Lecture 20

CIS 341: COMPILERS

Announcements

- HW5: Oat v. 2.0
  - records, function pointers, type checking, array-bounds checks, etc.
  - typechecker & safety
  - Due: Wednesday, April 13^{th}

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

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:

```
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 = type { %_class_B*, i64, i64, 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 @vtbl_B
- The first entry of the class table for B is a pointer to its superclass:

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

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);
  void baz();
}
```

1
2

Inheritance / Subtyping:
C <: B <: A

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

0
1
2
3

Multiple Inheritance

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

```
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

```
interface A { int m(); }
interface B { int m(); }
class C implements A,B {int m() {…}}     // only one m
```

Dispatch Vector Layout Strategy Breaks

```
interface Shape {                                D.V.Index
  void setCorner(int w, Point p);                    0
}

interface Color {
  float get(int rgb);                                0
  void set(int rgb, float value);                    1
}

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 2 variant 1: Sparse D.V. Tables

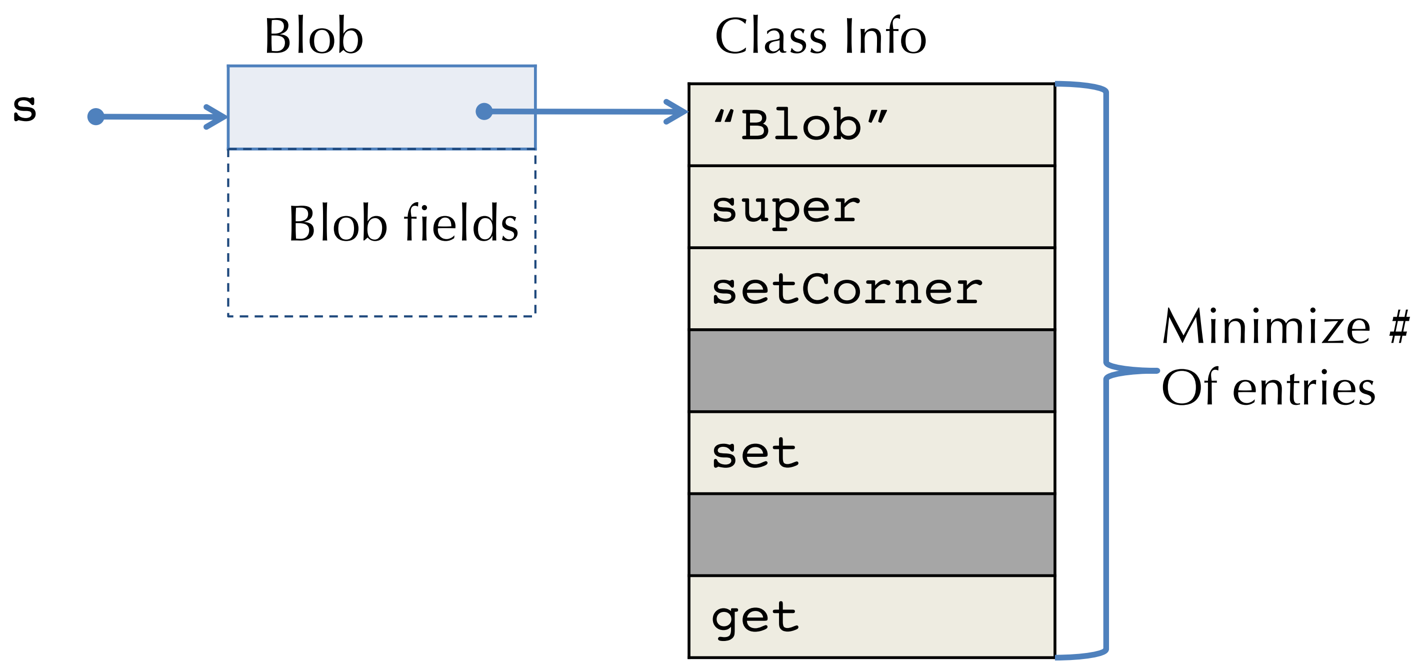
- Give up on separate compilation…
- Now we have access to the whole class hierarchy.

