Lecture 19

CIS 341: COMPILERS

Announcements / Plan

- HW5: OAT – typechecking, structs, function pointers
  - Due: Thursday, April 13

  As always, *start early*!

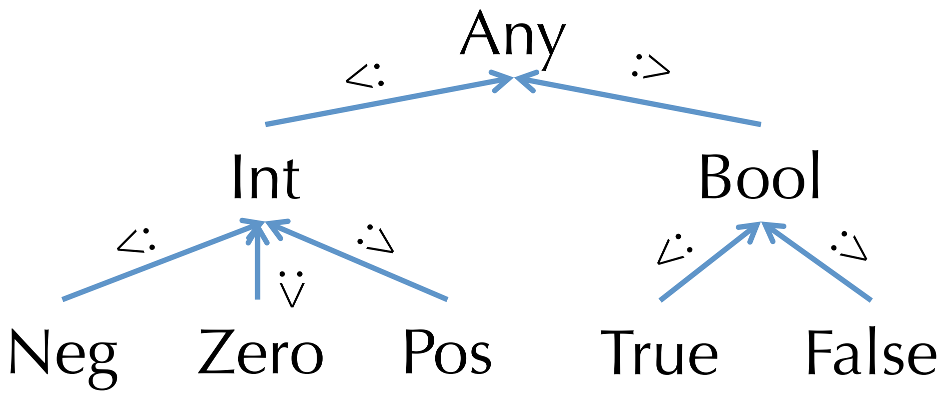
- HW6: LLVM Optimization: analysis and register allocation
  - Due: Wednesday, April 26

- FINAL EXAM: Thursday, May 4th noon – 2:00p.m.

SUBTYPING OTHER TYPES

Subtyping and Upper Bounds

- If we think of types as sets of values, we have a natural inclusion relation: Pos ⊆ Int
- This subset relation gives rise to a *subtype* relation: Pos <: Int
- Such inclusions give rise to a *subtyping hierarchy*:



- Given any two types T_1 and T_2, we can calculate their *least upper bound* (LUB) according to the hierarchy.
  - Example: LUB(True, False) = Bool, LUB(Int, Bool) = Any
  - Note: might want to add types for "NonZero", "NonNegative", and "NonPositive" so that set union on values corresponds to taking LUBs on types.

"If" Typing Rule Revisited

- For statically unknown conditionals, we want the return value to be the LUB of the types of the branches:

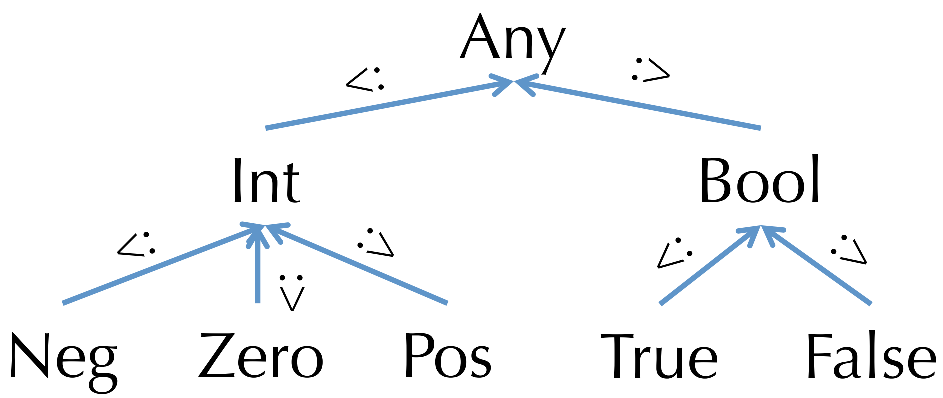
IF-BOOL

$$\frac{E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \quad E \vdash e_3 : T_2}{E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : \text{LUB}(T_1,T_2)}$$

- Note that LUB(T_1, T_2) is the most precise type (according to the hierarchy) that is able to describe any value that has either type T_1 or type T_2.
- In math notation, LUB(T1, T2) is sometimes written $T_1 \vee T_2$
- LUB is also called the *join* operation.

Subtyping Hierarchy

- A *subtyping hierarchy*:



- The subtyping relation is a *partial order*:
  - Reflexive: T <: T   for any type T
  - Transitive: T_1 <: T_2 and T_2 <: T_3 then T_1 <: T_3
  - Antisymmetric: It T_1 <: T_2 and T_2 <: T_1 then $T_1 = T_2$

Downcasting

- What happens if we have an Int but need something of type Pos?
  - At compile time, we don't know whether the Int is greater than zero.
  - At run time, we do.

