IR Code Generation (Part II)

Object-oriented language issues.

- *Classes and Objects*
  - *Storage allocation*
  - *Static class variables*
  - *Dynamic class variables*
- *Method Invocations*
  - *Static methods*
  - *Static-binding methods*
  - *Dynamic-binding methods*
  - *MINI's method invocation*
- *Others*
  - *Local variables and parameters*
  - *Non-local (class) variables*
  - *Activation record size*
  - *This pointer*
- *Advanced Topics*
  - *Multiple inheritance*
  - *Membership testing*

Storage Allocation for Class Objects

Observations:

- Static class variables are established per class; they should be allocated to a single static place
- Dynamic class variables are cloned every time a class object is created; they should be stored in the allocated space for the object.

General Strategies:

- A *class descriptor* for each class —
  - pointer(s) to parent class descriptor(s)
  - pointers to (local) methods
  - storage for static variables

  Alternatively, these items can be allocated individually with proper IDs attached to indicate their association to the class, e.g. using class name as a prefix.
- An *object record* for each class object —
  - pointer to class descriptor
  - storage for (local) class variables
  - storage for inherited variables

Object Record Layout

An object record contains space not only for variables belong to *this* class, but also for variables belong to ancestor classes.

How should the variables be laid out so that their offsets can be computed statically by the compiler? For *single-inheritance* languages, we have a solution.

The Prefixing Method:

> *When a class B extends a class A, those variables of B that are inherited from A are laid out in a B record at the beginning, in the same order they appear in an A record.*

Since a subclass always extends the set of variables defined in its base class, compiler can consistently assign each variable a *fixed (static) offset* in the object record; this offset will be the same in every object record for that class and any of its subclasses. Compiled methods can then reference variables by offset rather than by name.

An Example

```
class A           { int i=1, j=2; }
class B extends A { int m=3, n=4; }
class C extends A { int k=5; }
class D extends C { int l=6; }

class Test {
  ...
  A a = new A();
  B b = new B();
  C c = new C();
  D d = new D();
  ...
}

A's record:  B's record:  C's record:  D's record:
  i            i            i            i
  j            j            j            j
               m            k            k
               n                         l
```

Deciding Object's Size

To decide an object's size, the corresponding class's inheritance information (i.e. the total size of inherited variables) must be known.

Solution — When processing class declarations, collect class inheritance info and store it in class's symbol table entry.

Example:

```
class decls                           object sizes
--------------------------------------------------------
class A { int i=1, j=2; }             2*wdSize
class B extends A { int m=3, n=4; }   A's size + 2*wdSize
class C extends A { int k=5; }        A's size + 1*wdSize
class D extends C { int l=6; }        C's size + 1*wdSize
--------------------------------------------------------

class Test {
  A a = new A();   // allocate 2*wdSize
  B b = new B();   // allocate 4*wdSize
  C c = new C();   // allocate 3*wdSize
  D d = new D();   // allocate 4*wdSize
}
```

What if the declarations are not in the right order?

```
class D extends C { int l=6; }
class C extends A { int k=5; }
class B extends A { int m=3, n=4; }
class A { int i=1, j=2; }
```

Solution — Perform a topological sort on class decls based on inheritance relationship, then collect the size info.

Deciding Class Variables' Offsets

Once objects' sizes are known, class variables offsets can be computed easily:

> The offset of a subclass's first instance variable equals the parent class's object size.

Example:

```
class decls                 variable offsets
-------------------------------------------
class A {
  int i=1,                  0
      j=2;                  wdSize
}
class B extends A {
  int m=3,                  2*wdSize
      n=4;                  3*wdSize
}
class C extends A {
  int k=5;                  2*wdSize
}
class D extends C {
  int l=6;                  3*wdSize
}
-------------------------------------------
```

Class Variables' Initialization

According to Java's semantics, an instance variable will always be initialized, either by the user or by the compiler.

The difficulty is that the initialization happens when an new class object is created (could be anywhere in a program) while the user-provided initialization information is the class declaration section.

