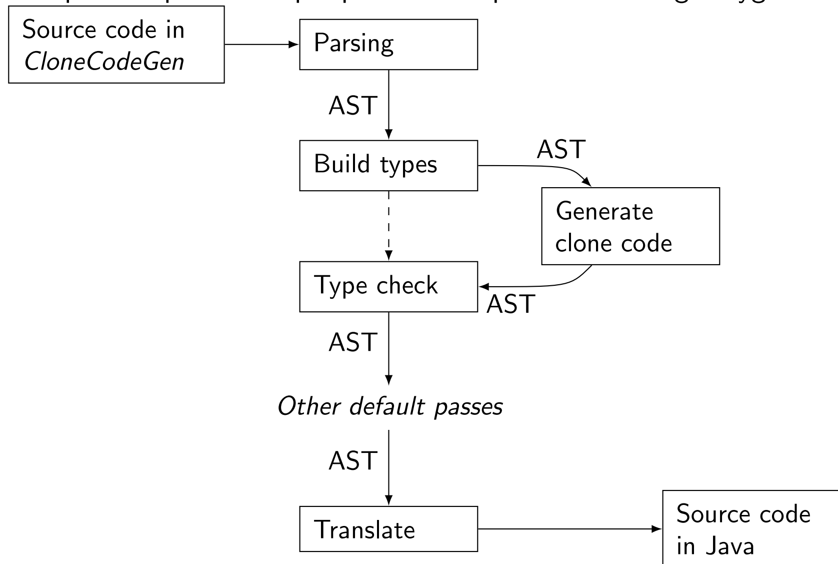




- Extension for arrays
- Extension for subclassing
- Extension for generics

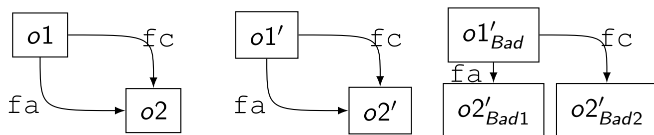
Implementation - Java language Extension, *CloneCodeGen*

Compilation process of pre-processor implemented using Polyglot:



Desired Properties of Cloning

- When cloning an object, objects inside are also cloned
- When cloning an object, objects outside are not cloned
- The clone has the same shape as the original object

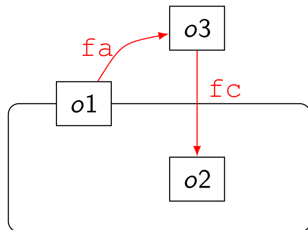


- Minimise dynamic information about ownership stored in derived code

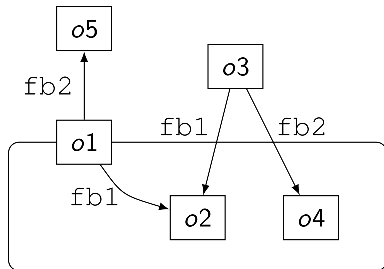
Cloning without Owners-as-Dominators

Problematic:

Re-entering domain paths (RDP)



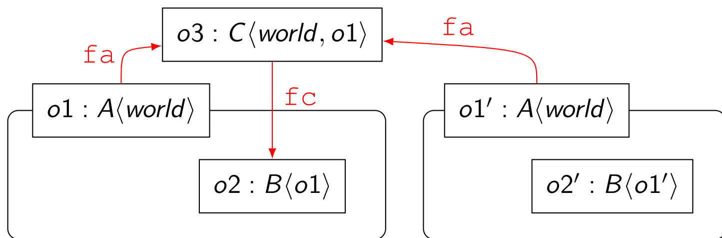
Non-problematic:



Cloning when Re-entering Domain Paths exist: Type Error

```
class A<c> {
    C<c,this> fa;
}

class B<c> {}

class C<c1,c2> {
    B<c2> fc;
}
```

$o1 : \texttt{A}\langle\texttt{world}\rangle$	(in diagram)
$o1' : \texttt{A}\langle\texttt{world}\rangle$	(in diagram)
$o1'.\texttt{fa} : \texttt{C}\langle \mathit{world}, o1'\rangle$	(by class declaration)
$o1'.\texttt{fa} = o3$	(in diagram)
$o3 : \texttt{C}\langle \mathit{world}, o1\rangle$	(in diagram)
$\texttt{C}\langle \mathit{world}, o1'\rangle \neq \texttt{C}\langle \mathit{world}, o1\rangle$	

