Lecture 19

CIS 341: COMPILERS
Announcements

• HW5: Oat v. 2.0
  – records, function pointers, type checking, array-bounds checks, etc.
  – typechecker & safety
  – Due: Friday, April 17th
  – *Please start soon (if you haven’t already!)*

• Oat Syntax Highlighting for VSCode
  – See Piazza post
RECAP: SUBTYPING
Subtyping and Upper Bounds

• If we think of types as sets of values, we have a natural inclusion relation: \( \text{Pos} \subseteq \text{Int} \)
• This subset relation gives rise to a subtype relation: \( \text{Pos} <: \text{Int} \)
• Such inclusions give rise to a subtyping hierarchy:

```
  Any  <=  Int  =>  Bool
    v
  Neg <= Zero <= Pos
    v
  True <= False
```

• Given any two types \( T_1 \) and \( T_2 \), we can calculate their least upper bound (LUB) according to the hierarchy.
  – Example: \( \text{LUB}(\text{True}, \text{False}) = \text{Bool} \), \( \text{LUB}(\text{Int}, \text{Bool}) = \text{Any} \)
  – Note: might want to add types for “NonZero”, “NonNegative”, and “NonPositive” so that set union on values corresponds to taking LUBs on types.
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 \textit{contravariant} and the output type is \textit{covariant}.

  

  $S_1 <: T_1 \quad T_2 <: S_2$

  

  $(T_1 \rightarrow T_2) <: (S_1 \rightarrow S_2)$
Immutable Record Subtyping

• Depth subtyping:
  – Corresponding fields may be subtypes

\[
\begin{align*}
\text{DEPTH} & \quad T_1 <: U_1 \quad T_2 <: U_2 \quad \ldots \quad T_n <: U_n \\
& \quad \{\text{lab}_1:T_1; \text{lab}_2:T_2; \ldots; \text{lab}_n:T_n\} <: \{\text{lab}_1:U_1; \text{lab}_2:U_2; \ldots; \text{lab}_n:U_n\}
\end{align*}
\]

• Width subtyping:
  – Subtype record may have *more* fields:

\[
\begin{align*}
\text{WIDTH} & \quad m \leq n \\
& \quad \{\text{lab}_1:T_1; \text{lab}_2:T_2; \ldots; \text{lab}_n:T_n\} <: \{\text{lab}_1:T_1; \text{lab}_2:T_2; \ldots; \text{lab}_m:T_m\}
\end{align*}
\]
MUTABILITY & SUBTYPING
• What is the type of null?
• Consider:
  
  ```java
  int[] a = null;  // OK?
  int x   = null;  // not OK?
  string s = null;  // OK?
  ```

  $G \vdash \text{null} : r$

• Null has any reference type
  – Null is generic

• What about type safety?
  – Requires defined behavior when dereferencing null
e.g. Java's NullPointerException
  – Requires a safety check for every dereference operation
    (typically implemented using low-level hardware "trap" mechanisms.)
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:

```plaintext
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 $\text{ref } T_2 <: \text{ref } T_1$ is also unsound
  - Exercise: construct a program that breaks contravariant references.

- Moral: Mutable structures are invariant:
  $$T_1 \text{ref } <: T_2 \text{ref } \implies T_1 = T_2$$

- Same holds for arrays, OCaml-style mutable records, object fields, etc.
  - Note: Java and C# get this wrong. They allow 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:

\[
T \text{ ref } \simeq \{\text{get: unit } \rightarrow T; \ \text{set: } T \rightarrow \text{unit}\}
\]

– get returns the value hidden in the state.
– set updates the value hidden in the state.

• When is \( T \text{ ref } <: S \text{ ref} \)?

• Records are like tuples: subtyping extends pointwise over each component.

• \( \{\text{get: unit } \rightarrow T; \ \text{set: } T \rightarrow \text{unit}\} <: \{\text{get: unit } \rightarrow S; \ \text{set: } S \rightarrow \text{unit}\} \)

  – get components are subtypes: \( \text{unit } \rightarrow T <: \text{unit } \rightarrow S \)
  – set components are subtypes: \( T \rightarrow \text{unit } <: S \rightarrow \text{unit} \)

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

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

(* Haskell: *)
```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
/* 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
See oat.pdf in HW5

OAT'S TYPE SYSTEM
OAT's Treatment of Types

- Primitive (non-reference) types:
  - `int`, `bool`
- Definitely non-null reference types: `R`
  - (named) mutable structs with (right-oriented) width subtyping
  - `string`
  - arrays (including length information, per HW4)
- Possibly-null reference types: `R?`
  - Subtyping: `R <: R?`
  - Checked downcast syntax `if?:`

```java
int sum(int[]? arr) {
    var z = 0;
    if?(int[] a = arr) {
        for(var i = 0; i<length(a); i = i + 1;) {
            z = z + a[i];
        }
    }
    return z;
}
```
OAT Features

• Named structure types with mutable fields
  – but using structural, width subtyping

• Typed function pointers

• Polymorphic operations: length and == / !=
  – need special case handling in the typechecker

• Type-annotated null values: t null always has type t?