- 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

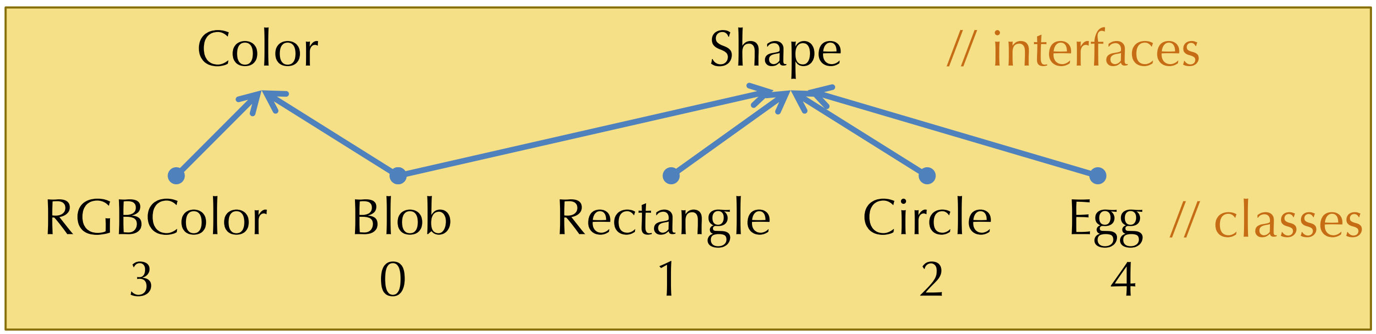


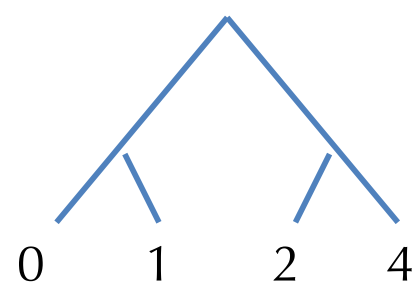
Option 2 variant 2: Binary Search Trees

- 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.

```
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
```





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)