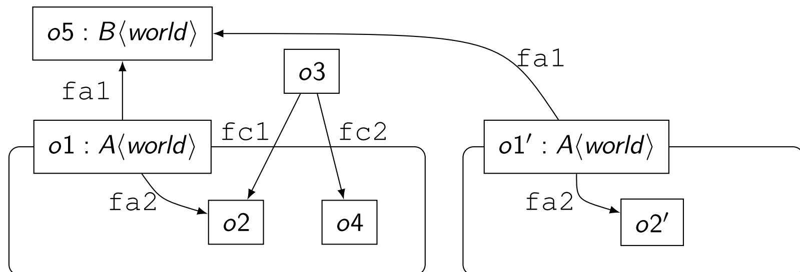


Cloning when Re-entering Domain Paths does not exist

Referenced objects outside of *o*1 are not given *o*1 as an owner parameter

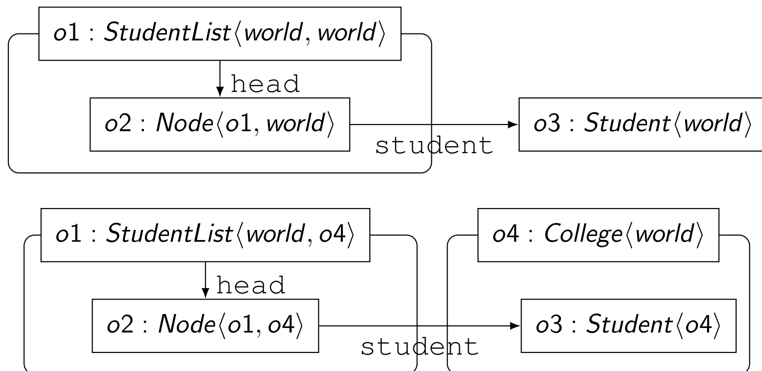
```
class A<c> {
    B<c> fa1;
    B<this> fa2;
}
```

```
class B<c> {}
class C<c> {
    B<c> fc1;
    B<c> fc2;
}
```



Preventing Re-entering Domain Paths

- We want to allow owners-as-dominators to be broken as long as there are no re-entering domain paths
- Re-entering domain paths occur dynamically

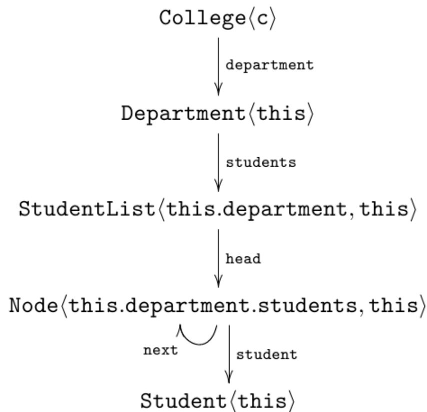


- Remove ownership information in derived code

Solution: Prevent Possibility of Re-entering Domain Paths

For each class declaration: check relative positions of objects along all field paths

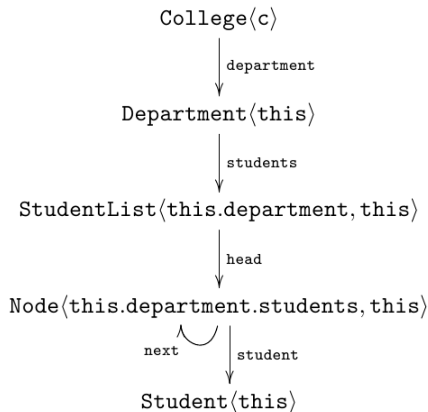
```
class College<c> {
  Department<this>
      department;
}

class Department<c> {
  StudentList<this,c>
      students;
}

class StudentList<c1,c2> {
  Node<this,c2> head;
}

class Node<c1,c2> {
  Student<c2> student;
  Node<c1,c2> next;
}

class Student<c> {}
```



Solution: Prevent Possibility of Re-entering Domain Paths

If owner of object is:

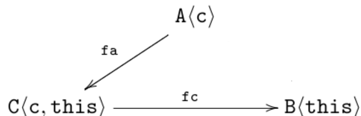
- A formal owner parameter:
  object is outside `this`
- `this`:
  object is inside `this`
- `world`:
  object is outside `this`
- A path, *p*:
  object is inside `this` *iff* object at *p* is inside `this`



Solution: Prevent Possibility of Re-entering Domain Paths

Example where RDP exists:

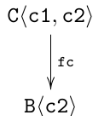
```
class A<c> {
    C<c,this> fa;
}

class B<c> {}

class C<c1,c2> {
    B<c2> fc;
}
```



(a) Graph for class A<c>, where there is a RDP.

B⟨c⟩

(b) Graph for class B<c>, where there is no RDP.