• Definitely-not-null values means we need an "atomic" array initialization syntax
  – for example, null is not allowed as a value of type int[], so to construct a record containing a field of type int[], we need to initialize it
  – subtlety: int[][] cannot be initialized by default, but int[] can be
OAT "Returns" Analysis

- Typesafe, statement-oriented imperative languages like OAT (or Java) must ensure that a function (always) returns a value of the appropriate type.
  - Does the returned expression's type match the one declared by the function?
  - Do all paths through the code return appropriately?

- OAT's statement checking judgment
  - Takes the expected return type as input: what type should the statement return (or `void` if none)
  - Produces a boolean flag as output: does the statement definitely return?
Closures
Objects
Dynamic Dispatch

COMPILING HIGHER-ORDER FEATURES
Compiling lambda calculus to straight-line code.
Representing evaluation environments at runtime.
Compiling First-class Functions

• To implement first-class functions on a processor, there are two problems:
  – First: we must implement substitution of free variables
  – Second: we must separate ‘code’ from ‘data’

• Reify the substitution:
  – Move substitution from the meta language to the object language by making the data structure & lookup operation explicit
  – The environment-based interpreter is one step in this direction

• Closure Conversion:
  – Eliminates free variables by packaging up the needed environment in the data structure.

• Hoisting:
  – Separates code from data, pulling closed code to the top level.
See: fun.ml “closure-based” interpreter
cc.ml

CODE EXAMPLE
Example of closure creation

- Recall the “add” function:
  \[
  \text{let add} = \text{fun } x \rightarrow \text{fun } y \rightarrow x + y
  \]

- Consider the inner function: \( \text{fun } y \rightarrow x + y \)

- When run the function application: \texttt{add 4} the program builds a closure and returns it.
  - The closure is a pair of the environment and a code pointer.

\[
\begin{array}{|c|c|}
\hline
\text{ptr} & \text{Code(env, y, body)} \\
\hline
(4) & \text{code body} \\
\hline
\end{array}
\]

- The code pointer takes a pair of parameters: env and y
  - The function code is (essentially):
    \[
    \text{fun (env, y)} \rightarrow \text{let } x = \text{nth env 0 in } x + y
    \]
Representing Closures

- As we saw, the simple closure conversion algorithm doesn’t generate very efficient code.
  - It stores all the values for variables in the environment, even if they aren’t needed by the function body.
  - It copies the environment values each time a nested closure is created.
  - It uses a linked-list datastructure for tuples.

- There are many options:
  - Store only the values for free variables in the body of the closure.
  - Share subcomponents of the environment to avoid copying
  - Use vectors or arrays rather than linked structures
Array-based Closures with N-ary Functions

(fun (x y z) -> (fun (n m) -> (fun p -> (fun q -> n + z) x)))

Note how free variables are “addressed” relative to the closure due to shared env.
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:

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

```java
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();}
}
```

```java
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:

```java
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.
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.

```java
interface A {
    void foo();
}

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

class C implements B {
    void foo() {...}
    void bar(int x) {...}
    void baz() {...}
    void quux() {...}
}
```

<table>
<thead>
<tr>
<th>Interface</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

Inheritance / Subtyping:

C <: B <: A
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 (Java)

```java
class A {
    A (int x) { super(); int x = x; }  // constructor
    void print() { return; }  // method1
    int blah(A a) { return 0; }  // method2
}

class B extends A {
    B (int x, int y, int z) {
        super(x);
        int y = y;
        int z = z;
    }
    void print() { return; }  // overrides A
}

class C extends B {
    C (int x, int y, int z, int w) {
        super(x, y, z);
        int w = w;
    }
    void foo(int a, int b) { return; }
    void print() { return; }  // overrides B
}
```
Example OO Hierarchy in LLVM

```c
%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 }
```
• Methods bodies are compiled just like top-level procedures...
• ... except that they have an implicit extra argument: this or self
  – Historically (Smalltalk), these were called the “receiver object”
  – Method calls were thought of a sending “messages” to “receivers”