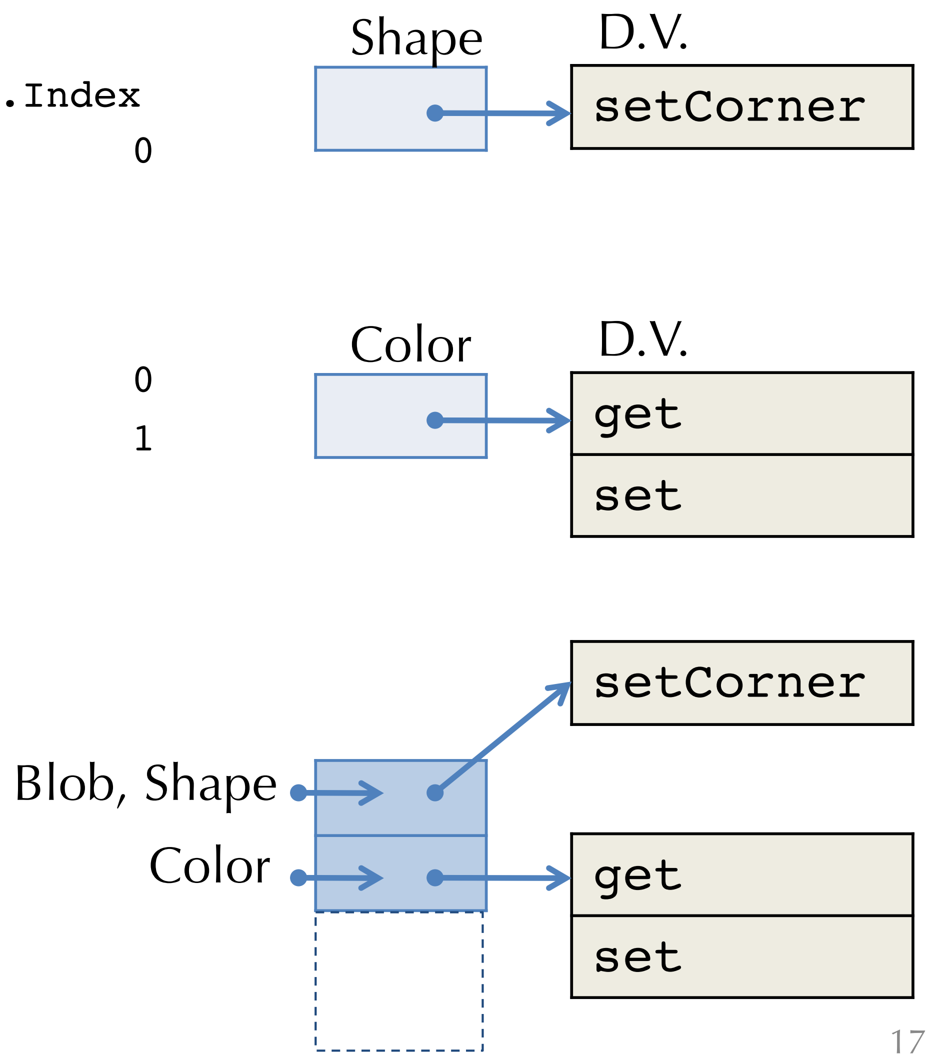
Option 3: Multiple Dispatch Vectors

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

```
interface Shape {                              D.V.Index
  void setCorner(int w, Point p);                  0
}

interface Color {
  float get(int rgb);                              0
  void set(int rgb, float value);                  1
}

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.

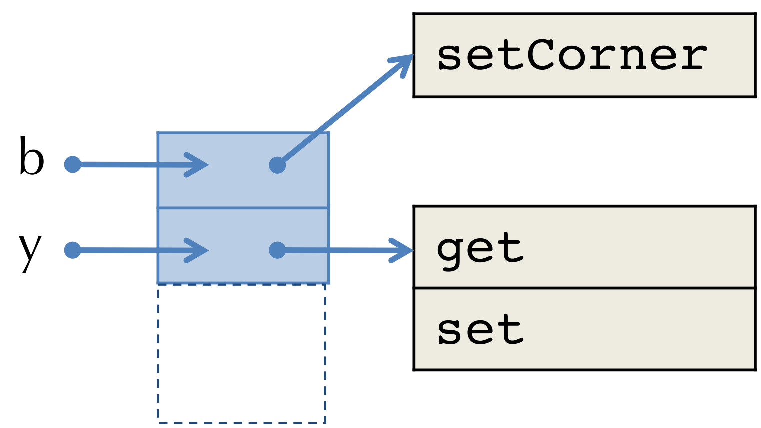
```
Blob b = new Blob();
Color y = b;     // implicit cast!
```

- Compile

  `Color y = b;`

  As

  `Movq ⟦b⟧ + 8 , y`



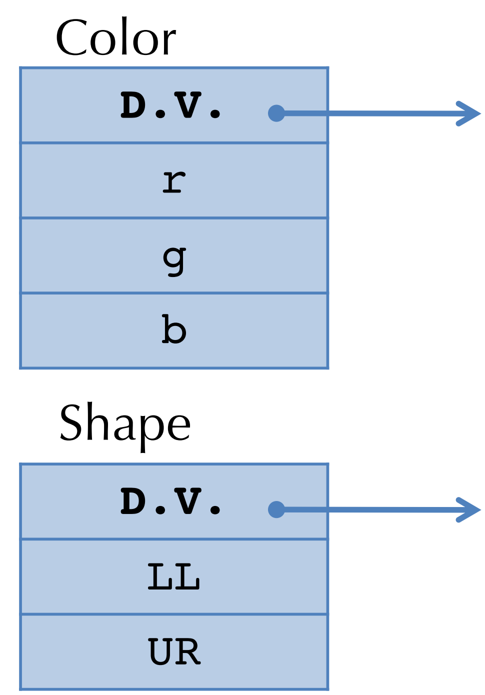
Multiple D.V. Summary

- 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.

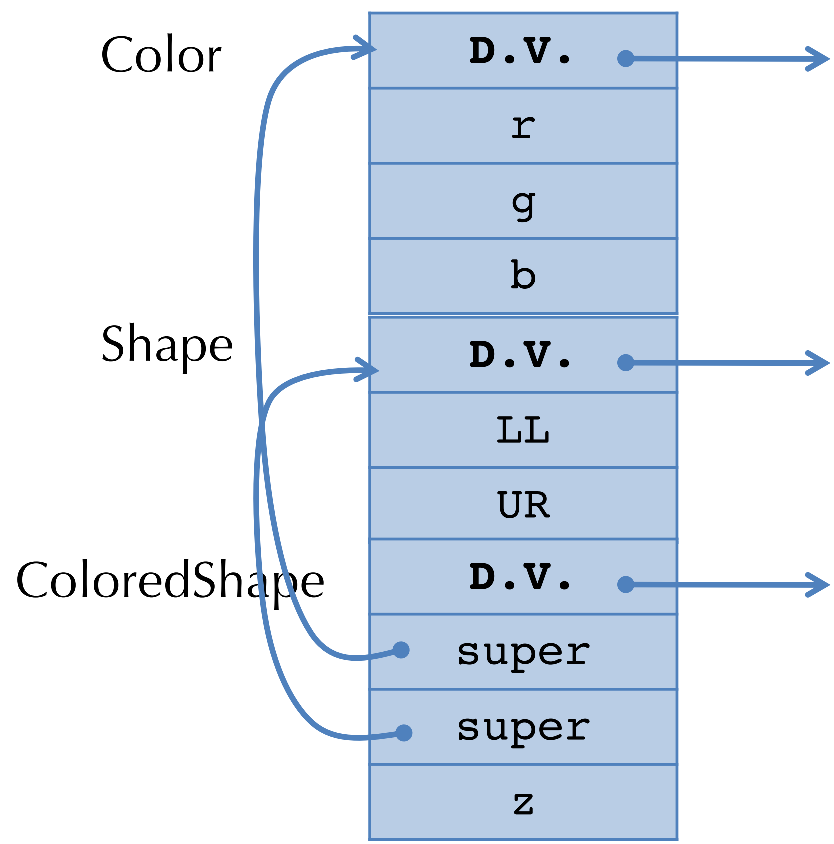
```
class Color {
  float r, g, b; /* offsets: 4,8,12 */
}
class Shape {
  Point LL, UR; /* offsets: 4, 8 */
}
class ColoredShape extends
Color, Shape {
  int z;
}
```



ColoredShape ??

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



Compiling lambda calculus to straight-line code.

