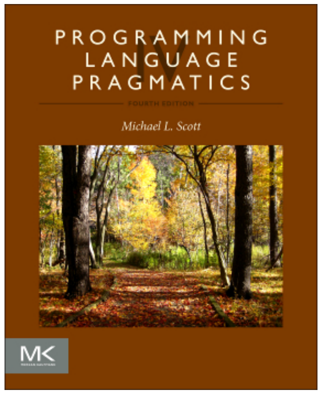
Implementation of Objects

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 10

Acknowledgments: some presentation ideas from Ben Titzer, Craig Chambers

Prof. Jonathan Aldrich

Copyright © 2016 Elsevier

HW 6 Mystery Explained

- When we studied composite types, we learned that records have two subtyping rules:

$$\frac{\overline{\tau} \leq \overline{\tau'}}{\{\, \overline{f : \tau} \,\} \leq \{\, \overline{f : \tau'} \,\}} \; \textit{S-depth}$$

$$\frac{}{\{\, \overline{f : \tau}, \overline{g : \tau'} \,\} \leq \{\, \overline{f : \tau} \,\}} \; \textit{S-width}$$

- But in the µTS specification, there is only the *S-width* rule. Why?

HW 6 Mystery Explained

- µTS has no *S-width* rule. Why?

$$\frac{\overline{\tau} \leq \overline{\tau'}}{\{\, \overline{f : \tau} \,\} \leq \{\, \overline{f : \tau'} \,\}} \; \textit{S-depth}$$

$$\frac{}{\{\, \overline{f : \tau}, \overline{g : \tau'} \,\} \leq \{\, \overline{f : \tau} \,\}} \; \textit{S-width}$$

- µTS interfaces are a combination of 3 things:
  - Records – *because there are several fields*
  - Recursive types – *because the interface type can be used in its own definition*
  - Pointer types – *because the fields are mutable*
    - Remember – $\tau_1{}^* \leq \tau_2{}^*$ only if $\tau_1 = \tau_2$
  - Fun fact: TypeScript interfaces are tagged unions too!
    - But not in the µTS language you are implementing

HW 6 Mystery Explained

- µTS has no *S-width* rule. Why?

$$\frac{\overline{\tau} \leq \overline{\tau'}}{\{\, \overline{f : \tau} \,\} \leq \{\, \overline{f : \tau'} \,\}}\ \textit{S-depth}$$

$$\frac{}{\{\, \overline{f : \tau}, \overline{g : \tau'} \,\} \leq \{\, \overline{f : \tau} \,\}}\ \textit{S-width}$$

- Our µTS rule is similar to the rules of Typescript
- Interestingly, Flow does support depth subtyping!
  - Flow is a different type system for JavaScript
  - Why? Flow lets you designate some fields as immutable
    - Can't write to those fields after initialization
    - Depth subtyping applies *only* to immutable fields
      - These fields are still implemented with pointers, but don't have to follow the invariant subtyping rules that pointers do

HW 6 Thoughts

- The main challenge in HW6 is probably just writing tree traversals in OCaml
- We assigned a checkpoint (due Thursday, October 27) to make sure you get started
- The checkpoint is a small portion of the overall work but we hope it will help you get over this “hump.”

ELSEVIER

Implementing Methods: `this`

- Methods are passed an extra, hidden, initial parameter: *this* (called *self* in Smalltalk and some other languages)
  - Allows the method to access the fields of the object, and call other methods

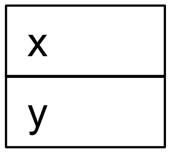
Object Models

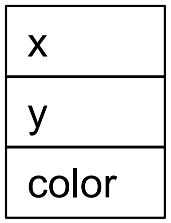
- An *object model* describes the data representation used by a language implementation
- Criteria for object models:
  - Implementation complexity
  - Performance
  - Space use
- Common features
  - An object is usually a contiguous block of memory
    - But sometimes several related blocks are used
  - Objects usually needs meta information – a “tag” or “header”
    - Typically more information is needed if the language is more dynamic, has reflection, or is garbage collected
- We’ll start with object models for statically typed single-inheritance OO languages like Java and C#