A method in a class...

```
class IntSet1 implements IntSet {
    ...
    IntSet1 insert(int i) { <body> }
}
```

... is compiled like this (top-level) procedure:

```
IntSet1 insert(IntSet1 this, int i) { <body> }
```

• Note 1: the type of “this” is the class containing the method.
• Note 2: references to fields inside <body> are compiled like this.field
LLVM Method Invocation Compilation

- Consider method invocation:
  \[ [H;G;L \vdash e.m(e_1, \ldots, e_n) : t] \]

- First, compile \([H;G;L \vdash e : C]\) to get a (pointer to) an object value of class type \(C\)
  - Call this value \(\text{obj	extunderscore ptr}\)

- Use \text{Getelementptr} to extract the vtable pointer from \(\text{obj	extunderscore ptr}\)

- Load the vtable pointer

- Use \text{Getelementptr} to extract the address of the function pointer from the vtable
  - using the information about \(C\) in \(H\)

- Load the function pointer

- Call through the function pointer, passing \('\text{obj	extunderscore ptr}'\) for this:
  \[
  \text{call (cmp	extunderscore typ t) m(obj	extunderscore ptr, } [e_1], \ldots, [e_n])\]

- In general, function calls may require \text{bitcast} to account for subtyping: arguments may be a subtype of the expected “formal” type
• Suppose b : B
• What code for b.bar(3)?
  – bar has index 1
  – Offset = 8 * 1

movq [b], %rax
movq [%rax], %rbx
movq [rbx+8], %rcx     // D.V. + offset
movq %rax, %rdi       // “this” pointer
movq 3, %rsi          // Method argument
call %ecx             // Indirect call
Sharing Dispatch Vectors

• All instances of a class may share the same dispatch vector.
  – Assuming that methods are immutable.
• Code pointers stored in the dispatch vector are available at link time – dispatch vectors can be built once at link time.

• One job of the object constructor is to fill in the object’s pointer to the appropriate dispatch vector.
• Note: The address of the D.V. is the run-time representation of the object’s type.
Inheritance: Sharing Code

• Inheritance: Method code “copied down” from the superclass
  – If not overridden in the subclass
• Works with separate compilation – superclass code not needed.

CIS 341: Compilers
Compiling Static Methods

• Java supports \textit{static} methods
  – Methods that belong to a class, not the instances of the class.
  – They have no “this” parameter (no receiver object)

• Compiled exactly like normal top-level procedures
  – No slots needed in the dispatch vectors
  – No implicit “this” parameter

• They’re not really methods
  – They can only access static fields of the class
Compiling Constructors

• Java and C++ classes can declare constructors that create new objects.
  – Initialization code may have parameters supplied to the constructor
  – e.g. `new Color(r,g,b);`

• Modula-3: object constructors take no parameters
  – e.g. `new Color;`
  – Initialization would typically be done in a separate method.

• Constructors are compiled just like static methods, except:
  – The “this” variable is initialized to a newly allocated block of memory big enough to hold D.V. pointer + fields according to object layout
  – Constructor code initializes the fields
    • What methods (if any) are allowed?
  – The D.V. pointer is initialized
    • When? Before/After running the initialization code?
Compiling Checked Casts

• How do we compile downcast in general? Consider this generalization of Oat's checked cast:

\[
\text{if? (t x = exp) \{ ... \} else \{ ... \}}
\]

• Reason by cases:
  – t must be either null, ref or ref? (can’t be just int or bool)
• If t is null:
  – The static type of exp must be ref? for some ref.
  – If exp == null then take the true branch, otherwise take the false branch
• If t is string or t[]:
  – The static type of exp must be the corresponding string? Or t[]?
  – If exp == null take the false branch, otherwise take the true branch
• If t is C:
  – The static type of exp must be D or D? (where C <: D)
  – If exp == null take the false branch, otherwise:
  – emit code to walk up the class hierarchy starting at D, looking for C
  – If found, then take true branch else take false branch
• If t is C?:
  – The static type of exp must be D? (where C <: D)
  – If exp == null take the true branch, otherwise:
  – Emit code to walk up the class hierarchy starting at D, looking for C
  – If found, then take true branch else take false branch
“Walking up the Class Hierarchy”

• A non-null object pointer refers to an LLVM struct with a type like:
  \[
  %B = \text{type} \{ \text{_class} \text{-} B*, \text{i64}, \text{i64}, \text{i64} \}
  \]