Representing evaluation environments at runtime.

CLOSURE CONVERSION REVISITED

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’

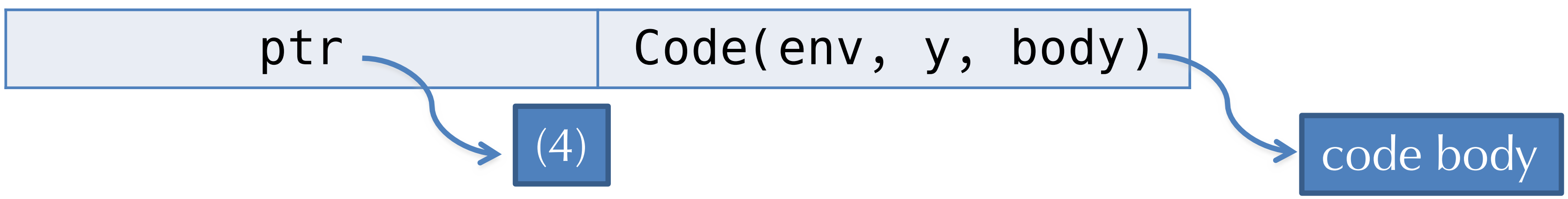
- Reify the substitution:
  - Move substitution from the meta language to the object language by making the data structure & lookup operation explicit
  - The environment-based interpreter is one step in this direction
- Closure Conversion:
  - Eliminates free variables by packaging up the needed environment in the data structure.
- Hoisting:
  - Separates code from data, pulling closed code to the top level.

Example of closure creation

- Recall the "add" function:
  `let add = fun x -> fun y -> x + y`

- Consider the inner function: `fun y -> x + y`

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



- The code pointer takes a pair of parameters: env and y
  - The function code is (essentially):
    `fun (env, y) -> let x = nth env 0 in x + y`

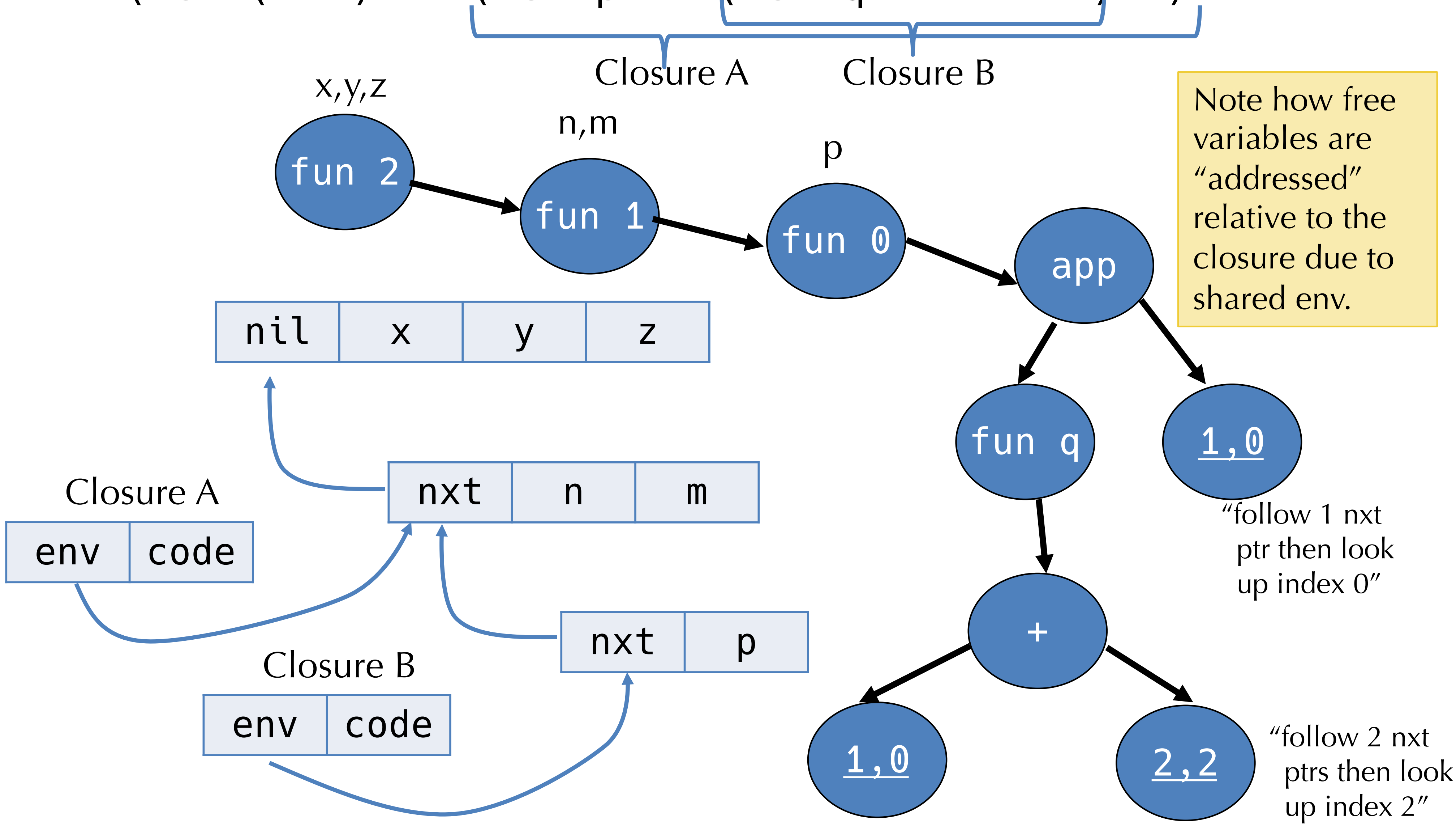
Representing Closures

- 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)
```