- Add a "checked downcast"

$$\frac{E \vdash e_1 : \text{Int} \qquad E, x : \text{Pos} \vdash e_2 : T_2 \qquad E \vdash e_3 : T_3}{E \vdash \text{ifPos } (x = e_1)\ e_2 \text{ else } e_3 : T_2 \vee T_3}$$

- At runtime, ifPos checks whether e_1 is > 0. If so, branches to e_2 and otherwise branches to e_3.
- Inside the expression e_2, x is the name for e_1's value, which is known to be strictly positive because of the dynamic check.
- Note that such rules force the programmer to add the appropriate checks
  - We could give integer division the type: Int -> NonZero -> Int

Extending Subtyping to Other Types

- What about subtyping for tuples?
  - Intuition: whenever a program expects something of type S_1 * S_2, it is sound to give it a T_1 * T_2.
  - Example: (Pos * Neg) <: (Int * Int)

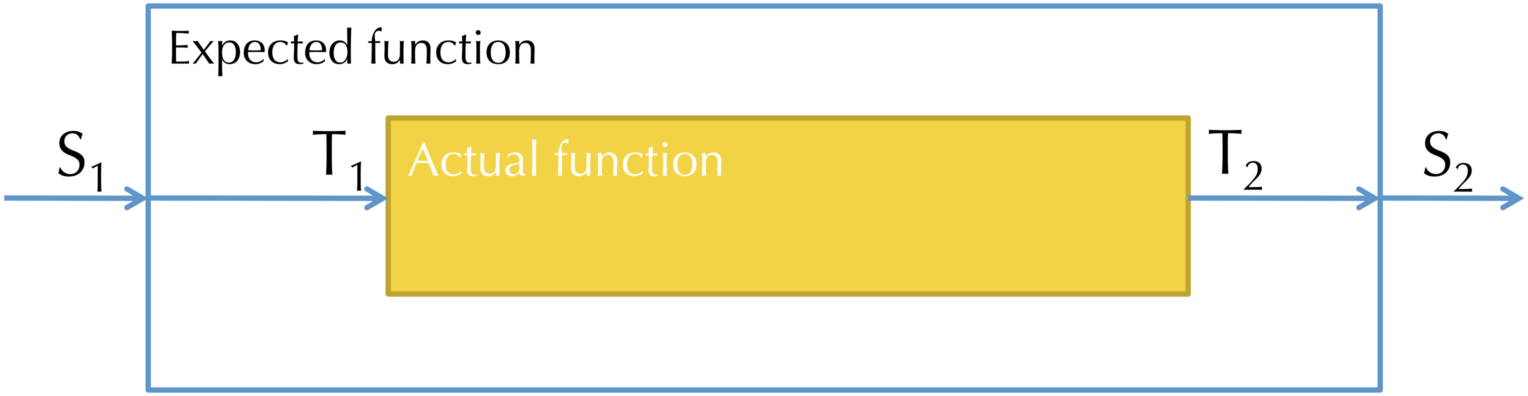
$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

- What about functions?

- When is T_1 -> T_2 <: S_1 -> S_2 ?

Subtyping for Function Types

- One way to see it:



- Need to convert an S_1 to a T_1 and T_2 to S_2, so the argument type is *contravariant* and the output type is *covariant*.

$$\frac{S_1 <: T_1 \qquad T_2 <: S_2}{(T_1 \text{ -> } T_2) <: (S_1 \text{ -> } S_2)}$$

Immutable Records

- Record type: $\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}$
  - Each lab_i is a label drawn from a set of identifiers.

RECORD

$$\frac{E \vdash e_1 : T_1 \qquad E \vdash e_2 : T_2 \qquad \ldots \qquad E \vdash e_n : T_n}{E \vdash \{lab_1 = e_1;\ lab_2 = e_2;\ \ldots\ ;\ lab_n = e_n\} : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}}$$

PROJECTION

$$\frac{E \vdash e : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}}{E \vdash e.lab_i : T_i}$$

Immutable Record Subtyping

- Depth subtyping:
  - Corresponding fields may be subtypes

DEPTH

$$\frac{T_1 <: U_1 \quad T_2 <: U_2 \quad \ldots \quad T_n <: U_n}{\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\} <: \{lab_1{:}U_1;\ lab_2{:}U_2;\ \ldots\ ;\ lab_n{:}U_n\}}$$