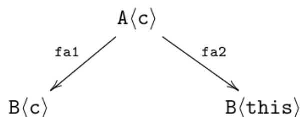


(c) Graph for class C<c1,c2>, where there is no RDP.

Solution: Prevent Possibility of Re-entering Domain Paths

Example where RDP does not exist:

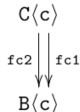
```
class A<c> {
    B<c> fa1;
    B<this> fa2;
}

class B<c> {}

class C<c> {
    B<c> fc1;
    B<c> fc2;
}
```



(a) Graph for class A<c>, where there is no RDP.

B⟨c⟩

(b) Graph for class B<c>, where there is no RDP.



(c) Graph for class C<c>, where there is no RDP.

Solution: Prevent Possibility of Re-entering Domain Paths Formalisation

$$Program, P = ClassId \to (\overline{c} \times (FieldId \to type) \times (MethId \to meth))$$

$$\begin{aligned} F(P, C, f) &= P(C) \downarrow_2 (f) \\ Fs(P, C) &= \{f | F(P, C, f) \text{ is defined}\} \\ O(P, C) &= P(C) \downarrow_1 \end{aligned}$$

$$\begin{aligned} EType &::= ClassId\langle\overline{por}\rangle \\ PathOrOwner, po &::= p | ca \end{aligned}$$

$$\begin{aligned} PG &: EType \to (FieldId \to EType) && \text{Path Graph} \\ OG &: EType \to (Path) && \text{Original Graph} \end{aligned}$$

Solution: Prevent Possibility of Re-entering Domain Paths Formalisation

$$inside: \; OriginGraph \times PathOrOwner \rightarrow Boolean$$

$$inside(OG, po) = \begin{cases} true & po = \texttt{this} \\ false & po = \texttt{world} \\ false & po = ca \\ inside(OG, po_1) & \text{otherwise} \\ \text{ where } OG(C\langle po_1, ..., po_n\rangle) = po & \end{cases}$$

Solution: Prevent Possibility of Re-entering Domain Paths Formalisation

The pair (PG, OG) is *complete* for a class C in program P *iff* the following conditions hold:

- $C\langle c_1, ..., c_n \rangle \in dom(PG)$ where $C \in dom(P)$ and $O(P, C) = c_1, ..., c_n$
- $Fs(P, C) = dom(PG(C\langle c_1, ..., c_n\rangle))$
- $OG(C\langle c_1, ..., c_n\rangle) = \texttt{this}$
- For any $D \in dom(P)$ and any field, f:

$$D\langle d_1, ...d_n\rangle \in \bigcup\{range(fToET) \mid fToET \in range(PG)\}$$

and $F(P, D, f) = t$

$$\Longrightarrow$$

$$PG(D\langle d_1, ..., d_n\rangle) = t' \text{ and } OG(t') = OG(D\langle d_1, ..., d_n\rangle).f$$

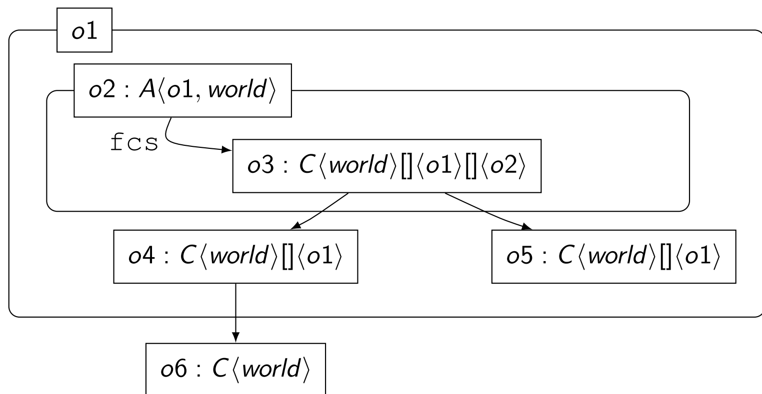
where $t' = t[d_1, ..., dn/O(P, D), OG(D\langle d_1, ..., d_n\rangle)/\texttt{this}]$

Solution: Prevent Possibility of Re-entering Domain Paths Formalisation

If (PG, OG) is *complete* for class C in program P, then a C object may have RDP's *iff*:

$$\exists D\langle d_1, ..., d_n\rangle, E\langle e_1, ...e_n\rangle, f: \begin{array}{l} PG(D\langle d_1, ..., d_n\rangle)(f) = E\langle e_1, ...e_n\rangle \\ \text{ and} \\ D\langle d_1, ...d_n\rangle \neq C\langle O(P, C)\rangle \\ \text{ and} \\ \neg inside(OG, d_1) \text{ and} inside(OG, e_1) \end{array}$$

