Week 5: IR Code Generation II

Jingke Li, Portland State University

Winter 2015

Today's IR CodeGen Topics

- Classes
  - Storage allocation
  - Field variable declarations
- Objects and Instance Variables
  - Storage layout and offset computation
  - New objects
  - Instance variable access
- Methods
  - Static methods
  - Static-binding vs. dynamic-binding

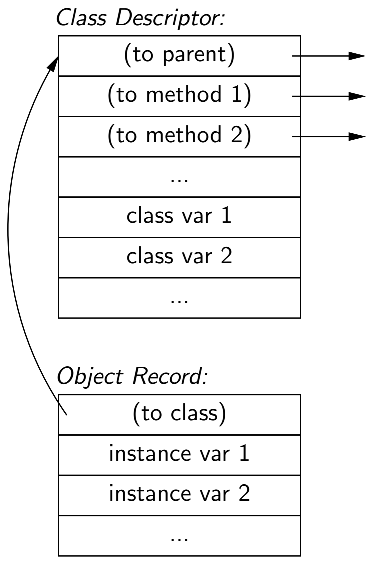
Storage Allocation for Classes and Objects

Both class and object need storage support, since they both have data associated with them. The general allocation strategy is:

- A *class descriptor* for each class, to store
  - Pointer to parent class descriptor
  - Pointers to (local) methods
  - Class variables (static fields)

  Class descriptors are allocated in the static area.
- An *object record* for each object, to store
  - Pointer to class descriptor
  - Instance variables (both local and inherited)

  Object records are allocated on the heap.



Class vs. Instance Variables

```
class P {
  static int i;
  int j;
}
class Q extends P {
  int k;
}
class Demo {
  public static void main(String[] x) {
    P x = new P();
    Q y = new Q();
    int val = P.i + x.j + y.j + y.k;
  }
}
```

Observations:

- Class variables are established per class and accessed through class references. Hence they are naturally allocated in class descriptors.
- Instance variables are cloned every time a class object is created. They therefore are stored in the allocated space for the object.

Storage Issues with Instance Variables

In IR, instance variables are typically represented as offsets from the address of the object record, rather than by their names.

Two issues need to be resolved:

- *Layout* — In what order should the variables be laid out in an object record?
- *Offset* — How to compute the offset for each variable?

Instance Variables' Layout

For a class object, instance variables can come from the current class, or any of the ancestor classes. The challenge to an IR generator is to assign each instance variable a *fixed* offset location, regardless of which object it is stored in (*e.g.* whether it's a base class object or a sub-class object).

Example:

```
class P           { int i; int j;}
class Q extends P { int k; }
class R extends P { int l; int m; }
...
P x = new P();
Q y = new Q();
R z = new R();
```

Although `x`, `y`, and `z` are three different objects, we want `x.i`, `y.i`, and `z.i` mapped to the same offset.

Why? To support sub-type polymorphism. *E.g.* If we assign `y` to `x`, we still want to access `x.i` the same way.

The Prefixing Layout Method

For *single-inheritance* languages such as Java, we have a solution for the instance variable layout problem:

- For a base class, its instance variables are laid out in the order of their declarations.
- When a class B extends a class A, the inherited variables from A are laid out at the beginning in a B object, in the same order they appear in an A object.
- Class B's own instance variables then follow in the order of their declarations.

```
class P           { int i; int j;}
class Q extends P { int k; }
class R extends P { int l; int m; }
...
P x = new P();
Q y = new Q();
R z = new R();
```

```
Variable Layout

x      y      z
---------------
i      i      i
j      j      j
       k      l
              m
```

An IR Example

```
class P           { int i; int j;}
class Q extends P { int k; }
class R extends P { int l; int m; }

class Demo {
  public static void main(String[] a) {
    P x = new P();
    Q y = new Q();
    R z = new R();
    System.out.println(x.i);
    System.out.println(x.j);
    System.out.println(y.i);
    System.out.println(y.j);
    System.out.println(y.k);
    System.out.println(z.i);
    System.out.println(z.j);
    System.out.println(z.l);
    System.out.println(z.m);
  }
}
```

```
x = call _malloc(16)
y = call _malloc(20)
z = call _malloc(24)
...
t4 = 8[x]:I         # x.i
call _printInt(t4)
t5 = 12[x]:I        # x.j
call _printInt(t5)
t6 = 8[y]:I         # y.i
call _printInt(t6)
t7 = 12[y]:I        # y.j
call _printInt(t7)
t8 = 16[y]:I        # y.k
call _printInt(t8)
t9 = 8[z]:I         # z.i
call _printInt(t9)
t10 = 12[z]:I       # z.j
call _printInt(t10)
t11 = 16[z]:I       # z.l
call _printInt(t11)
t12 = 20[z]:I       # z.m
call _printInt(t12)
```

Instance Variables' Offsets

- If we are dealing with a base class, then it's easy — the first variable directly follows the pointer to class descriptor, so it has an offset of 8, the second variable follows the first, and so on.

```
x             offset
--------------------
ptr_to_class       0
i                  8
j                 12
```

- If we are dealing with a sub-class, then the first variable needs to follow the inherited variables:

```
y             offset
--------------------
ptr_to_class       0
(inherited vars)
k                  ?
```

Observation: The parent's object record contains all the content up to the first variable k, so k's offset equals the parent's object record size.