- Width subtyping:
  - Subtype record may have *more* fields:

WIDTH

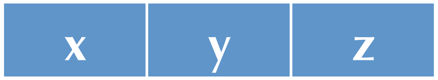
$$\frac{m \leq n}{\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\} <: \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_m{:}T_m\}}$$

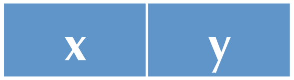
Depth & Width Subtyping vs. Layout

- Width subtyping (without depth) is compatible with "inlined" record representation as with C structs:

```
{x:int; y:int; z:int}   <:   {x:int; y:int}
```

[Width Subtyping]





  - The layout and underlying field indices for 'x' and 'y' are identical.
  - The 'z' field is just ignored

- Depth subtyping (without width) is similarly compatible, assuming that the space used by A is the same as the space used by B whenever A <: B
- But… they don't mix without

Immutable Record Subtyping (cont'd)

- Width subtyping assumes an implementation in which order of fields in a record matters:

  `{x:int; y:int}` ≠ `{y:int; x:int}`

- But: `{x:int; y:int; z:int}` <: `{x:int; y:int}`
  - Implementation: a record is a struct, subtypes just add fields at the *end* of the struct.

- Alternative: allow permutation of record fields:

  `{x:int; y:int}` = `{y:int; x:int}`

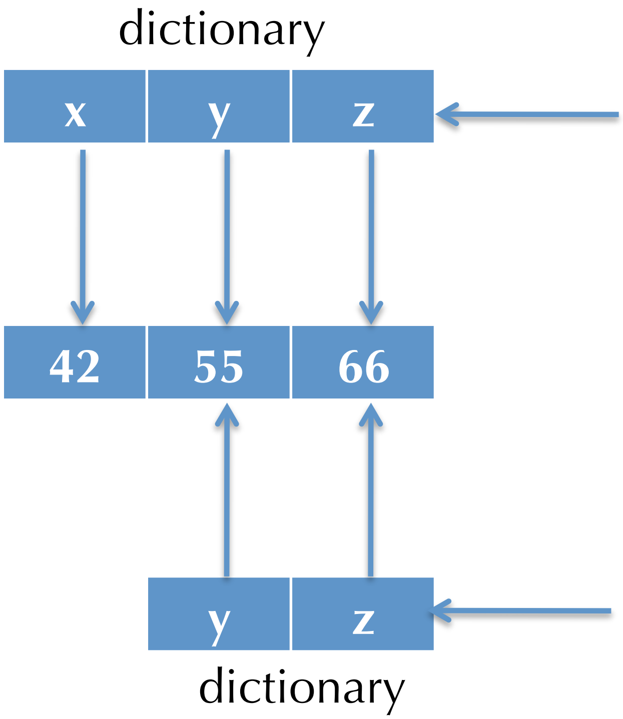
  - Implementation: compiler sorts the fields before code generation.
  - Need to know *all* of the fields to generate the code

- Permutation is not directly compatible with width subtyping:

  `{x:int; z:int; y:int}` =
  `{x:int; y:int; z:int}` </: `{y:int; z:int}`

If you want both:

- If you want permutability & dropping, you need to either copy (to rearrange the fields) or use a dictionary like this:



```
p = {x=42; y=55; z=66}
  : {x:int; y:int; z:int}
```

```
q : {y:int; z:int}
```

Subtyping and References

- What is the proper subtyping relationship for references and arrays?

- Suppose we have NonZero as a type and the division operation has type: Int -> NonZero -> Int
  - Recall that NonZero <: Int
- Should (NonZero ref) <: (Int ref) ?
- Consider this program:

```
Int bad(NonZero ref r) {
  Int ref a = r;   (* OK because (NonZero ref <: Int ref*)
  a := 0;          (* OK because 0 : Zero <: Int *)
  return (42 / !r) (* OK because !r has type NonZero *)
}
```

Mutable Structures are Invariant

- Covariant reference types are *unsound*
  - As demonstrated in the previous example
- Contravariant reference types are also unsound
  - i.e. If T_1 <: T_2 then ref T_2 <: ref T_1 is also unsound
  - Exercise: construct a program that breaks contravariant references.

