A little bit on Class-Based OO Languages

CS4410: Spring 2013
Java Objects

Representing Objects

– 1st field is a pointer to a vtable
  • vtable: virtual method table.
  • each method is a procedure that takes an extra (implicit) argument corresponding to self.

– Remaining fields are instance variables.
**Simple Inheritance**

class Pt2d extends Object {
    int x;
    int y;
    void movex(int i) { x = x + i; }
    void movey(int i) { y = y + i; }
}

class Pt3d extends Pt2d {
    int z;
    void movez(int i) { z = z + i; }
}
class Pt2d {
    int x;
    int y;
    void movex(int i) { x = x + i; }
    void movey(int i) { y = y + i; }
}

class Pt3d {
    int x;
    int y;
    int z;
    void movex(int i) { x = x + i; }
    void movey(int i) { y = y + i; }
    void movez(int i) { z = z + i; }
}
At Run-Time:

Pt2d Object

- vtable pointer
- x
- y

Pt2d Vtable
- address of movex
- address of movey

Pt3d Object

- vtable pointer
- x
- y
- z

Pt3d Vtable
- address of movex
- address of movey
- address of movez
Jish Abstract Syntax

type tipe = Int_t | Bool_t | Class_t of class_name

type exp = Var of var | Int of int | Nil |
  Assign of var * exp | New of class_name |
  Invoke of exp * var * (exp list) | ...

type stmt = Exp of exp | Seq of stmt*stmt | ...

type method =
  Method of {mname:var, mret_tipe:tipe option, 
  margs:var*tipe list, mbody:stmt}

type class =
  Class of {cname:class_name, csuper:class_name, 
  cinstance_vars:var*tipe list, 
  cmethods:method list}
Compiling to Cish

- For every method $m(x_1,\ldots,x_n)$, generate a Cish function $m(self,vtables,x_1,\ldots,x_n)$.
- At startup, for every class $C$, create a record of $C$'s methods (the vtable.)
- Collect all of the vtables into a big record.
  - we will pass this data structure to each method as the vtables argument.
  - wouldn't need this if we had a global variable in Cish for storing the vtables.
- Create a Main object and invoke its main() method.
Operations:

• new C
  – create a record big enough to hold a C object
  – initialize the object's vtable pointer using vtables.
  – initialize instance variables with default values
    • 0 is default for int, false for bool, nil for classes.
  – return pointer to object as result

• e.m(e₁, ..., eₙ)
  – evaluate e to an object.
  – extract a pointer to the m method from e's vtable
  – invoke m, passing to it e, vtables, e₁, ..., eₙ
    • e is passed as self.
    • vtables is threaded through to every method.
  – in a real system, must check that e isn't nil!
Operations Continued:

• \(x, x := e\)
  – read or write a variable.
  – the variable could be a local or an instance variable.
  – if it's an instance variable, we must use the "self" pointer to access the value.
  – Real Java provides \(e.x\). Do we need this?

• \((C)e\) -- type casts
  – if \(e\) has type \(D\) and \(D \leq C\), succeeds.
  – if \(e\) has type \(D\) and \(C \leq D\), performs a run-time check to make sure the object is actually (at least) a \(C\).
  – if \(e\) has type \(D\), and \(C\) is unrelated to \(D\), then generates a compile-time error.
Subtleties in Type-Checking:

• Every object has a *run-time* type.
  – essentially, its vtable

• The type-checker tracks a *static* type.
  – some super-type of the object.
  – NB: Java confuses super-types and super-classes.

• In reality, if e is of type C, then e could be nil or a C object.
  – Java "C" = ML "C option"
Subtyping vs. Inheritance

• Inheritance is a way to assemble classes

• Simple inheritance:
  – D extends C implies D \leq C
  – a read of instance variable x defined in C?
    • okay because D has it too.
  – an invocation of method m defined in C?
    • okay because D has it too.
  – m : (C self,T_1,\ldots,T_n) \rightarrow T
    • What can m do to self?
    • Read C instance variables, invoke C methods.
Overriding:

```java
class List {
    int hd;  List tl;
    void append(List y) {
        if (tl == Nil) tl := y;
        else tl.append(y);
    }
}

class DList extends List {
    DList prev;
    void append(DList y) {
        if (tl == Nil) {
            tl := y;
            if (y != Nil) y.prev := self;
        } else {
            tl.append(y);
        }
    }
}
```

Java won't let you say this…
Best you can do:

class List {
    int hd;  List tl;
    void append(List y) {
        if (tl == Nil) tl := y;
        else tl.append(y);
    }
}

class DList extends List {
    DList prev;
    void append(List y) {
        if (tl == Nil) {
            tl := y;
            if (y != Nil) ((DList)y).prev := self;
        } else {
            tl.append(y);
        }
    }
}

Run-time type-check
What We Wish we Had…

• Don't just "copy" when inheriting:
  – Also replace super-class name with sub-class name.
  – That is, we need a "self" type as much as a self value.
  – But this will not, in general, give you that the sub-class is a sub-type of the super-class.
  – why?
Run-time Type Checks:

• Given an object x, how do we (quickly) determine if it has a run-time type D that is a sub-class of C?

• option 1: Have a link to the parent's vtable in the child's vtable.
  – crawl up the chain until you reach the parent (or Object).
  – disadvantage?

• other options?
Displays:

<table>
<thead>
<tr>
<th>address of method 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>address of method 2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>address of method n</td>
</tr>
<tr>
<td>num ancestors</td>
</tr>
<tr>
<td>parent @ level 0</td>
</tr>
<tr>
<td>parent @ level 1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>parent @ level m</td>
</tr>
</tbody>
</table>

Just have a pointer to all ancestors.

To check if C is a super-class:
  - statically calculate depth of C
  - check that num ancestors is at least that depth.
  - check that this ancestor is C.
Interfaces

• Consider an interface
• \( I = \{ \text{void foo(); void bar();} \} \)
• Any object of a class \( C \) that implements methods named \texttt{foo} and \texttt{bar} can be treated as if it has interface type \( I \).
• Can we use \( C \)'s vtable?
  – no.
  – In general, \( C \) may have defined methods before, between, or after \texttt{foo} and \texttt{bar} or may have defined them in a different order.
• So to support interfaces, we need a level of indirection...
Interfaces:

Wrapper Object

| vtable pointer | actual object |

Shared Vtable for Interface

| address of method 1 |
| address of method 2 |
| ... |
| address of method n |

Actual Object

| vtable pointer |
| instance variable 1 |
| instance variable 2 |
| ... |
| instance variable m |