• The first entry of the struct is a pointer to the vtable for Class B
  – This pointer is the dynamic type of the object.
  – It will have the value \(@\text{vtbl} \text{-} B\)

• The first entry of the class table for B is a pointer to its superclass:
  \[
  @\text{vtbl} \text{-} B = \text{global} \ %\text{-}class \ B \ \{ \ %\text{-}class \ A*, \text{void} (\%B*)* @\text{print} \ B, \\
  \text{i64} (\%A*, \%A*)* @\text{blah} \ A \}
  \]

• Therefore, to find out whether an unknown type X is a subtype of C:
  – Assume C is not Object (ruled out by “silliness” checks for downcast)
  LOOP:
  – If X == @vtbl_Object then NO, X is not a subtype of C
  – If X == @vtbl_C then YES, X is a subtype of C
  – If X = @vtbl_D, so set X to @vtbl_E where E is D’s parent and goto LOOP
MULTIPLE INHERITANCE
Multiple Inheritance

• C++: a class may declare more than one superclass.
• Semantic problem: Ambiguity

```cpp
class A { int m(); }
class B { int m(); }
class C extends A,B {...} // which m?
```

– Same problem can happen with fields.
– In C++, fields and methods can be duplicated when such ambiguity arises
  (though explicit sharing can be declared too)

• Java: a class may implement more than one interface.
  – No semantic ambiguity: if two interfaces contain the same method declaration, then the class will implement a single method

```java
interface A { int m(); }
interface B { int m(); }
class C implements A,B {int m() {...}} // only one m
```
interface Shape {
    void setCorner(int w, Point p);
}

interface Color {
    float get(int rgb);
    void set(int rgb, float value);
}

class Blob implements Shape, Color {
    void setCorner(int w, Point p) {...} 0?
    float get(int rgb) {...} 0?
    void set(int rgb, float value) {...} 1?
}
General Approaches

• Can’t directly identify methods by position anymore.

• Option 1: Use a level of indirection:
  – Map method identifiers to code pointers (e.g. index by method name)
  – Use a hash table
  – May need to do search up the class hierarchy

• Option 2: Give up separate compilation
  – Use “sparse” dispatch vectors, or binary decision trees
  – Must know then entire class hierarchy

• Option 3: Allow multiple D.V. tables (C++)
  – Choose which D.V. to use based on static type
  – Casting from/to a class may require run-time operations

• Note: many variations on these themes
  – Different Java compilers pick different approaches to options1 and 2…
Option 1: Search + Inline Cache

• For each class & interface keep a table mapping method names to method code
  – Recursively walk up the hierarchy looking for the method name
• Note: Identifiers are in quotes are not strings; in practice they are some kind of unique identifier.

<table>
<thead>
<tr>
<th>Interface Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>“setCorner”</td>
</tr>
<tr>
<td>“get”</td>
</tr>
<tr>
<td>“set”</td>
</tr>
</tbody>
</table>

```
 Blob
 Blob fields
```

```
Class Info

“Blob”
super
itable
setCorner
get
set
```

```__get: <code>```
Inline Cache Code

- Optimization: At call site, store class and code pointer in a cache
  - On method call, check whether class matches cached value
- Compiling: \texttt{Shape s = new Blob(); s.get();}
  \hspace{1cm} Call site 434
- Compiler knows that \texttt{s} is a \texttt{Shape}
  - Suppose \texttt{%rax} holds object pointer

- Cached interface dispatch:
  
  ```
  // set up parameters
  movq [%rax], tmp s
  cmpq tmp, [cacheClass434]
  Jnz __miss434
  callq [cacheCode434]
  __miss434:
  // do the slow search
  ```

Table in data seg.

- cacheClass434: "Blob"
- cacheCode434: <ptr>
Option 1 variant 2: Hash Table

- Idea: don’t try to give all methods unique indices
  - Resolve conflicts by checking that the entry is correct at dispatch
- Use hashing to generate indices
  - Range of the hash values should be relatively small
  - Hash indices can be pre computed, but passed as an extra parameter