- Moral: Mutable structures are invariant:

  T_1 ref <: T_2 ref implies $T_1 = T_2$

- Same holds for arrays, OCaml-style mutable records, object fields, etc.
  - Java generics are invariant for this reason too:
    Queue<String> </: Queue<Object>

  - Note: Java and C# get subtyping of arrays wrong. They allows covariant array subtyping, but then compensate by adding a dynamic check on *every* array update!

Another Way to See It

- We can think of a reference cell as an immutable record (object) with two functions (methods) and some hidden state:

  T ref ≃ `{get: unit -> T;   set: T -> unit}`

  – get returns the value hidden in the state.

  – set updates the value hidden in the state.

- When is T ref <: S ref?
- Consider depth subtyping of these records…

  `{get: unit -> T; set: T -> unit}` <:
  `{get: unit -> S; set: S -> unit}`

  – `get` components are subtypes: unit -> T <: unit -> S
  `set` components are subtypes: T -> unit <: S -> unit

- From get, we must have T <: S (covariant return)
- From set, we must have S <: T (contravariant arg.)
- From T <: S and S <: T we conclude T = S.

STRUCTURAL VS. NOMINAL TYPES

Structural vs. Nominal Typing

- Is type equality / subsumption defined by the *structure* of the data or the *name* of the data?
- Example 1: type abbreviations (OCaml) vs. “newtypes” (a la Haskell)

```
(* OCaml: *)
type cents = int    (* cents = int in this scope *)
type age = int

let foo (x:cents) (y:age) = x + y
```

```
(* Haskell: *)
newtype Cents = Cents Integer  (* Integer and Cents arr
                                isomorphic, not identical. *)
newtype Age = Age Integer

foo :: Cents -> Age -> Int
foo x y = x + y                (* Ill typed! *)
```

- Type abbreviations are treated “structurally”
  Newtypes are treated “by name”

Nominal Subtyping in Java

- In Java, Classes and Interfaces must be named and their relationships *explicitly* declared:

```
(* Java: *)
interface Foo {
  int foo();
}

class C {       /* Does not implement the Foo interface */
  int foo() {return 2;}
}

class D implements Foo {
  int foo() {return 341;}
}
```

- Similarly for inheritance: programmers must declare the subclass relation via the "`extends`" keyword.
  - Typechecker still checks that the classes are structurally compatible

COMPILING CLASSES AND OBJECTS

Code Generation for Objects

- Classes:
  - Generate data structure types
    - For objects that are instances of the class and for the class tables
  - Generate the class tables for dynamic dispatch
- Methods:
  - Method body code is similar to functions/closures
  - Method calls require *dispatch*
- Fields:
  - Issues are the same as for records
  - Generating access code
- Constructors:
  - Object initialization
- Dynamic Types:
  - Checked downcasts
  - "instanceof" and similar type dispatch

Multiple Implementations

- The same interface can be implemented by multiple classes:

```
interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
    public int size();
}
```

```
class IntSet1 implements IntSet {
  private List<Integer> rep;
  public IntSet1() {
    rep = new LinkedList<Integer>();}

  public IntSet1 insert(int i) {
    rep.add(new Integer(i));
    return this;}

  public boolean has(int i) {
    return rep.contains(new Integer(i));}

  public int size() {return rep.size();}
}
```

```
class IntSet2 implements IntSet {
  private Tree rep;
  private int size;
  public IntSet2() {
    rep = new Leaf(); size = 0;}

  public IntSet2 insert(int i) {
    Tree nrep = rep.insert(i);
    if (nrep != rep) {
      rep = nrep; size += 1;
    }
    return this;}

  public boolean has(int i) {
    return rep.find(i);}

  public int size() {return size;}
}
```

The Dispatch Problem

- Consider a client program that uses the IntSet interface:

