CS412/413

Introduction to Compilers
Radu Rugina

Lecture 20: Implementing Objects
13 Mar 06

Code Generation for Objects

- Methods
  - Generating method code
  - Generating method calls (dispatching)
  - Constructors and destructors
- Fields
  - Memory layout
  - Generating code to access fields
  - Field alignment

Compiling Methods

- Methods look like functions, are type-checked like functions…what is different?
- Argument list: implicit receiver argument
- Calling sequence: use dispatch vector instead of jumping to absolute address

The Need for Dispatching

- Example:

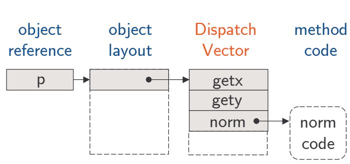
```
class Point { int x, y;
   float norm() { return sqrt(x*x+y*y); }
class 3DPoint extends Point { int z;
   float norm() { return sqrt(x*x+y*y+z*z); }

Point p;
if (cond) p = new Point();
else      p = new 3DPoint();
int n = p.norm();
```

- Compiler can't tell what code to run when method is called!

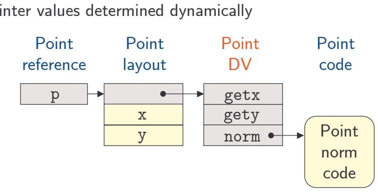
Dynamic Dispatch

- Solution: dispatch vector (dispatch table, selector table…)
  - Entries in the table are pointers to method code
  - Pointers are computed dynamically
  - If T ≤ S, then vector for objects of type S is a prefix of vector for objects of type T



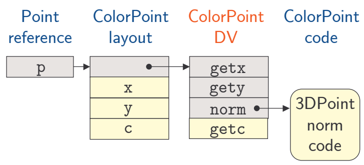
Why It Works

- If S ≤ T and f is a method of an object of type T, then
  - Objects of type S inherit f; f can be overridden by S
  - Pointer to f has same index in the DV for type T and S!
- Statically generate code to look up pointer to method f
- Pointer values determined dynamically



Why It Works

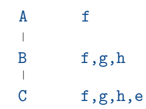
- If $S \leq T$ and f is a method of an object of type T, then
  - Objects of type S inherit f; f can be overridden by S
  - Pointer to f has same index in the DV for type T and S!
- Statically generate code to look up pointer to method f
- Pointer values determined dynamically



Dispatch Vector Lookup

- Every method has its own small integer index
- Index is used to look up method in dispatch vector

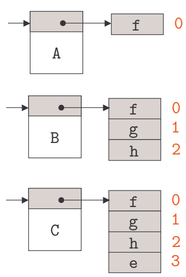
$C \leq B \leq A$




```
interface A {
     void f();       0
}
class B implements A {
     void f() {...}  0
     void g() {...}  1
     void h() {...}  2
}
class C extends B {
     void e() {...}  3
}
```


Dispatch Vector Layouts

- Index of f is the same in any object of type $T \leq A$
- Virtual methods may have multiple implementations
  - When subclass overrides method
- To execute a method i:
  - Lookup entry i in vector
  - Execute code pointed to by entry value

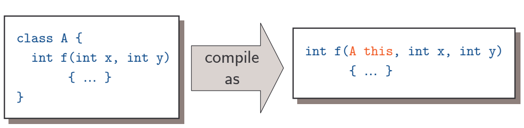


Interfaces, Abstract Classes

- Classes define a type and some values (methods)
- Interfaces are pure object types : no implementation
  - no dispatch vector: only a DV layout
- Abstract classes are halfway:
  - define some methods
  - leave others unimplemented
  - no objects (instances) of abstract class
- DV needed only for concrete classes

Method Arguments

- Methods have a special variable (Java, C++: this) called the receiver object
- Historically (Smalltalk): method calls thought of as messages sent to receivers
- Receiver object is (implicit) argument to method



Static Methods

- In Java or IC, one can declare methods static
  - they have no receiver object
- Called exactly like normal functions
  - don't need to enter into dispatch vector
  - don't need implicit extra argument for receiver
- Treated as methods as way of getting functions inside the class scope (access to module internals for semantic analysis)
- Not really methods