Prefixing - Implementing Inheritance

- Prefixing: layout of subclass has layout of superclass as a prefix

```
class Point {
   int x;
   int y;
}
class ColorPoint extends Point {
      Color color;
}
```





```
// OK, ColorPoint is a subtype of Point
Point p = new ColorPoint(0, 1, green);
// all subclasses of Point have x and y in the same place
int manhattanDistance = p.x + p.y;
```

Example due to Craig Chambers

Implementing Dynamic Dispatch

Possible Strategies

1. Each object knows its type; search the inheritance hierarchy
   - Very slow
2. Use a hash table
   - Can be a cache for strategy #1
   - Still slow, but was used in early Smalltalk systems
3. Store function pointers in objects, as if they were fields
   - This is conceptually how JavaScript works!
   - Invocation is fast & constant time: load and indirect jump
   - Con: objects are big!

   - Observation: in this strategy, all objects of the same class will store the same function pointers. Can we factor them out?

Virtual Function Tables

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```

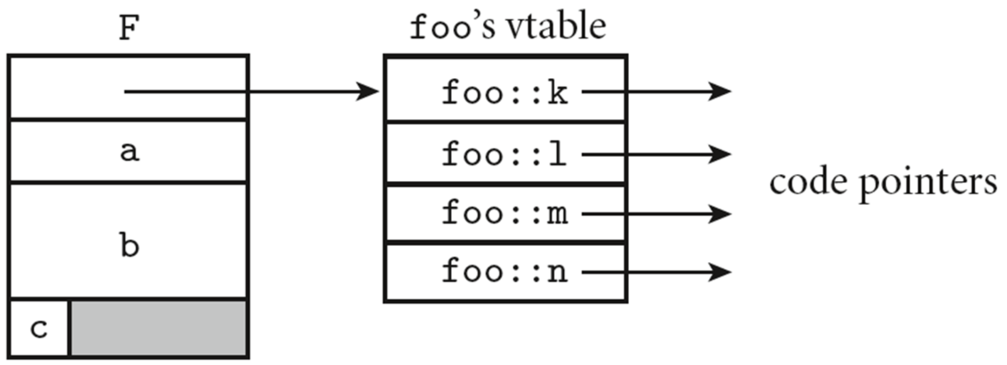


Figure 10.3 **Implementation of virtual methods.** The representation of object `F` begins with the address of the vtable for class `foo`. (All objects of this class will point to the same vtable.) The vtable itself consists of an array of addresses, one for the code of each virtual method of the class. The remainder of `F` consists of the representations of its fields.

- The assembly pseudocode generated for `f->m()` is:

```
r1 := f
r2 := *r1                  -- vtable address
r2 := *(r2 + (3−1) × 4) -- assuming 4=sizeof(address)
call *r2
```