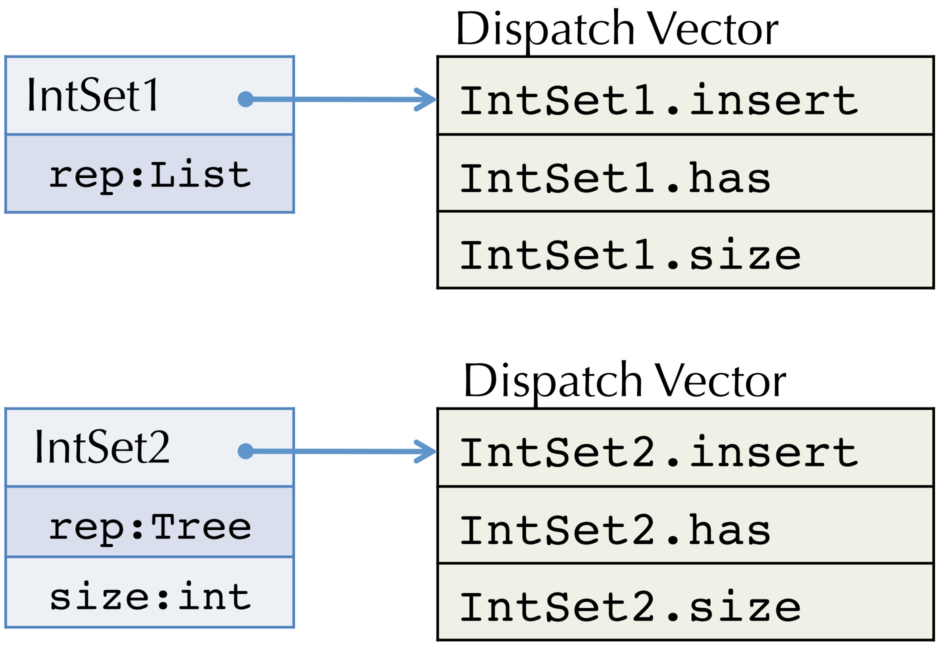
```
IntSet set = …;
int x = set.size();
```

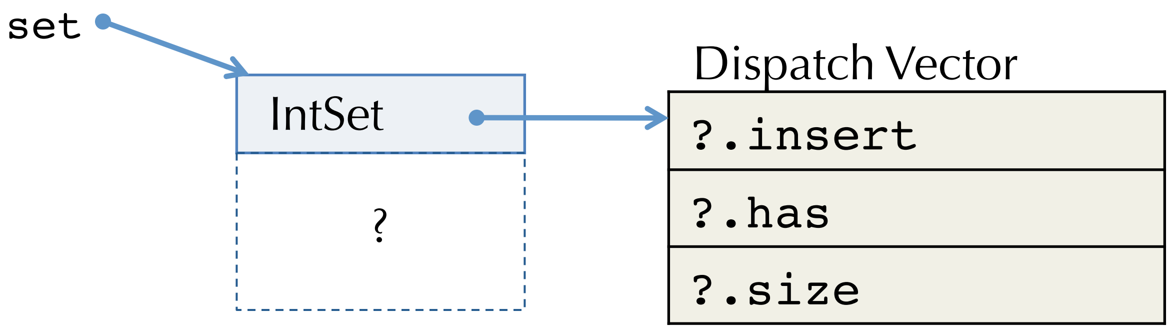
- Which code to call?
  - `IntSet1.size` ?
  - `IntSet2.size` ?
- Client code doesn't know the answer.
  - So objects must "know" which code to call.
  - Invocation of a method must indirect through the object.

Compiling Objects

- Objects contain a pointer to a *dispatch vector* (also called a *virtual table* or *vtable*) with pointers to method code.

- Code receiving `set:IntSet` only knows that `set` has an initial dispatch vector pointer and the layout of that vector.





Method Dispatch (Single Inheritance)