Compiling Closures to LLVM IR

- The "types" of the environment data structures are generic tuples
  - The tuples contain a mix of int and closure values
  - We know statically what the tuple-type of the environment should be
  - LLVM IR doesn't have generic types

- Type translations:
  - $[\![-]\!]$ for "intepretation" that retains type information
    $[\![\text{int}]\!] = \text{i64}$
    $[\![(t1, \ldots, tn)]\!] = \{[\![t1]\!], \ldots, [\![tn]\!]\}^*$
    $[\![t1 \rightarrow t2]\!] = [\![t1 \rightarrow t2]\!]_C$
  - $[\![t1 \rightarrow t2]\!]_C = \{\text{i8}^*, ((\text{i8}^*, [\![t1]\!]) \rightarrow [\![t2]\!])^*\}^*$ "Closure Representation"

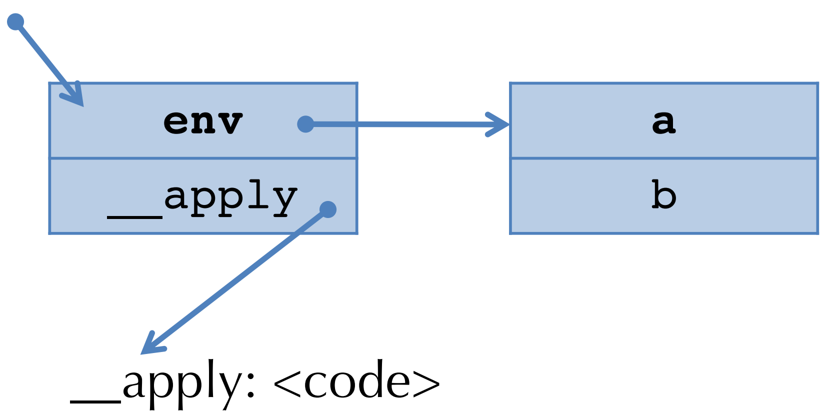
- Rough sketch:
  - Allocation & uses of objects us the "interpretation" translation
  - Anywhere an environment is passed or stored, use i8* and bitcast to/from the translation type.