Objects' Sizes

To compute class object sizes for a program, perform a topological sort on class declarations of the program based on inheritance relationship; then compute object size for classes in the sorted order, starting from the base class.

Example:

```
class R extends P { int l; int m; }
class Q extends P { int k; }
class P           { int i; int j;}
```

After sorting:

```
Class Decl                            Object Size
----------------------------------------------------------------------
class P           { int i; int j;}    1*8 + 2*4 = 16 (1 ptr + 2 int)
class Q extends P { int k; }          16 + 1*4 = 20  (P's size + 1 int)
class R extends P { int l; int m; }   16 + 2*4 = 24  (P's size + 2 int)
----------------------------------------------------------------------
```

Instance Variables' Initialization

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

One common approach for dealing with this requirement is

- collect instance variable initializations from class declaration;
- turn them into store instructions into these variables' storage;
- place these instructions into a special "`init`" routine;
- generate code to invoke this routine every time a new object is created.

If a variable does not have a user-provided initial value, the IR generator will use a default value (*e.g.* `0`/`false`/`null`).

User-defined constructor routines become additional versions of the "`init`" routine.

Variable Initialization Example

```
class A {
  int i;
  int j = 2;
  boolean b;
  int[] a;
}

class Demo {
  public static void main(String[] x) {
    A a = new A();
    System.out.println(a.i);
    System.out.println(a.j);
    System.out.println(a.b);
    System.out.println(a.a);
  }
}
```

```
_init_A (obj:P)
{
 8[obj]:I = 0        # i
 12[obj]:I = 2       # j
 16[obj]:B = false   # b
 17[obj]:P = null    # a
}

_main ()
(a:P)
{
 t1 = call _malloc(25)
 [t1]:P = _class_A
 call _init_A(t1)
 a = t1
 t2 = 8[a]:I
 call _printInt(t2)
 t3 = 12[a]:I
 call _printInt(t3)
 ...
}
```

New Objects

`Exp → (NewObj <StrLit>)`

CodeGen Step:

- Use class name to get object's size from CodeGen's internal info store.

IR Code Template:

- Call `malloc` to allocate space
- Store class descriptor pointer to object
- Call the `init` routine to initialize object fields
- Return the object's address as `Exp`'s value

Accessing Instance Variables

`Exp → (Field Exp₁ <StrLit>)`

CodeGen Step:

- Obtain offset value for the field from CodeGen's internal info store

IR Code Template:

- Obtain object's address by recursively processing the object component ($\texttt{Exp}_1$)
- Load field value from the object's storage to a temp
- Return the temp as `Exp`'s value

An IR Example

```
class P            { int i; int j;}
class Q extends P { int k; }
class R extends P { int l; int m; }

class Demo {
  public static void main(String[] a) {
    P x = new P();
    Q y = new Q();
    R z = new R();
    System.out.println(x.i);
    System.out.println(x.j);
    System.out.println(y.i);
    System.out.println(y.j);
    System.out.println(y.k);
    System.out.println(z.i);
    System.out.println(z.j);
    System.out.println(z.l);
    System.out.println(z.m);
  }
}
```

```
# class descriptors
data _class_P (sz=0):
data _class_Q (sz=0):
data _class_R (sz=0):

_main ()
(x:P, y:P, z:P)
{
  t1 = call _malloc(16)
  [t1]:P = _class_P
  call _init_P(t1)
  x = t1
  t2 = call _malloc(20)
  [t2]:P = _class_Q
  call _init_Q(t2)
  y = t2
  ...
  t4 = 8[x]:I
  call _printInt(t4)
  t5 = 12[x]:I
  call _printInt(t5)
  ...
}
```

Accessing Instance Variables (cont.)

What if an instance variable is represented by an `Id`, or the object component in the `Field` node is a `"This"` pointer?

```
class A  {
  int i = 3;
  int j = 4;
  int k = 5;
  public void sum() {
    System.out.println(i + j + this.k);
  }
}
class Demo {
  public static void main(String[] x) {
    A a = new A();
    a.sum();
  }
}
```

```
# AST Program
ClassDecl A
 VarDecl IntType i 3
 VarDecl IntType j 4
 VarDecl IntType k 5
 MethodDecl void sum ()
  Print (Binop + (Binop + i j)
                 (Field This k))
ClassDecl Demo
 MethodDecl void main ()
  VarDecl (ObjType A) a
            (NewObj A ())
  CallStmt a sum ()
```