```java
interface Shape {
    void setCorner(int w, Point p);   hash("setCorner") = 11
}

interface Color {
    float get(int rgb);   hash("get") = 4
    void set(int rgb, float value);   hash("set") = 7
}

class Blob implements Shape, Color {
    void setCorner(int w, Point p) {...}   11
    float get(int rgb) {...}   4
    void set(int rgb, float value) {...}   7
}
```
Dispatch with Hash Tables

• What if there is a conflict?
  – Entries containing several methods point to code that resolves conflict (e.g. by searching through a table based on class name)

• Advantage:
  – Simple, basic code dispatch is (almost) identical
  – Reasonably efficient

• Disadvantage:
  – Wasted space in DV
  – Extra argument needed for resolution
  – Slower dispatch if conflict
Option 2 variant 1: Sparse D.V. Tables

- Give up on separate compilation…
- Now we have access to the whole class hierarchy.

- So: ensure that no two methods in the same class are allocated the same D.V. offset.
  - Allow holes in the D.V. just like the hash table solution
  - Unlike hash table, there is never a conflict!

- Compiler needs to construct the method indices
  - Graph coloring techniques can be used to construct the D.V. layouts in a reasonably efficient way (to minimize size)
  - Finding an optimal solution is NP complete!
Example Object Layout

- Advantage: Identical dispatch and performance to single-inheritance case
- Disadvantage: Must know entire class hierarchy
Option 2 variant 2: Binary Search Trees

- Idea: Use conditional branches not indirect jumps
- Each object has a class index (unique per class) as first word
  - Instead of D.V. pointer (no need for one!)
- Method invocation uses range tests to select among $n$ possible classes in $\lg n$ time
  - Direct branches to code at the leaves.

```assembly
Shape x;
x.SetCorner(...);
      Mov eax, [x]
      Mov ebx, [eax]
      Cmp ebx, 1
      Jle __L1
      Cmp ebx, 2
      Je __CircleSetCorner
      Jmp __EggSetCorner
__L1:
      Cmp ebx, 0
      Je __BlobSetCorner
      Jmp __RectangleSetCorner
```

Shape

Decision tree
Search Tree Tradeoffs

• Binary decision trees work well if the distribution of classes that may appear at a call site is skewed.
  – Branch prediction hardware eliminates the branch stall of ~10 cycles (on X86)

• Can use profiling to find the common paths for each call site individually
  – Put the common case at the top of the decision tree (so less search)
  – 90%/10% rule of thumb: 90% of the invocations at a call site go to the same class

• Drawbacks:
  – Like sparse D.V.’s you need the whole class hierarchy to know how many leaves you need in the search tree.
  – Indirect jumps can have better performance if there are >2 classes (at most one mispredict)
Option 3: Multiple Dispatch Vectors

- Duplicate the D.V. pointers in the object representation.
- Static type of the object determines which D.V. is used.

```java
interface Shape {
    void setCorner(int w, Point p);
}

interface Color {
    float get(int rgb);
    void set(int rgb, float value);
}

class Blob implements Shape, Color {
    void setCorner(int w, Point p) {...}
    float get(int rgb) {...}
    void set(int rgb, float value) {...}
}
```
Multiple Dispatch Vectors

- A reference to an object might have multiple “entry points”
  - Each entry point corresponds to a dispatch vector
  - Which one is used depends on the statically known type of the program.

```java
Blob b = new Blob();
Color y = b; // implicit cast!
```

- Compile
  ```java
  Color y = b;
  As
  Movq [b] + 8, y
  ```
Multiple D.V. Summary

- Benefit: Efficient dispatch, same cost as for multiple inheritance
- Drawbacks:
  - Cast has a runtime cost
  - More complicated programming model... hard to understand/debug?

- What about multiple inheritance and fields?
Multiple Inheritance: Fields

- Multiple supertypes (Java): methods conflict (as we saw)
- Multiple inheritance (C++): fields can also conflict
- Location of the object’s fields can no longer be a constant offset from the start of the object.

```java
class Color {
    float r, g, b; /* offsets: 4, 8, 12 */
}
class Shape {
    Point LL, UR; /* offsets: 4, 8 */
}
class ColoredShape extends Color, Shape {
    int z;
}
```
C++ approach:

- Add pointers to the superclass fields
  - Need to have multiple dispatch vectors anyway (to deal with methods)
- Extra indirection needed to access superclass fields
- Used even if there is a single superclass
  - Uniformity
Observe: Closure \(\approx\) Single-method Object

- Free variables
- Environment pointer
- Closure for function:

```plaintext
fun (x,y) ->
  x + y + a + b
```

- Fields
- “this” parameter
- Instance of this class:

```plaintext
class C {
  int a, b;
  int apply(x, y) {
    x + y + a + b
  }
}
```

---

Diagram:

- `env` connected to `a` and `b` with `__apply`
- `a` and `b` with `__apply: <code>`
- `D.V.` connected to `a` and `b` with `__apply`
- `__apply: <code>`