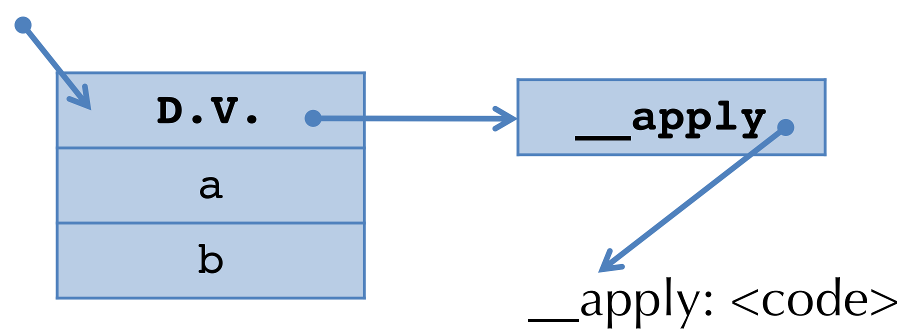
Observe: Closure ≈ Single-method Object

- Free variables ≈ Fields
- Environment pointer ≈ "this" parameter
- Closure for function: ≈ Instance of this class:

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

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





Why optimize?

OPTIMIZATIONS, GENERALLY

Optimizations

- The code generated by our OAT compiler so far is pretty inefficient.
  - Lots of redundant moves.
  - Lots of unnecessary arithmetic instructions.

- Consider this OAT program:

```
int foo(int w) {
  var x = 3 + 5;
  var y = x * w;
  var z = y - 0;
  return z * 4;
}
```

frontend.ml

```
define i64 @foo(i64 %_w1) {
  %_w2 = alloca i64
  %_x5 = alloca i64
  %_y10 = alloca i64
  %_z14 = alloca i64
  store i64 %_w1, i64* %_w2
  %_bop4 = add i64 3, 5
  store i64 %_bop4, i64* %_x5
  %_x7 = load i64, i64* %_x5
  %_w8 = load i64, i64* %_w2
  %_bop9 = mul i64 %_x7, %_w8
  store i64 %_bop9, i64* %_y10
  %_y12 = load i64, i64* %_y10
  %_bop13 = sub i64 %_y12, 0
  store i64 %_bop13, i64* %_z14
  %_z16 = load i64, i64* %_z14
  %_bop17 = mul i64 %_z16, 4
  ret i64 %_bop17
}
```

- opt-example.c, opt-example.oat

Unoptimized vs. Optimized Output

```
define i64 @foo(i64 %_w1) {
  %_w2 = alloca i64
  %_x5 = alloca i64
  %_y10 = alloca i64
  %_z14 = alloca i64
  store i64 %_w1, i64* %_w2
  %_bop4 = add i64 3, 5
  store i64 %_bop4, i64* %_x5
  %_x7 = load i64, i64* %_x5
  %_w8 = load i64, i64* %_w2
  %_bop9 = mul i64 %_x7, %_w8
  store i64 %_bop9, i64* %_y10
  %_y12 = load i64, i64* %_y10
  %_bop13 = sub i64 %_y12, 0
  store i64 %_bop13, i64* %_z14
  %_z16 = load i64, i64* %_z14
  %_bop17 = mul i64 %_z16, 4
  ret i64 %_bop17
}
```

backend.ml

```
        .text
        .globl      _foo
_foo:
        pushq       %rbp
        movq        %rsp, %rbp
        subq        $136, %rsp
        movq        %rdi, %rax
        movq        %rax, -8(%rbp)
        pushq       $0
        movq        %rsp, -16(%rbp)
        pushq       $0
        movq        %rsp, -24(%rbp)
        pushq       $0
        movq        %rsp, -32(%rbp)
        pushq       $0
        movq        %rsp, -40(%rbp)
        movq        -8(%rbp), %rcx
        movq        -16(%rbp), %rax
        movq        %rcx, (%rax)
        movq        $3, %rax
        movq        $5, %rcx
        addq        %rcx, %rax
        movq        %rax, -56(%rbp)
        movq        -56(%rbp), %rcx
        movq        -24(%rbp), %rax
        movq        %rcx, (%rax)
        movq        -24(%rbp), %rax
        movq        (%rax), %rcx
        movq        %rcx, -72(%rbp)
        movq        -16(%rbp), %rax
        movq        (%rax), %rcx
        movq        %rcx, -80(%rbp)
        movq        -72(%rbp), %rax
        movq        -80(%rbp), %rcx
        imulq       %rcx, %rax
        movq        %rax, -88(%rbp)
        movq        -88(%rbp), %rcx
        movq        -32(%rbp), %rax
        movq        %rcx, (%rax)
        movq        -32(%rbp), %rax
        movq        (%rax), %rcx
        movq        %rcx, -104(%rbp)
        movq        -104(%rbp), %rax
        movq        $0, %rcx
        subq        %rcx, %rax
        movq        %rax, -112(%rbp)
        movq        -112(%rbp), %rcx
        movq        -40(%rbp), %rax
        movq        %rcx, (%rax)
        movq        -40(%rbp), %rax
        movq        (%rax), %rcx
        movq        %rcx, -128(%rbp)
        movq        -128(%rbp), %rax
        movq        $4, %rcx
        imulq       %rcx, %rax
        movq        %rax, -136(%rbp)
        movq        -136(%rbp), %rax
        movq        %rbp, %rsp
        popq        %rbp
        retq
```

???