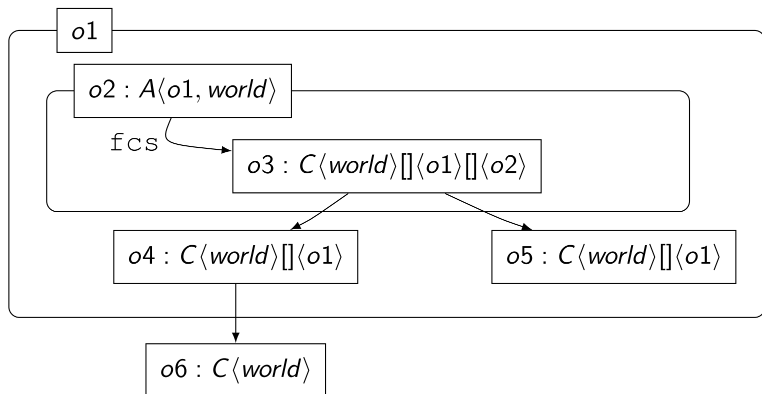
Arrays



Problems/considerations:

- Array elements may have different owner parameters than the array object
- Array class is predefined in Java
- Multi-dimensional arrays

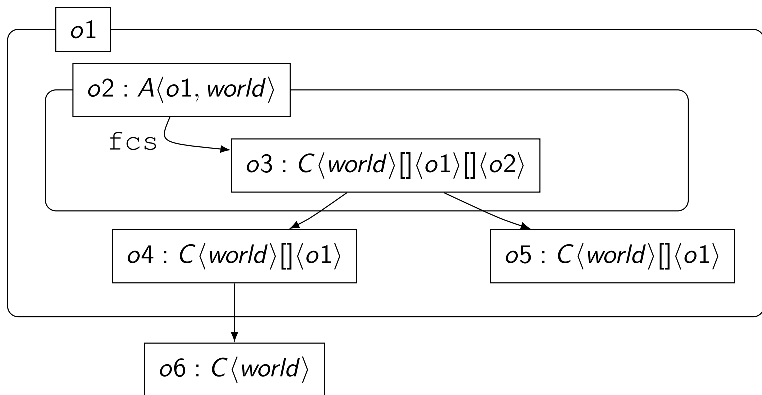
Arrays



For $o3 : C\langle world\rangle[]\langle o1\rangle[]\langle o2\rangle$

$$o2 = \text{owner of 2-dimensional array object}$$
$$o1 = \text{owner of 1-dimensional array objects}$$
$$C\langle world\rangle = \text{array leaf elements}$$

Arrays




```
class A<c1,c2> {
  C<c2>[]<c1>[]<this> fcs;
}

class B<c> {}

class C<c> {}
```

Generate Clone Method for Each Class & Dimension

Pass Boolean values to clone arrays as a list:

- Order according to depth of the array element
- For `C<c2>[]<c1>[]<this> fcs` Boolean values in order: [c1, c2]

```
class A/*<c1,c2>*/ {
  C[][] fcs; // C<c2>[]<c1>[]<this> fcs;

  A clone() {...}

  A clone(List<Boolean> bs, Map m) {
    ...

    newbs = new List<Boolean>();
    newbs.add(s1);
    newbs.add(s2);
    clone.fcs = true ? this.fcs.cloneC2(newbs, m) :
        this.fcs;
    ...
  }
}
```

Generate Clone Method for Each Class & Dimension

```
class C/*<c>*/ {
  C clone() {...}

  C clone(List<Boolean> bs, Map m) {...}

  static C[] cloneC1(C[] c, List<Boolean> bs, Map m)
      {
    ...
  }

  static C[][] cloneC2(C[][] c, List<Boolean> bs, Map m)
      {
    ...
  }
}
```

Generate Clone Method for Each Class & Dimension

1. Check whether array has already been cloned.
2. Otherwise create an array object as the clone.
3. 1st element of `bs` is not removed.

```
static C[]  cloneC1 ( C[] c ,
    List<Boolean> bs, Map m){
  Object n = m.get(c);
  if (n != null) {
    return (C[]) n;
  }

  C[] clone = new C[c.length];
  m.put(c, clone);

  Boolean owner = bs.get(0);
  for(int i = 0; i < c.length; i
      ++) {
    clone[i] = owner && c[i] !=
        null
      ? c[i].clone(bs, m)
      : c[i];
  }

  return clone;
}
```