Solution — Collect class variable initialization info while processing class decls and store the info in class variable's symbol table entry. (If there no user-provided init value, a default value is stored.)

Example:

```
class decls                 offsets    init exps
-----------------------------------------------
class A {
  int i=1,                  0             1
      j=2;                  wdSize        2
}
class B extends A {
  int m=3,                  2*wdSize      3
      n=4;                  3*wdSize      4
}
class C extends A {
  int k=5;                  2*wdSize      5
}
class D extends C {
  int l=6;                  3*wdSize      6
}
-----------------------------------------------
```


New Objects

$\boxed{\texttt{E -> 'newObject' Id}}$

Issues:

- *Object size* — look up symbol table for info
- *Variable offsets* — calculate from object size
- *Variable initialization* — look up symbol table for info

Pseudo IR Code:

```
-------------------------------------------------------------
t1,t2,t3: new Temps;
E.s := [ <stmts embedded in var inits>; // maybe empty
         t1 := <#vars> * wdSize;     // info from symbol table
         t2 := malloc(t1);           // t2 points to obj record
         t3 := 0;                    // t3 points 1st var
         t2[t3] := <init-val1>;      // info from symbol table
         t3 := t3 + wdSize;          // t3 points 2nd var
         t2[t3] := <init-val2>;      // info from symbol table
         ... ]
E.t := t2;
-------------------------------------------------------------
```

An Example

```
class test {
  public static void main(String[] a) {
    A a = new A();
    B b = new B();
  }
}
class B extends A {
  int m = 3;
  int n = 4;
  int k;     // no init value
}
class A {
  int i = 1; int j = 2;
}
```

```
IR_PROGRAM
main (locals=2, max_args=1) {
 [MOVE (TEMP 100) (CALL (NAME malloc) ( (BINOP * (CONST 2) (NAME wSZ))))]
 [MOVE (MEM (TEMP 100)) (CONST 1)]
 [MOVE (MEM (BINOP + (TEMP 100) (NAME wSZ))) (CONST 2)]
 [MOVE (VAR 1) (TEMP 100)]
 [MOVE (TEMP 101) (CALL (NAME malloc) ( (BINOP * (CONST 5) (NAME wSZ))))]
 [MOVE (MEM (TEMP 101)) (CONST 1)]
 [MOVE (MEM (BINOP + (TEMP 101) (NAME wSZ))) (CONST 2)]
 [MOVE (MEM (BINOP + (TEMP 101) (BINOP * (CONST 2) (NAME wSZ)))) (CONST 3)
 [MOVE (MEM (BINOP + (TEMP 101) (BINOP * (CONST 3) (NAME wSZ)))) (CONST 4)
 [MOVE (MEM (BINOP + (TEMP 101) (BINOP * (CONST 4) (NAME wSZ)))) (CONST 0)
 [MOVE (VAR 2) (TEMP 101)]
}
```


Accessing Object's Instance Variables

`E -> E1 '.' Id`

Obtain object's address from `E1` and variable offset from `Id`.

```
class t4 {
  public static void main(String[] a) {
    A a = new A();
    B b = new B();
    System.out.println(a.i+a.j+b.x+b.y);
  }
}
class B { int x = 1; int y = 2; }
class A { int i = 3; int j = 4; }
```

```
IR_PROGRAM
main (locals=2, max_args=0) {
 [MOVE (TEMP 100) (CALL (NAME malloc)
                        ( (BINOP * (CONST 2) (NAME wSZ))))]
 [MOVE (MEM (TEMP 100)) (CONST 3)]
 [MOVE (MEM (BINOP + (TEMP 100) (NAME wSZ))) (CONST 4)]
 [MOVE (VAR 1) (TEMP 100)]
 [MOVE (TEMP 101) (CALL (NAME malloc)
                        ( (BINOP * (CONST 2) (NAME wSZ))))]
 [MOVE (MEM (TEMP 101)) (CONST 1)]
 [MOVE (MEM (BINOP + (TEMP 101) (NAME wSZ))) (CONST 2)]
 [MOVE (VAR 2) (TEMP 101)]
 [CALLST (NAME prInt) ( (BINOP + (BINOP + (BINOP +
                         (MEMBER (VAR 1) 0)    // 1st var of a (i)
                         (MEMBER (VAR 1) 1))   // 2nd var of a (j)
                         (MEMBER (VAR 2) 0))   // 1st var of b (x)
                         (MEMBER (VAR 2) 1)))] // 2nd var of b (y)
}
```

Static Methods

Static methods can be invoked *only* through class names. However, inheritance needs to be taken into consideration.