Dynamic Method Binding

```
class bar : public foo {
    int w;
public:
    void m() override;
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```

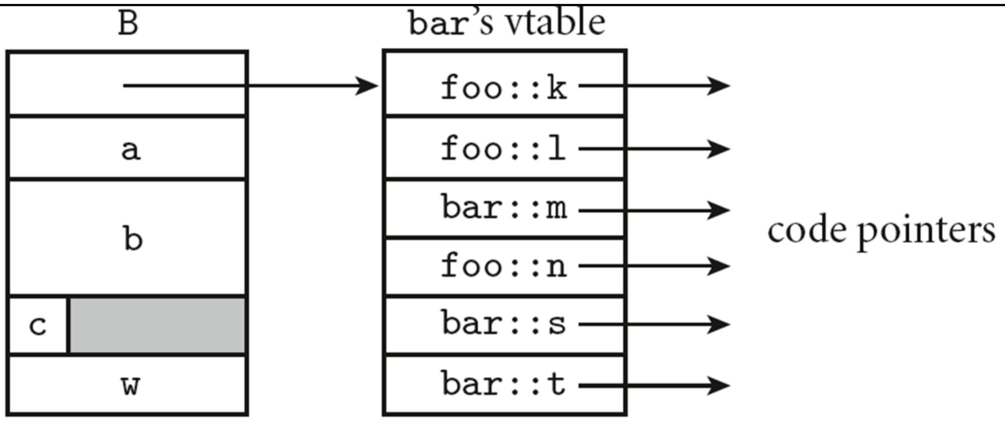
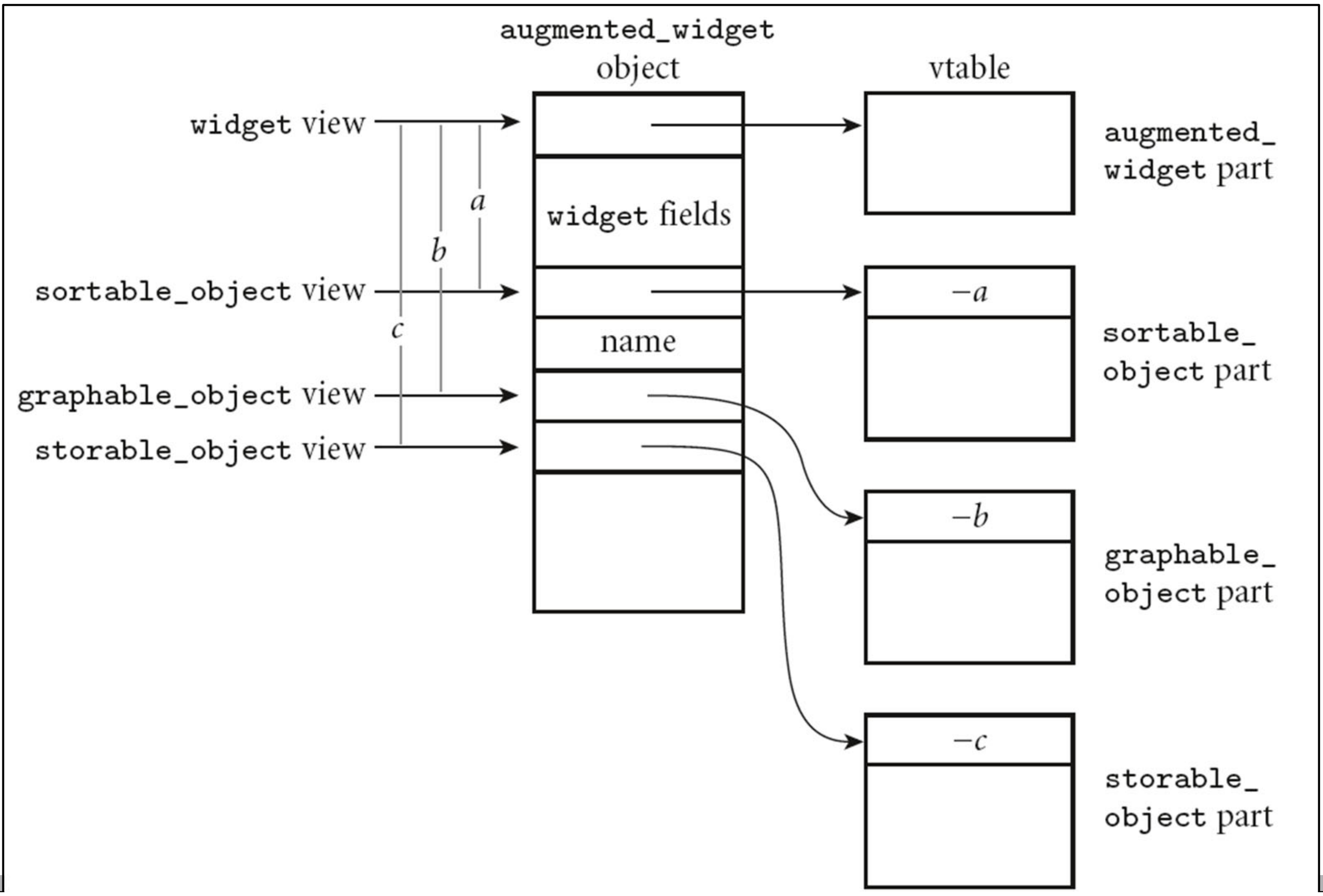


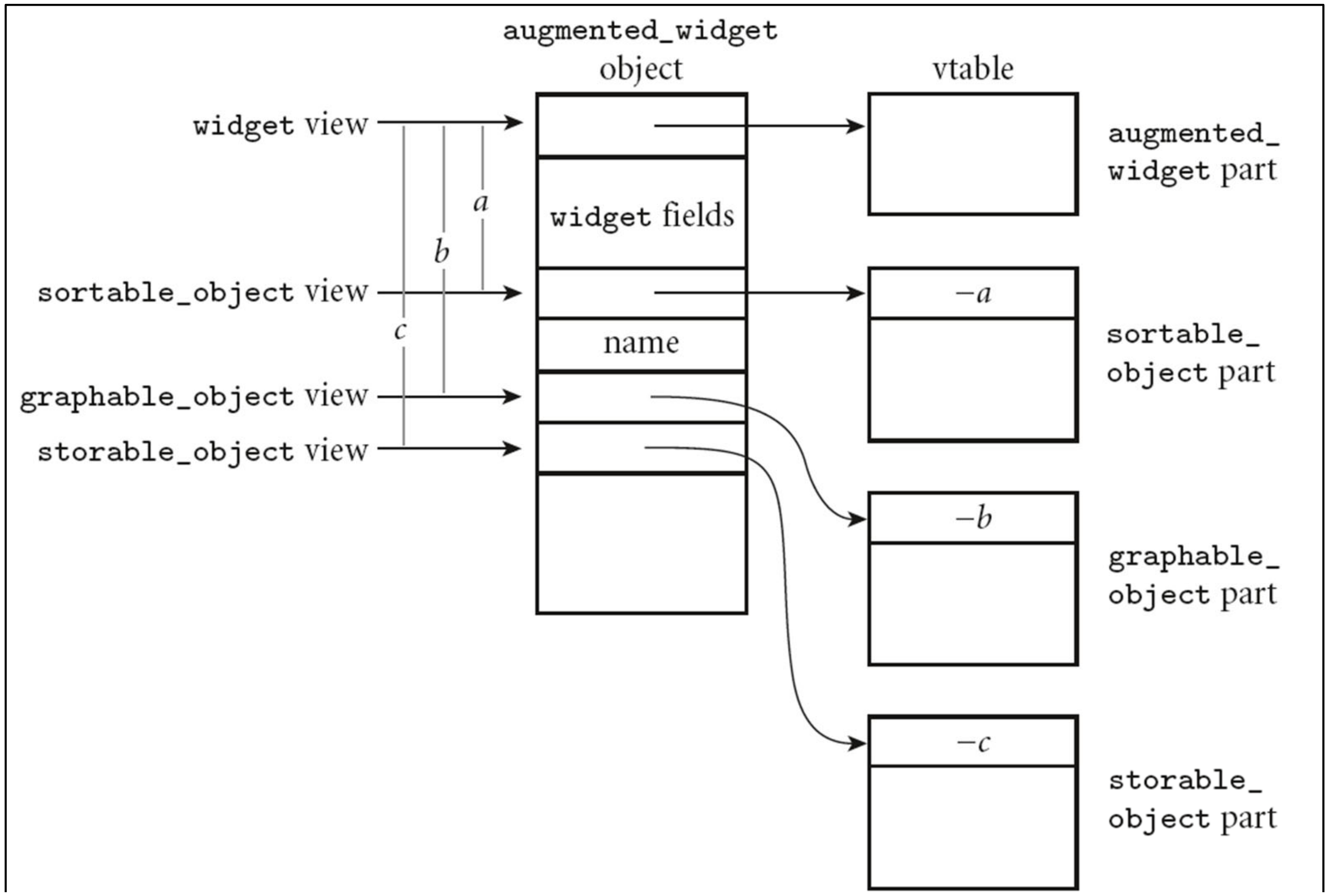
Figure 10.4 Implementation of single inheritance. As in Figure 10.3, the representation of object B begins with the address of its class's vtable. The first four entries in the table represent the same members as they do for foo, except that one—m—has been overridden and now contains the address of the code for a different subroutine. Additional fields of bar follow the ones inherited from foo in the representation of B; additional virtual methods follow the ones inherited from foo in the vtable of class bar.

Multiple Interface Inheritance