Optimized code:

```
_foo:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rdi, %rax
        shlq    $5, %rax
        popq    %rbp
        retq
```

- Code above generated by `clang -O3`
- Function foo may be inlined by the compiler, so it can be implemented by just one instruction!

Why do we need optimizations?

- To help programmers…
  - They write modular, clean, high-level programs
  - Compiler generates efficient, high-performance assembly

- Programmers don't write optimal code
- High-level languages make avoiding redundant computation inconvenient or impossible
  - e.g. `A[i][j] = A[i][j] + 1`
- Architectural independence
  - Optimal code depends on features not expressed to the programmer
  - Modern architectures *assume* optimization

- Different kinds of optimizations:
  - Time: improve execution speed
  - Space: reduce amount of memory needed
  - Power: lower power consumption (e.g. to extend battery life)

Some caveats

- Optimization are code transformations:
  - They can be applied at any stage of the compiler
  - They must be *safe* – they shouldn't change the meaning of the program.
- In general, optimizations require some program analysis:
  - To determine if the transformation really is safe
  - To determine whether the transformation is cost effective
- This course: most common and valuable performance optimizations
  - See Muchnick (optional text) for ~10 chapters about optimization

When to apply optimization

High level

- AST
- IR

Mid level

- Canonical IR

Low level

- Abstract assembly
- Assembly

- Inlining
- Function specialization
- Constant folding
- Constant propagation
- Value numbering
- Dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength Reduction
- Constant folding & propagation
- Branch prediction / optimization
- Register allocation
- Loop unrolling
- Cache optimization

Where to Optimize?

- Usual goal: improve time performance
- Problem: many optimizations trade space for time
- Example: *Loop unrolling*
  - Idea: rewrite a loop like:
    ```
    for(int i=0; i<100; i=i+1) {
      s = s + a[i];
    }
    ```
  - Into a loop like:
    ```
    for(int i=0; i<99; i=i+2){
      s = s + a[i];
      s = s + a[i+1];
    }
    ```
- Tradeoffs:
  - Increasing code space slows down whole program a tiny bit (extra instructions to manage) but speeds up the loop a lot
  - For frequently executed code with long loops: generally a win
  - Interacts with instruction cache and branch prediction hardware
- Complex optimizations may never pay off!

Writing Fast Programs In Practice

- Pick the right algorithms and data structures.
  - These have a much bigger impact on performance that compiler optimizations.
  - Reduce # of operations
  - Reduce memory accesses
  - Minimize indirection – it breaks working-set coherence
- *Then* turn on compiler optimizations
- Profile to determine program hot spots
- Evaluate whether the algorithm/data structure design works
- …if so: "tweak" the source code until the optimizer does "the right thing" to the machine code

Safety

- Whether an optimization is *safe* depends on the programming language semantics.
  - Languages that provide weaker guarantees to the programmer permit more optimizations but have more ambiguity in their behavior.
  - e.g. In Java tail-call optimization (that turns recursive function calls into loops) is not valid.
  - e.g. In C, loading from initialized memory is undefined, so the compiler can do anything.
- Example: *loop-invariant code motion*
  - Idea: hoist invariant code out of a loop

```
while (b) {
  z = y/x;
  …            // y, x not updated
}
```

```
z = y/x;
while (b) {
  …            // y, x not updated
}
```

- Is this more efficient?
- Is this safe?

A high-level tour of a variety of optimizations.

BASIC OPTIMIZATIONS

Constant Folding

- Idea: If operands are known at compile type, perform the operation statically.

```
int x = (2 + 3) * y   ➔  int x = 5 * y
b  & false            ➔  false
```

- Performed at every stage of optimization…
- Why?
  - Constant expressions can be created by translation or earlier optimizations
- Example: `A[2]` might be compiled to:
  `MEM[MEM[A] + 2 * 4]   ➔  MEM[MEM[A] + 8]`

Constant Folding Conditionals

```
if (true) S                → S
if (false) S               → ;
if (true) S else S'        → S
if (false) S else S'       → S'
while (false) S            → ;

if (2 > 3) S               → ;
```

Algebraic Simplification

- More general form of constant folding
  - Take advantage of mathematically sound simplification rules

- Identities:
  - `a * 1 → a` `a * 0 → 0`
  - `a + 0 → a` `a - 0 → a`
  - `b | false → b` `b & true → b`
- Reassociation & commutativity:
  - `(a + 1) + 2 → a + (1 + 2) → a + 3`
  - `(2 + a) + 4 → (a + 2) + 4 → a + (2 + 4) → a + 6`
- Strength reduction: (replace expensive op with cheaper op)
  - `a * 4 → a << 2`
  - `a * 7 → (a << 3) - a`
  - `a / 32767 → (a >> 15) + (a >> 30)`

- Note 1: must be careful with floating point (due to rounding) and integer arithmetic (due to overflow/underflow)
- Note 2: iteration of these optimizations is useful… how much?