Generate Clone Method for Each Class & Dimension

1. Check whether array has already been cloned.
2. Otherwise create an array object as the clone.
3. 1st element of `bs` *is removed*.

```
static C[][] cloneC2 ( C[][] c ,
    List<Boolean> bs, Map m){
  Object n = m.get(c);
  if (n != null) {
    return (C[][]) n;
  }

  C[][] clone = new C[c.length][];
  m.put(c, clone);

  Boolean owner = bs.remove(0);
  for(int i = 0; i < c.length; i
      ++) {
    clone[i] = owner && c[i] !=
        null
      ? cloneC1(c[i], bs, m)
      : c[i];
  }

  return clone;
}
```

Generics

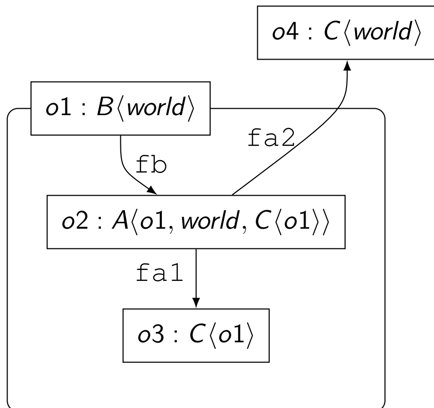
Problem:

- Actual owner parameters of generic type parameter is unknown statically

Solution:

- Store generic type parameters as indices into formal owner parameters (*permutation-like order list*)

```
class A<c1,c2, G > {
  G fa1;
  C<c2> fa2;
}

class B<c> {
  A<this,c,C<this>> fb;
}

class C<c> {}
```



Permutation-like Order List

Type	*Permuation-like order list stored in A objects*
$A\langle o1, world, C\langle o1\rangle\rangle$	[0]
$A\langle o1, o2, C\langle world\rangle\rangle$	[-2]

```
class Perm {
  static List<Boolean> reorder(List<Boolean> bs, List
      <Integer> perm) {
    List<Boolean> reordered = new List<Boolean>();

    for(Integer index : perm) {
      Boolean b = index == -1
        ? true
        : (index == -2 ? false : bs.get(index));
      reordered.add(b);
    }

    return reordered;
  }
}
```

Permutation-like Order List

```
class A<G> {
  G f1;
  C f2;

  List<Integer> gPerm;

  A(List<Integer> gPerm) {
    this.gPerm = gPerm;
  }

  A<G> clone() {
    List<Boolean> bs = new List<Boolean>();
    bs.add(false);
    bs.add(false);
    return this.clone(bs, new Map());
  }

...
```

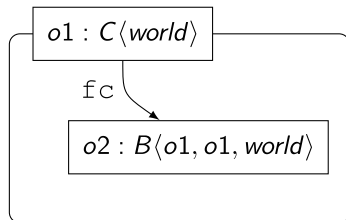
Permutation-like Order List

```
...
  A<G> clone(List<Boolean> bs, Map m) {
    Object n = m.get(this);
    if (n != null) {
      return (A<G>)n;
    }

    A<G> clone = new A<G>(this.gPerm);
    m.put(this, clone);

    clone.f1 = this.f1!=null &&
        bs.get(this.gPerm(0))
      ? this.f1.clone( Perm.reorder(bs,this.gPerm) ,m)
      : this.f1;

    List<Boolean> bsc = new List<Boolean>();
    bsc.add(bs.get(1));
    clone.f2 = this.f2 != null && bs.get(1)
      ? this.f2.clone(bsc, m) : this.f2;

    return clone;
  }
}
```

Other areas of the project not discussed in the presentation

- Extension to deal with subclassing

```
class A<c1,c2> {}

class B<c1,c2,c3>
    extends A<c1,c2> {
  A<world> fb;
}

class C<c> {
  A<c,c> fc;
}
```



- Combining the extensions for arrays, subclassing & generics

Applicability

Annotating the Java Class Library:

- Many classes implementing Cloneable use default clone method
- Ownership types may be too restrictive

```
private int width(String s) {
  ...
  width = Integer.parseInt(s);
  ...
}
```

- Few fields use `this` as owner parameter

```
class FilterOutputStream extends OutputStream {
  protected OutputStream out;

  public FilterOutputStream(OutputStream out) {
    this.out = out;
  }
```

- Other features: interfaces, static fields/classes, abstract classes

Using the Java Class Library:

- Classes do not have parametric clone methods

Conclusion

- Implemented basic cloning approach
- Explored the program & solution to cloning without owners-as-dominators
- Extended for arrays, subclassing and generics.
- Require further work to make cloning approach more applicable