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:
  ```java
  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

<table>
<thead>
<tr>
<th>object reference</th>
<th>object layout</th>
<th>Dispatch Vector</th>
<th>method code</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
<td>getx</td>
<td>norm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gety</td>
<td>code</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Point reference</th>
<th>Point layout</th>
<th>Point DV</th>
<th>Point code</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>x</td>
<td>norm</td>
<td>code</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

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

```java
class A {
    int f(int x, int y) {
        ...)
    }
}
```

### 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

```plaintext
.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

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

Example

```plaintext
o.foo(2,3);

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

Code Generation: Allocation

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

```plaintext
push $12, %ebp
      # 3 fields=DV
call GC_malloc
mov %PointDV, (%eax)
add $4, %esp
```

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

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

Allocation of Objects

- Objects can be stack- or heap-allocated

```plaintext
(C++) Point p;

push x
getx
gety
```

```plaintext
(C++)
Point *p = new Point;

push x
getx
gety
```

```plaintext
(Java)
Point p = new Point();

push x
getx
gety
```

Constructors

- Java, C++: classes can declare object constructors that create new objects:
  ```plaintext
  new C(x, y, z)
  ```

- Other languages (Modula-3): objects constructed by “new C,” no initialization code
  ```plaintext
  class LenList {
    int len;  // Call 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”

```java
1 = new LenList();
LenList() { len = 0; }
```

| push $16 | e 3 fields=UV |
call _GC_malloc |
mov $LenList_UV, ($eax) |
add $4, $eax |
push $eax |
call LenListConstructor |
add $4, $eax |

• 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:

```java
class Shape {
  Point LL /= 4 */ , UL /= 8 */
  void setCorner(int which, Point p);
}
class Color 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 16x faster if aligned – avoids
  extra load

⇒ Fields should be aligned

```java
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