```
class widget { ... }
class named_widget  extends widget
      implements sortable_object { ... }
class augmented_widget extends named_widget
      implements graphable_object, storable_object { ... }
```



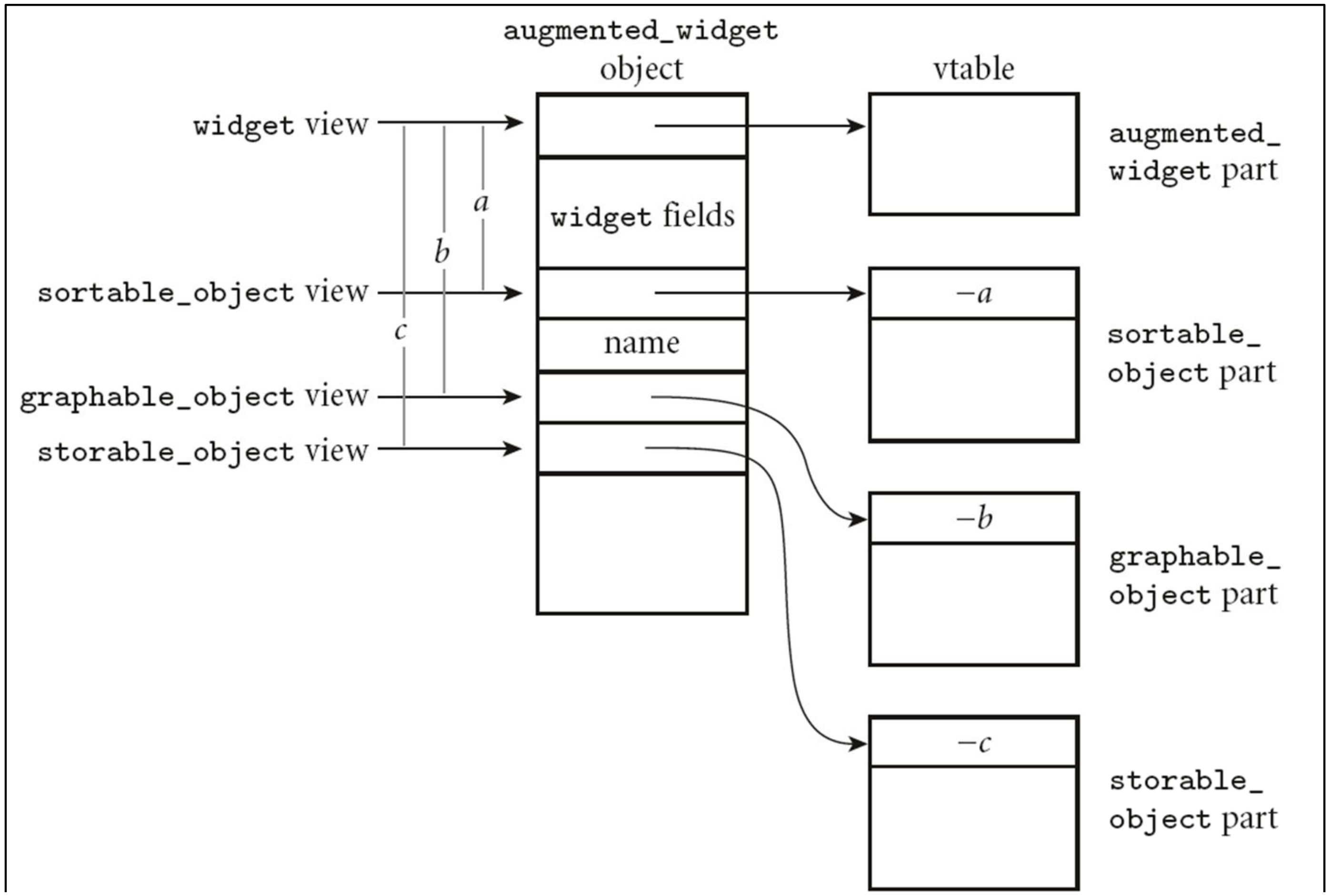
Multiple Interface Inheritance



- Consider a cast from `augmented_widget` to `sortable_object`:

