Code Generation for Objects

- OO implementation = Basic code generation + more stuff
- OO slogan: If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected
- This means that code in class A works unmodified for an object of class B

- How are objects represented in memory?
- How is dynamic dispatch implemented?

```
class A {
  a: Int <- 0;
  d: Int <- 1;
  f(): Int { a <- a + d };
};

class B inherits A {
  b: Int <- 2;
  f(): Int { a };
  g(): Int { a <- a - b };
};

class C inherits A {
  c: Int <- 3;
  h(): Int { a <- a*c };
};
```

For A methods to work correctly in A, B and C objects, attribute `a` must be in the same "place" in each object.

Object layout

- Objects are laid out in contiguous memory
- Each attribute stored at a fixed offset in the object
  - The attribute is in the same place in every object of that class
  - e.g., `this` points to the start offset of the object, and `a` is at offset 10 in all objects of type A, B, C
- When a method is invoked, the object is `this` and the fields are the object's attributes

Example layout

```
Offset   Information stored
0   Class tag
4   Object size
8   Dispatch ptr
12  Attribute 1
16  Attribute 2
...
```

- Class tag is an integer which identifies the class of the object (e.g., A = 1, B = 2, C = 3, ..)
- Object size is an integer: size of object in bytes
- Dispatch ptr is a pointer to a table of methods (more later)
- Attributes in subsequent slots
- Lay out in contiguous memory

Observation: Given a layout for class `A`, a layout for subclass `B` can be defined by *extending* the layout of `A` with additional slots for the additional attributes of `B`.

Leaves the layout of `A` unchanged (`B` is an extension).

Looking at our example:

```
Class A: {0: Atag, 4: 5, 8: *, 12: a, 16: d}
Class B: {0: Btag, 4: 6, 8: *, 12: a, 16: d, 20: b}
Class C: {0: Ctag, 4: 6, 8: *, 12: a, 16: d, 20: c}
```

The offset of an attribute is the same in a class and all of its subclasses

- Any method for an A1 can be used on a subclass A2

Consider layout for An < ... A3 < A2 < A1

```
Header     |A1 object  |            |
A1 attrs   |           |A2 object   |
A2 attrs                |            |A3 object
A3 attrs                             |
...
An attrs
```

OO Code generation

- Consider the dispatch: `e.g()`
  - Straightforward
- Consider the dispatch: `e.f()`
  - If `e` is type A: invoke A's `f`
  - If `e` is type B: invoke B's `f`
  - If `e` is type C: invoke A's `f`

Every class has a fixed set of methods

- including inherited methods

A *dispatch table* indexes these methods

- An array of method entry points
- A method `f` lives at a fixed offset in the dispatch table for a class and all of its subclasses
  - A compiler can figure out all the methods of a certain class. e.g., for A there are two methods
  - Based on this the compiler can assign a fixed offset for each method. This offset will be identical for all overriding functions of all its subclasses.

Dispatch table layout

```
Class A: {0: fA}
Class B: {0: fB, 4: g}
class C: {0: fA, 4: h}
```

- The dispatch table for class `A` contains only one method
- The tables for `B` and `C` extend the table for `A` to the right
- Because methods can be overriden, the method for `f` is not the same in every class, but is always at the same offset

The dispatch pointer in an object of class `X` points to the dispatch table for class `X`. Every method `f` of class `X` is assigned an offset `Of` in the dispatch table at compile time

To implement a dynamic dispatch `e.f()`, we

- Evaluate `e`, giving an object `x`
- Call D[Of]
  - `D` is the dispatch table for `x`
  - In the call, `this` is bound to `x`

Multiple Inheritance

If `C` is a subclass of two independent classes `A` and `B` simultaneously (`C` inherits from both `A` and `B`), then:

- The object layout involves first laying-out C's header, then laying-out A's header and attributes, then laying-out B's header and attributes, and finally laying-out C's attributes
- For each use of a C object in a context where C is expected:
  - Generate code assuming that `this` points to C's header and the object layout
- For each use of a C object in a context where A is expected:
  - Generate code to convert C's `this` to A's `this` (by adding an offset to reach A's header), and then using the code generation for A.
- For each use of a C object in a context where B is expected:
  - Generate code to convert C's `this` to B's `this` (by adding an offset to reach B's header), and then using the code generation for B.
- For methods:
  - If `C` overrides a method of `A`, we appropriately modify `A`'s dispatch table.
  - Similarly, if `C` overrides a method of `B`, we appropriately modify `B`'s dispatch table.
  - While generating dispatch code in a context where `C` is expected, if it is an overriding method, then index into the appropriate class (e.g., A or B)