CS412/413

Introduction to Compilers
Radu Rugina

Lecture 21: Implementing Objects
17 Mar 04

Classes

- Components
  - fields/instance variables
    - values may differ from object to object
    - usually mutable
  - methods
    - values shared by all objects of a class
    - usually immutable
  - component visibility: public/private/protected

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:

```
interface Point {
    int getx(); int gety(); float norm(); }
class ColoredPoint implements Point {
    float norm() { return sqrt(x*x+y*y); }
class 3DPoint implements Point {
    float norm() { return sqrt(x*x+y*y+z*z); }
    point p;
    if (cond) p = new ColoredPoint();
    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

```
object reference object layout dispatch method
vector

p = [getx, gety, norm] code
```

1
**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

---

**Dispatch Vector Lookup**

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

\[
\begin{align*}
C & [: B <: A \\
A & f \\
B & f,g,h \\
C & f,g,h,e
\end{align*}
\]

interface A {
  void f();
}
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 \( T <: A \)
- Methods may have multiple implementations
  - For subclasses with unrelated types
  - If subclass overrides method
- To execute a method \( i \):
  - Lookup entry \( i \) in vector
  - Execute code pointed to by entry value

---

**Code Generation: Dispatch Vectors**

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

```java
.data
LenListDV: .long _LenList_first
         .long _LenList_rest
         .long _LenList_length
```

---

**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 receive 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, 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: Method Calls

- Code for function calls: pre-call + post-call code
  - Pre-function-call code:
    - Save registers
    - Push parameters
  - Pre-method call:
    - Save registers
    - Push parameters
    - Push receiver object reference
    - Lookup method in dispatch vector

Example

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

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

Object Layout

- Object consists of:
  - Methods
  - Fields
- Object layout consists of:
  - Pointer to DV, which contains pointers to methods
  - Fields

```
   (static data)    (code)
    UV
    x
    y
```

Allocation of Objects

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

```
   (stack)    (static data)
   DV
   x
   y
```

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

```
   (stack)    (heap)    (static data)
   p
   DV
   x
   y
```

Inheritance and Object Layout

- Method code copied down from superclass if not overridden by subclass
- Fields also inherited (needed by inherited code in general)
- Inheritance: add fields, methods
  - Extend layout
  - Extend dispatch vector
  - A supertype object can be used whenever a subtype object can be used
Inheritance and Object Layout

```java
class Shape {
    Point LL, UR;
    void setCorner(int which, Point p);
}
class ColoredRect extends Shape {
    int color;
    void setColor(int col);
}
```

Field Offsets

- Offsets of fields from beginning of object known statically, same for all subclasses
- Example:
  ```java
  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 10X faster if aligned — avoids extra load
  - Fields should be aligned
  ```
  struct {
      int x; char c; int y; char d;
      int z; double e;
  }
  ```

Accessing Fields

- Access fields of current object
  - Access x equivalent to this.x
  - Current method has "this" as argument
- Access fields of other objects
  - Access of the form o.x
- In both cases:
  - Use pointer to object
  - Add offset to the field
- Access o.x depends on the kind of allocation of o
  - Stack allocation: stack access (%ebp + stack offset)
  - Heap allocation: stack access + dereference

Code Generation: Allocation

- Heap allocation: o = new C()  
  - Allocate heap space for object
  - Store pointer to dispatch vector
  ```
  push $16  // 3 fields+DV  
call GC_malloc  
mov $LENLIST_DV, (%eax)  
add $4, %esp
  ```
- Stack allocation:
  - Push object on stack
  - Pointer to DV on stack
  ```
  sub $16, %esp  // 3 fields+DV  
mov $LENLIST_DV, -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"

```java
I = new LenList();
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 destroyed
  - 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

Analysis and Optimizations

- Dataflow analysis reasons about variables and values
  - Records (objects) consist of a collection of variables (fields) – analysis must separately keep track of individual fields
- Difficult analysis for heap-allocated objects
  - Object lifetime outlines procedure lifetime
  - Need to perform inter-procedural analysis
- Constructors/destructors: must take into account their effects

Class Hierarchy Analysis

- Method calls = dynamic, via dispatch vectors
  - Overhead of going through DV
  - Prohibits function inlining
  - Makes other inter-procedural analyses less precise
- Static analysis of dynamic method calls
  - Determine possible methods invoked at each call site
  - Need to determine principal types of objects at each program point (Class Hierarchy Analysis)
  - If analysis determines object o is always of type T (not subtype), then it precisely knows the code for o.foo()
- Optimizations: transform dynamic method calls into static calls, inline method calls

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