A Solution

- Augment every method declaration with a *zero*-th parameter `obj` to represent the current object (*i.e.* `"This"` pointer).
- For every method invocation, pass the current object's address as the *zero*-th argument.
- Instance variables within the method body can then be accessed through offsets from this *zero*-th parameter.

```
data _class_A (sz=8): _A_sum

_A_sum (obj:P)
{
 t1 = 8[obj]:I
 t2 = 12[obj]:I
 t3 = t1 + t2
 t4 = 16[obj]:I
 t5 = t3 + t4
 call _printInt(t5)
 return
}
```

```
_main ()
(a:P)
{
 t1 = call _malloc(20)
 [t1]:P = _class_A
 call _init_A(t1)
 a = t1
 t2 = [a]:P
 t3 = [t2]:P
 call * t3(a)
 return
}
```

Overview of Java's Methods

```
class P {
  static void Print1() { System.out.println("1"); }
  void Print2()        { System.out.println("P2"); }
}

class Q extends P {
  void Print2()        { System.out.println("Q2"); }
  void Print3()        { System.out.println("3"); }
}

class Demo {
  public static void main(String[] a) {
    P x = new P();
    Q y = new Q();
    P z = new Q();  // mismatch between lhs and rhs' types
    P.Print1();     // Can we call x.Print1()?
    x.Print2();
    y.Print2();
    y.Print3();
    z.Print2();     // Can we call z.Print3()?
    ((P)y).Print2();
  }
}
```

Overview of Static Methods

Static methods can be invoked *only* through class names. (A static method can be invoked not only through the host class in which it is defined, but also through any sub-classes.)

```
class C1 {
  static void print(int i) { ... }
}

class C2 extends C1 {
  ...
}

class Demo {
  public static void main(String[] a) {
    C1.print(1);
    C2.print(2);
  }
}
```

Overview of Non-Static Methods

Non-static methods are invoked through objects. There are two different binding approaches, static and dynamic. Methods' binding approach is a choice of language design — some languages use static binding (e.g. C++), some use dynamic binding (e.g. Java).

- *Static Method Binding* — Uses the object's static type information to determine method binding *at compile-time*.
- *Dynamic Method Binding* — Uses the actual type of the object *at the time of invocation* to determine method binding.

CodeGen for Static Methods

Given $C.f()$, the compiler needs find the class in which f is defined.

1. Start with the class name C in the invocation. Check C's definition to see if 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.

CodeGen for Static-Binding Methods

Given $v.f()$, the compiler takes the following steps:

1. From v's declaration, figure out v's class type C.
2. Search C's definition to see if 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.

CodeGen for Dynamic-Binding Methods

With dynamic-binding, given $v.f()$, the method to invoke may not be the one defined in the host class of v.

Example:

```
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.

Static Method Table (V-Table) Approach

- In each class descriptor, keep a static method table (or *virtual* method table – v-table) containing 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:
  - Starting from the object's record, follow pointer to class descriptor;
  - Look up the method table for the method's address;
  - Jump to the address.

Issues with this Approach:

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

Alternative approach exists — Dunamic lookup.

V-Table Example

```
class Test {
  public static void main(String[] x) {
    A a = new A();
    A b = new B();
    a.f();
    b.f();
    b.g();
  }
}

class A {
  public void f() { System.out.println("A.f"); }
  public void g() { System.out.println("A.g"); }
}

class B extends A {
  public void f() { System.out.println("B.f"); }
}
```

V-Table Example (cont.)

```
data _class_A (sz=16): _A_f, _A_g
data _class_B (sz=16): _B_f, _A_g
_main ()
(a, b)
{
 t1 = call _malloc(8)
 [t1]:P = _class_A
 call _init_A(t1)
 a = t1
 t2 = call _malloc(8)
 [t2]:P = _class_B
 call _init_B(t2)
 b = t2
 t3 = [a]:P    # a.f()
 t4 = [t3]:P
 call * t4(a)
 t5 = [b]:P    # b.f()
 t6 = [t5]:P
 call * t6(b)
 t7 = [b]:P    # b.g()
 t8 = 8[t7]:P
 call * t8(b)
 return
}
```

```
_init_A (obj)
{
 return
}
_A_f (obj)
{
 call _printStr("A.f")
 return
}
_A_g (obj)
{
 call _printStr("A.g")
 return
}
_init_B (obj)
{
 return
}
_B_f (obj)
{
 call _printStr("B.f")
 return
}
```

Local Variables and Parameters

In our IR, both local variables and parameters keep their original names. This is a design choice. Other IRs may choose to represent them with lower forms (*e.g.* base address plus offsets).

The only IR issue we have is local variables' initialization. The initialization expressions need to be turned into real IR instructions, and inserted at the beginning of the instruction list that corresponds to the method's body.

IR CodeGen Summary

For classes, objects, fields, and methods:

- Need to compute offset for every class variable or instance variable. Storage layout is a key.
- Different category of methods need to be handled differently.
- Need to handle variable initialization properly. For instance variables, an `init` routine is created for the purpose.
- Need to handle `"This"` pointer properly. The solution is to pass an extra parameter to every method.