Example:

```
class c1 {
  static void print(int i) {
    System.out.println(i);
  }
  public static void main(String[] a) {
    c2.print(123);
  }
}
class c2 extends c1 { ... }
```

Given $C.f()$, the compiler can:

1. Search C's definition to see if the method f is defined there, if not, search the parent's class definition. Repeat this step until the method is found.
2. Generate code for accessing the method.

Static-Binding Methods

Method binding approach is a choice of language design — some languages use static binding (e.g. C++), some use dynamic binding (e.g. Java).

Static-binding methods are invoked through variables that represent objects.

Given $v.f()$, the compiler can:

1. From v's declaration, figure out the v's class type (say C).
2. Search C's definition to see if the method f is defined there, if not, search the parent's class definition. Repeat this step until the method is found.
3. Generate code for accessing the method.

Dynamic-Binding Methods

With dynamic-binding, given $v.f()$, the method to be invoked may not be the one defined in the declared class of variable v. E.g.

```
class A { public void foo() {...} ... }
class B extends A { public void foo() {...} ... }
A a = new B();
a.foo();
```

The declared type of a is `A`, yet the version of `foo` to be invoked by `a.foo()` should be the one defined in `B`.

So the previous approach won't work.

Simple Approach:

- Keep a static *method table* in each class descriptor —
  - the table contains both local *and* inherited methods
  - for overriding methods, only the overriding version is kept in the table
- Compile method invocations into indirect jumps through fixed offsets in this table:
  1. Starting from the object's record, follow pointer to class descriptor;
  2. Look up the method table for the method's address;
  3. Jump to the address.

Dynamic Methods (cont.)

Problems with the simple approach:

- Much of the method table contents will be duplicated between a class and its base classes.
- The method tables can get very large, yet many entries may never get used.

Alternative Approach:

- Use naive lookup scheme — starting from the object's local class descriptor, if the method can't be found there, goes up to the parent's, and so on.
- But keep a dynamic *method cache* recording the (target class, code pointer) pairs that have been discovered by recent lookups. Since the same methods tend to be called repeatedly on the same object classes, this can speed things up a lot.

Both of these "compilation" schemes are further hindered because OO environments are often very dynamic — new versions of classes can be reloaded at any time — so frequent recompilation and cache flushing may be needed.

MINI's Method Invocation

`E -> E1 '.' Id ExpList`

1. Extract the class name `C` out from the class object `E1`.
2. Concatenate `C` and method name (`Id`) to form a unique label.
3. Construct a `CALL` node with the label and an argument list.

```
class test {
  public static void main(String[] a) {
    A a = new A();
    int x = a.foo(1,2);
    a.bar(3,4);
  }
}
class A {
  public int foo(int i, int j) { return i+j; }
  public void bar(int i, int j) { System.out.println(i+j); }
}

IR_PROGRAM
main (locals=2, max_args=3) {
 [MOVE (TEMP 100) (CALL (NAME malloc) ( (NAME wSZ)))]
 [MOVE (VAR 1) (TEMP 100)]
 [MOVE (TEMP 101) (CALL (NAME A_foo)
                        ( (VAR 1) (CONST 1) (CONST 2)))]
 [MOVE (VAR 2) (TEMP 101)]
 [CALLST (NAME A_bar) ( (VAR 1) (CONST 3) (CONST 4))]
}
A_foo (locals=0, max_args=0) {
 [RETURN (BINOP + (PARAM 1) (PARAM 2))]
}
A_bar (locals=0, max_args=0) {
 [CALLST (NAME prInt) ( (BINOP + (PARAM 1) (PARAM 2)))]
}
```


Local Variables and Parameters

Both local variables and parameters are eventually stored in a method's *activation record* — space allocated on stack at runtime, and are accessed through offsets from a stack pointer.