Code Generation: Dispatch Vectors

- Allocate one dispatch vector per class
  - Objects of same class execute same method code
- Statically allocate dispatch vectors

```
.data
PointDV:    .long _getx
            .long _gety
            .long _norm_P
```


Code Generation: Dispatch Vectors

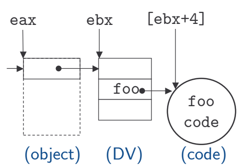
- Allocate one dispatch vector per class
  - Objects of same class execute same method code
- Statically allocate dispatch vectors

```
.data
3DPointDV:  .long _getx
            .long _gety
            .long _norm_3DP
            .long _getc
```


Example

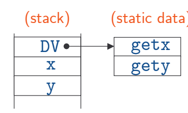
`o.foo(2,3);`

```
push $3
push $2
push %eax
mov (%eax), %ebx
call *4(%ebx)
add $12, %esp
```

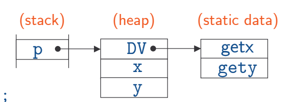


Allocation of Objects

- Objects can be stack- or heap-allocated
- Stack allocation:
  (C++) `Point p;`



- Heap:
  (C++)
  `Point *p = new Point;`
  (Java)
  `Point p = new Point();`



Code Generation: Allocation

- Heap allocation: o = new Point()
  - Allocate heap space for object
  - Store pointer to dispatch vector

```
push $12   # 2 fields+DV
call _GC_malloc
mov $PointDV, (%eax)
add $4, %esp
```

- Stack allocation:
  - Push object on stack
  - Pointer to DV on stack

```
sub $12, %esp    # 3 fields+DV
mov $PointDV, -4(%ebp)
```


Constructors

- Java, C++: classes can declare object constructors that create new objects:
  `new C(x, y, z)`
- Other languages (Modula-3): objects constructed by "new C"; no initialization code

```
class LenList {
    int len;  Cell head, tail;
    LenList() { len = 0; }
}
```

- Need to know when objects are constructed
  - Heap: new statement
  - Stack: at the beginning of their scope (blocks for locals, procedures for arguments, program for globals)

Compiling Constructors

- Compiled similarly with methods:
  - pseudo-variable "this" passed to constructor
  - return value is "this"

```
l = new LenList();
```

```
push $16   # 3 fields+DV
call _GC_malloc
mov $LenList_DV, (%eax)
add $4, %esp
push %eax
call LenList$constructor
add $4, %esp
```

```
LenList() { len = 0; }
```

```
LenList$constructor:
   push %ebp
   mov %esp,%ebp

   mov 8(%ebp), eax
   mov $0, 4(%eax)

   mov %ebp,%esp
   pop %ebp
   ret
```


Destructors

- In some languages (e.g. C++), objects can also declare code to execute when objects are destructed

- Heap: when invoking delete (explicit de-allocation)
- Stack: when scope of variables ends
  - End of blocks for local variables
  - End of program for global variables
  - End of procedure for function arguments

Field Offsets

- Offsets of fields from beginning of object known statically, same for all subclasses
- Example:

```
class Shape {
   Point LL /* 4 */ , UR; /* 8 */
   void setCorner(int which, Point p);
}
class ColoredRect extends Shape {
   Color c; /* 12 */
   void setColor(Color c_);
}
```

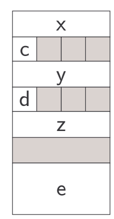
- Offsets known for stack and heap allocated objects

Field Alignment

- In many processors, a 32-bit load must be to an address divisible by 4, address of 64-bit load must be divisible by 8
- In rest (e.g. Pentium), loads are 10× faster if aligned -- avoids extra load

⇒ Fields should be aligned

```
class A {
   int x; char c;
   int y; char d;
   int z; double e;
}
```



Summary

- Method dispatch accomplished using dispatch vector, implicit method receiver argument
- No dispatch of static methods needed
- Inheritance causes extension of fields as well as methods; code can be shared
- Field alignment: declaration order matters!
- Each real class has a single dispatch vector in data segment: installed at object creation or constructor
- Analysis more difficult in the presence of objects
- Class hierarchy analysis = precisely determine object class