```
r2 := r1 + a
```

Multiple Interface Inheritance



- Consider a call to an interface method of `sortable_object`

```
r2 := *r1              -- vtable address
r3 := *r2              -- this correction
r3 += r1               -- add correction to old address
call *(r2 + 4)         -- call (assumes first method in vtable)
```

Object model practice

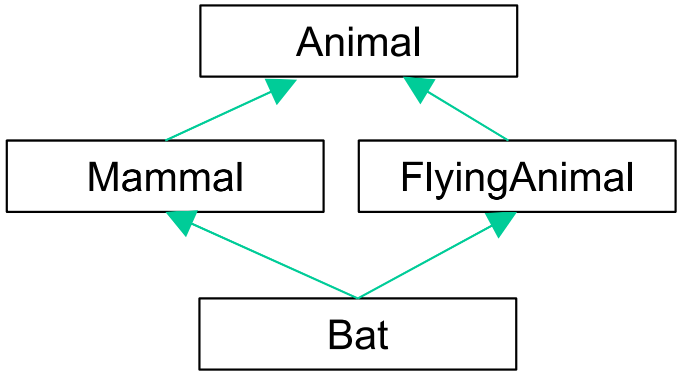
- Draw the layout of the object created at the end of this code. Show all virtual function tables.

```
interface Pingable {
  public void ping();
}
class Counter implements Pingable {
  int count = 0;
  public void ping() {
    ++count;
  }
  public int val() {
    return count;
  }
}

Counter c = new Counter();
```

ELSEVIER

Real Multiple Inheritance



Two approaches:

- "non-virtual inheritance" – A C++ hack
  - Just include state from both inherited classes
  - Works like multiple interface inheritance
  - If there's a diamond in the hierarchy, you get some fields twice
    - Good luck fixing bugs if the duplicate fields have inconsistent values!
  - Fast, simple, and works if there are no diamonds, or if the diamond classes have no state

- The right way (C++ virtual inheritance)
  - Essentially treat fields like methods – look up their location in a vtable
  - Slower, but has reasonable semantics

JavaScript’s Object Model

- Each object has multiple dynamically-typed properties
  - Indexed by strings
  - Can be added or deleted dynamically

- The vtable strategy doesn’t apply!
- Instead, start with a map from property name to value
  - Implemented as a list of pairs, or a hash table
  - Slow!

Optimizing JavaScript

- Start with a map from property name to value
  - Implemented as a list of pairs, or a hash table
  - Slow!

- Observation: most objects fall into one of a few “shapes”
- Used “hidden shapes”
  - A shared map that shows where to find an object’s properties
  - No need for a hash table for most objects

ELSEVIER

Inline Caches

- Consider looking up field x in the statement:

```
var f = o.x;
```

- An *inline cache* stores K entries, where an entry can be of the form:

  entry = {shape, offset}

- The access searches through the entries, looking for a matching shape
  - The hashtable is a backup

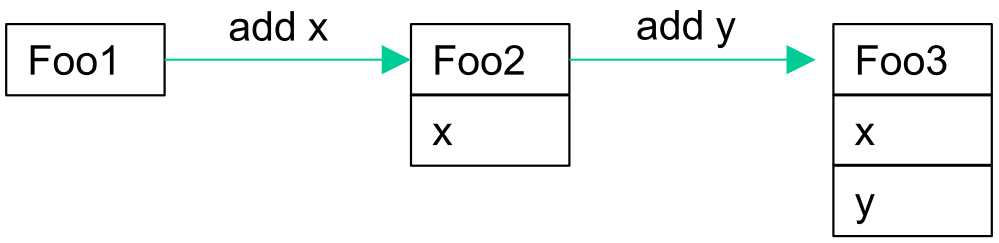
- Code for the inline cache access looks like:

```
  lookup(o: Object, ic: InlineCache, propertyName: string) {
    for (i = 0; i < K; i++) {
      if (o.shape == ic.entries[i].shape)
        return o.properties[ic.entries[i].offset];
    }
    // ic might be updated in this call
    return o.hashtable.lookup(propertyName, ic);
}
```

Hidden Classes

- Hidden classes form a tree with transitions
- Example:

```
function Foo(x, y) {
     this.x = x;
     this.y = y;
}
var x = new Foo(33, 44);
```



- Each time a field is added, the hidden class is updated
- Removing a field in LIFO order reverses the process
- Removing a different field? Typically go to hashtable strategy
  - otherwise we get too many hidden classes

ELSEVIER