At IR level, they are stored in two separate abstract arrays, `VAR` and `PARAM`, and are indexed by their positions in their corresponding declarations.

```
class t1 {
  public static void main(String[] a) {
    A a = new A(); a.sum(3,4); }
}
class A {
  public void sum(int m, int n) {
    int i = 1; int j = 2;
    System.out.println(i+j+m+n); }
}

IR_PROGRAM
main (locals=1, max_args=3) {
 [MOVE (TEMP 100) (CALL (NAME malloc) ( (NAME wSZ)))]
 [MOVE (VAR 1) (TEMP 100)]
 [CALLST (NAME A_sum) ( (VAR 1) (CONST 3) (CONST 4))]
}
A_sum (locals=2, max_args=0) {
 [MOVE (VAR 1) (CONST 1)]
 [MOVE (VAR 2) (CONST 2)]
 [CALLST (NAME prInt) ( (BINOP + (BINOP + (BINOP +
                          (VAR 1)       // 1st var (i)
                          (VAR 2))      // 2nd var (j)
                          (PARAM 1))    // 1st param (m)
                          (PARAM 2)))]  // 2nd param (n)
}
```

Accessing Class Variables

Class variables defined in a method's enclosing scopes can be accessed within the method's body.

But how does a method know about its class object's address?

```
class B {
  int x = 1; int y = 2;
}
class A extends B {
  int i = 3; int j = 4;
  public void sum() { System.out.println(i+j+x+y); }
}
```

Solution —

- Always pass the current object's address as the *zero*-th parameter (`(PARAM 0)`) to a method invocation; this special parameter is called *access link*.
- Class variables can then be accessed through offsets from the access link. We use an abstract reference form
  `(MEMBER <access link> <var's idx>)`
  to represent them.

An Example

```
class t3 {
  public static void main(String[] a)
   { A a = new A(); a.sum(); }
}
class B {
  int x = 1; int y = 2;
}
class A extends B {
  int i = 3; int j = 4;
  public void sum() { System.out.println(i+j+x+y); }
}

IR_PROGRAM
main (locals=1, max_args=1) {
 [MOVE (TEMP 100) (CALL (NAME malloc)
                        ( (BINOP * (CONST 4) (NAME wSZ))))]
 [MOVE (MEM (TEMP 100)) (CONST 1)]
 [MOVE (MEM (BINOP + (TEMP 100) (NAME wSZ))) (CONST 2)]
 [MOVE (MEM (BINOP + (TEMP 100) (BINOP * (CONST 2) (NAME wSZ))))
       (CONST 3)]
 [MOVE (MEM (BINOP + (TEMP 100) (BINOP * (CONST 3) (NAME wSZ))))
       (CONST 4)]
 [MOVE (VAR 1) (TEMP 100)]
 [CALLST (NAME A_sum) ( (VAR 1))]
}
A_sum (locals=1, max_args=0) {
 [CALLST (NAME prInt) ( (BINOP + (BINOP + (BINOP +
   (MEMBER (MEM (PARAM 0)) 2)
   (MEMBER (MEM (PARAM 0)) 3))
   (MEMBER (MEM (PARAM 0)) 0))
   (MEMBER (MEM (PARAM 0)) 1)))]
}
```

Space for Method Invocation

When invoking a method at runtime, the runtime system needs to know how much space to allocation for the method's activation record.
Among the factors, two can be computed by compiler

- *Space reserved for holding local variables* — can be obtained easily by counting the number of local variables
- *Space reserved for preparing arguments to method calls* — requires going through the method's body and check every `Call` node (excluding calls to system routines) — count the number of parameters, if the count succeeds the previous max, then update the max

```
class t1 {
  public static void main(String[] a) {
    A a = new A(); a.sum(3,4); }
}
class A {
  public void sum(int m, int n) {
    int i = 1; int j = 2;
    System.out.println(i+j+m+n); }
}

IR_PROGRAM
main (locals=1, max_args=3) {
 ...
}
A_sum (locals=2, max_args=0) {
 ...
}
```


Adv. Topic: Multiple Inheritance

When a class can extend several parent classes, the *prefixing* layout method for class variables doesn't work any more — it is not possible to put two parents' variables both at the beginning.