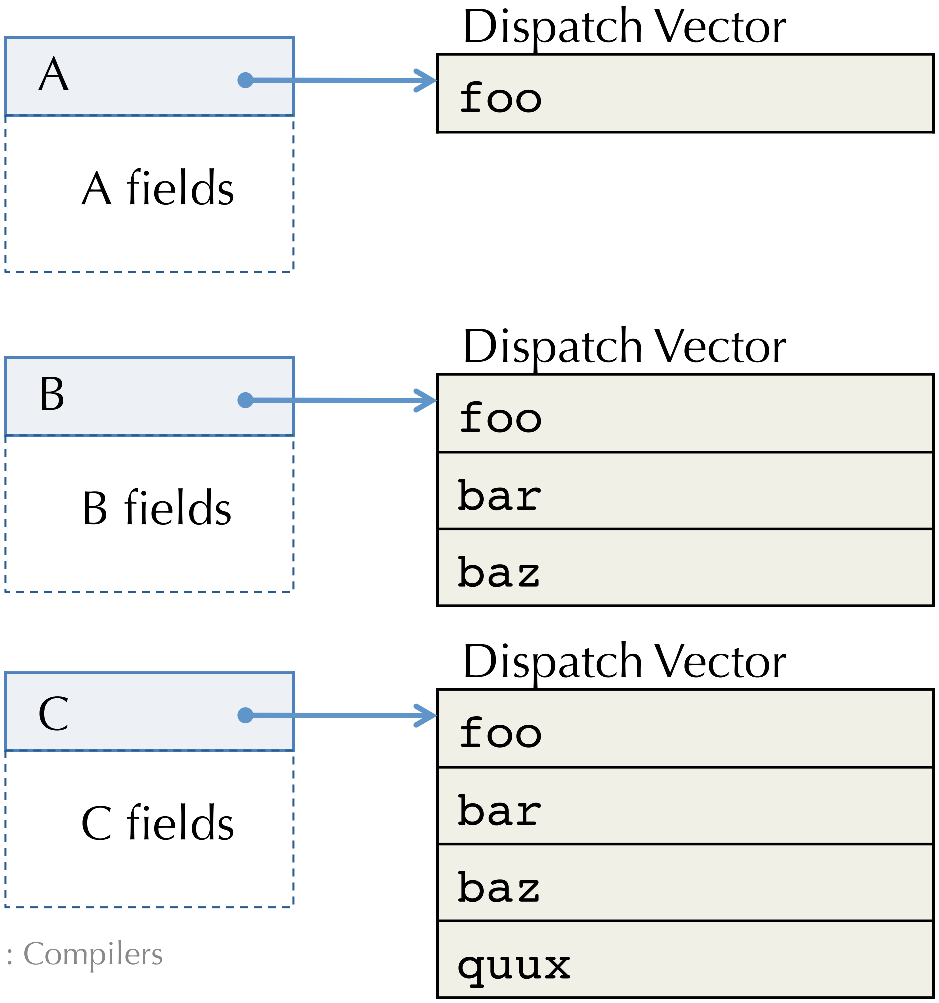
- Idea: every method has its own small integer index.
- Index is used to look up the method in the dispatch vector.

```
                                  Index
interface A {
  void foo();                      0
}

interface B extends A {
  void bar(int x);                 1
  void baz();                      2
}

class C implements B {
  void foo() {…}                   0
  void bar(int x) {…}              1
  void baz() {…}                   2
  void quux() {…}                  3
}
```

Inheritance / Subtyping:
C <: B <: A

Dispatch Vector Layouts

- Each interface and class gives rise to a dispatch vector layout.
- Note that inherited methods have identical dispatch indices in the subclass. (Width subtyping)



Representing Classes in the LLVM

- During typechecking, create a *class hierarchy*
  - Maps each class to its interface:
    - Superclass
    - Constructor type
    - Fields
    - Method types (plus whether they inherit & which class they inherit from)

- Compile the class hierarchy to produce:
  - An LLVM IR struct type for each object instance
  - An LLVM IR struct type for each vtable (a.k.a. class table)
  - Global definitions that implement the class tables

Example OO Code

```
class A {
  new (int x)()                    // constructor
  { int x = x; }

  void print() { return; }         // method1
  int blah(A a) { return 0; }      // method2

}

class B <: A {
  new (int x, int y, int z)(x){
    int y = y;
    int z = z;
  }

  void print() { return; }       // overrides A
}

class C <: B {
  new (int x, int y, int z, int w)(x,y,z){
    int w = w;
  }

  void foo(int a, int b) {return;}
  void print() {return;}       // overrides B
}
```

Example OO Hierarchy in LLVM

```
%Object = type { %_class_Object* }
%_class_Object = type {  }

%A = type { %_class_A*, i64 }
%_class_A = type { %_class_Object*, void (%A*)*, i64 (%A*, %A*)* }

%B = type { %_class_B*, i64, i64, i64 }
%_class_B = type { %_class_A*, void (%B*)*, i64 (%A*, %A*)* }

%C = type { %_class_C*, i64, i64, i64, i64 }
%_class_C = type { %_class_B*, void (%C*)*, i64 (%A*, %A*)*, void (%C*, i64, i64)* }

@_vtbl_Object = global %_class_Object {  }

@_vtbl_A = global %_class_A { %_class_Object* @_vtbl_Object,
                              void (%A*)* @print_A,
                              i64 (%A*, %A*)* @blah_A }

@_vtbl_B = global %_class_B { %_class_A* @_vtbl_A,
                              void (%B*)* @print_B,
                              i64 (%A*, %A*)* @blah_A }

@_vtbl_C = global %_class_C { %_class_B* @_vtbl_B,
                              void (%C*)* @print_C,
                              i64 (%A*, %A*)* @blah_A,
                              void (%C*, i64, i64)* @foo_C }
```