Constant Propagation

- If the value is known to be a constant, replace the use of the variable by the constant
- Value of the variable must be propagated forward from the point of assignment
  - This is a substitution operation

- Example:

```
int x = 5;
int y = x * 2; ➔ int y = 5 * 2; ➔ int y = 10;  ➔
int z = a[y];     int z = a[y];     int z = a[y];  int z = a[10];
```

- To be most effective, constant propagation should be interleaved with constant folding

Copy Propagation

- If one variable is assigned to another, replace uses of the assigned variable with the copied variable.
- Need to know where copies of the variable propagate.
- Interacts with the scoping rules of the language.

- Example:

```
x = y;                          ~~x = y;~~
if (x > 1) {          ➔        if (y > 1) {
  x = x * f(x - 1);              x = y * f(y - 1);
}                               }
```

- Can make the first assignment to x *dead* code (that can be eliminated).

Dead Code Elimination

- If a side-effect free statement can never be observed, it is safe to eliminate the statement.

```
x  = y * y  // x is dead!
…           // x never used  ➔      …
x = z * z                           x = z * z
```

- A variable is *dead* if it is never used after it is defined.
  - Computing such *definition* and *use* information is an important component of compiler

- Dead variables can be created by other optimizations…

Unreachable/Dead Code

- Basic blocks not reachable by any trace leading from the starting basic block are *unreachable* and can be deleted.
  - Performed at the IR or assembly level
  - Improves cache, TLB performance

- Dead code: similar to unreachable blocks.
  - A value might be computed but never subsequently used.
- Code for computing the value can be dropped
- But only if it's *pure*, i.e. it has *no externally visible side effects*
  - Externally visible effects: raising an exception, modifying a global variable, going into an infinite loop, printing to standard output, sending a network packet, launching a rocket
  - Note: Pure functional languages (e.g. Haskell) make reasoning about the safety of optimizations (and code transformations in general) easier!

Inlining

- Replace a call to a function with the body of the function itself with arguments rewritten to be local variables:
- Example in OAT code:

```
int g(int x) { return x + pow(x); }
int pow(int a) { int b = 1; int n = 0;
   while (n < a) {b = 2 * b}; return b; }
```

➔

```
int g(int x) { int a = x; int b = 1; int n = 0;
   while (n < a) {b = 2 * b}; tmp = b; return x + tmp;
}
```

- May need to rename variable names to avoid *name capture*
  - Example of what can go wrong?
- Best done at the AST or relatively high-level IR.
- When is it profitable?
  - Eliminates the stack manipulation, jump, etc.
  - Can increase code size.
  - Enables further optimizations

Code Specialization

- Idea: create specialized versions of a function that is called from different places with different arguments.
- Example: specialize function `f` in:

```
class A implements I { int m() {…} }
class B implements I { int m() {…} }
int f(I x) { x.m(); }       // don't know which m
A a = new A(); f(a);        // know it's A.m
B b = new B(); f(b);        // know it's B.m
```

- `f_A` would have code specialized to dispatch to `A.m`
- `f_B` would have code specialized to dispatch to `B.m`
- You can also inline methods when the run-time type is known statically
  - Often just one class implements a method.

Common Subexpression Elimination

- In some sense it's the opposite of inlining: fold redundant computations together
- Example:

`a[i] = a[i] + 1` compiles to:

```
[a + i*4] = [a + i*4] + 1
```

Common subexpression elimination removes the redundant add and multiply:

```
t = a + i*4; [t] = [t] + 1
```

- For safety, you must be sure that the shared expression always has the same value in both places!

Unsafe Common Subexpression Elimination

- Example: consider this OAT function:

```
unit f(int[] a, int[] b, int[] c) {
  int j = …; int i = …; int k = …;
  b[j] = a[i] + 1;
  c[k] = a[i];
  return;
}
```

- The optimization that shares the expression `a[i]` is unsafe… why?

```
unit f(int[] a, int[] b, int[] c) {
  int j = …; int i = …; int k = …;
  t = a[i];
  b[j] = t + 1;
  c[k] = t;
  return;
}
```