Some form of ordering among the parents' variables is needed. In fact, some form of *global* ordering among *all* classes' variables is needed.

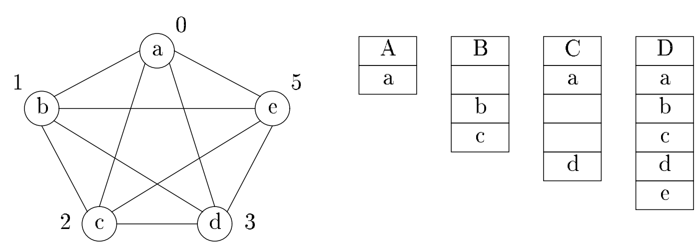
The Global Graph Coloring Approach:

The problem of ordering all classes' variables can be modeled as a graph-coloring problem:

- There is a node for each distinct variable name.
- There is an edge for any two variables which coexist (perhaps by inheritance) in the same class.

Example:

```
class A { int a = 0; }
class B { int b = 0; int c = 0; }
class C extends A { int d = 0; }
class D extends B,C { int e = 0; }
```

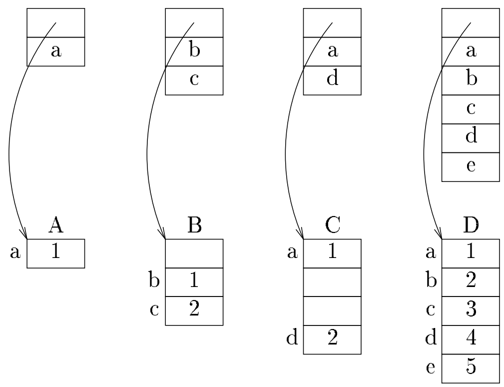


Multiple Inheritance (cont.)

Problem: Empty slot in the middle of objects.

Solution:

- Pack variables in objects records.
- Store graph coloring result and packing info in class descriptors.



Accessing Variables:

1. Fetch the descriptor pointer from the object.
2. Fetch the variable-offset value from the descriptor.
3. Access the data the appropriate offset in the object.

Methods can be looked up using the same technique.

Adv. Topic: Class Membership

Some OO languages have a construct for testing class membership of an object at runtime:

```
x instanceof C
```

Also, for strongly-typed OO languages require runtime membership checks when there is casting involved:

```
Vector fruits = new Vector();
fruits.add(new Apple());
fruits.add(new Orange());
...
Apple a2 = (Apple) fruits.elementAt(i);
```

How to implement membership testing?

Naive Approach:

1. Fetch `x`'s class descriptor; if the descriptor matches `C`, then return `true`.
2. Else fetch the parent's class descriptor and check. Repeat this step until one matches `C` (return `true`) or no more parent exists (return `false`).

Class Membership (cont.)

More Efficient Approach:

- Reserve K slots in each class descriptor for a *display*, where K is max class nesting depth.
- For a class C at depth d, store pointers to class descriptors of C and C's ancestors in slots d through 0. (This step is performed by the compiler.)
  Fact — After this step, in the class descriptor of any C's subclass, the dth display slot will contain a pointer to C's descriptor.
- To test `x instanceof C`, just check the jth display slot in `x`'s class descriptor to see if it matches with `C` (where j is the depth of `C`).

Example:

```
class A           { int a = 0; }
class B extends A { int b = 0; int c = 0; }
class C extends A { int d = 0; }
class D extends C { int e = 0; }
```

```
A:                    C:
  ptr to Object         ptr to Object
  ptr to A              ptr to A
                        ptr to C
B:
  ptr to Object       D:
  ptr to A              ptr to Object
  ptr to B              ptr to A
                        ptr to C
                        ptr to D
```