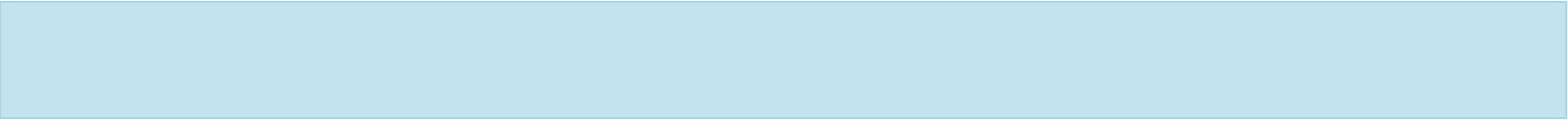


Lecture 18

CIS 341: COMPILERS

Announcements

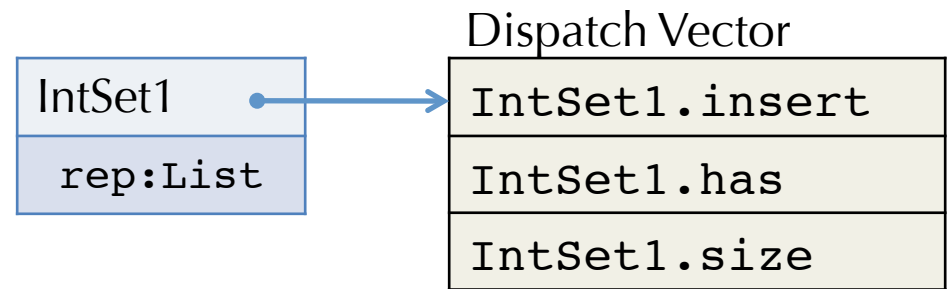
- HW5: Full OAT: Objects & Typechecking
 - Implement (parts of) the typechecker and compiler for an OO-language
- DUE: Monday April 6th



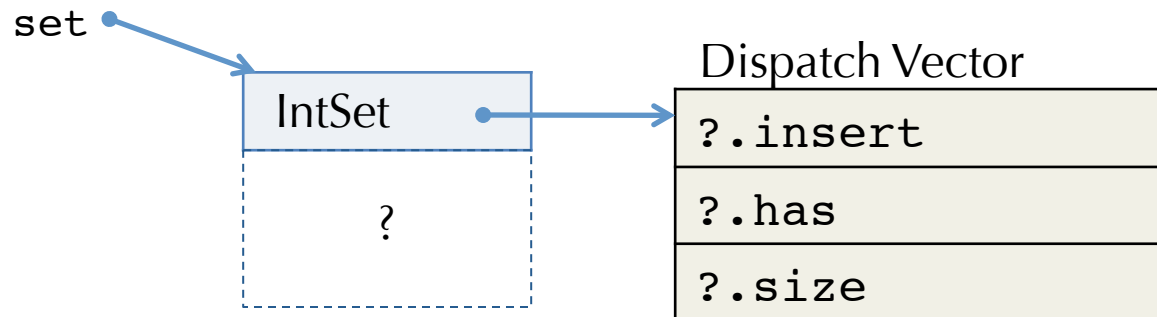
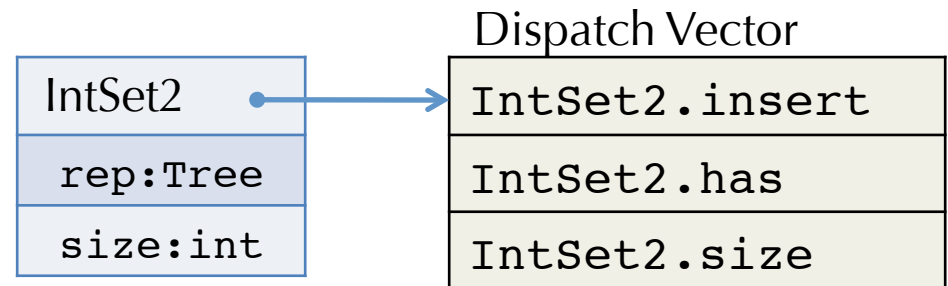
COMPILING CLASSES AND OBJECTS

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.

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

Index

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

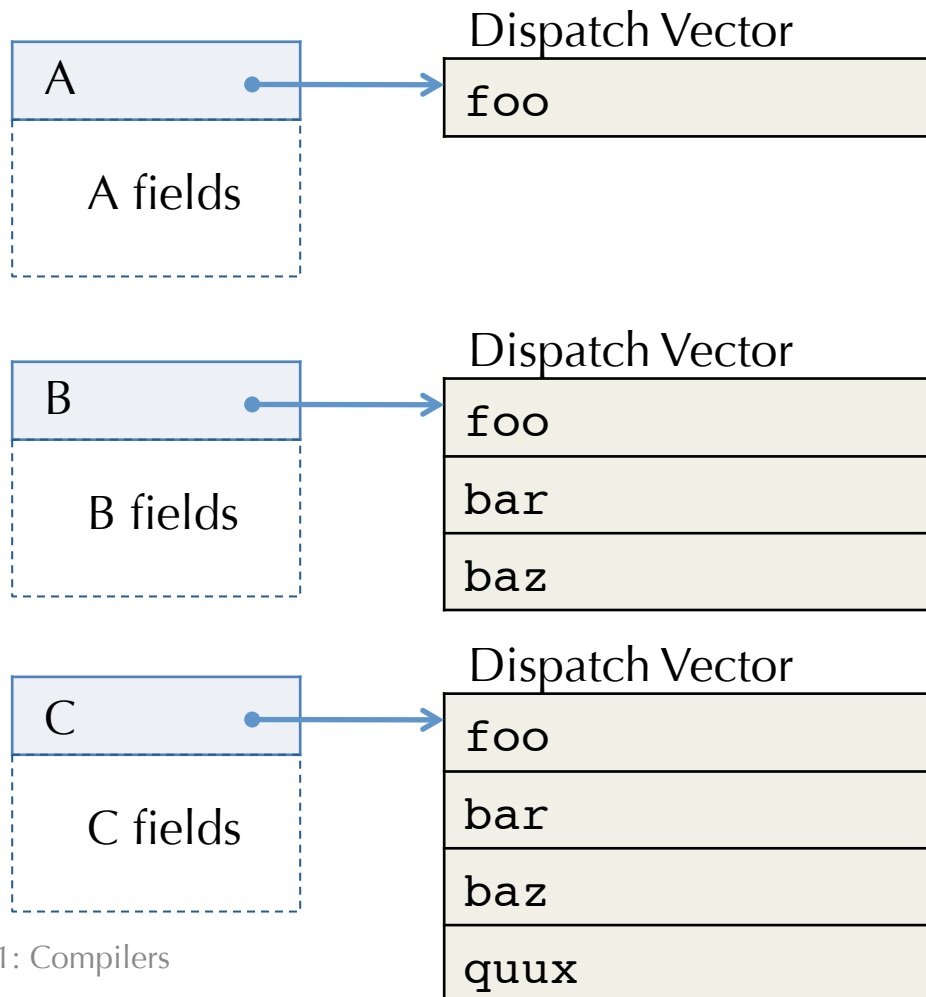
1

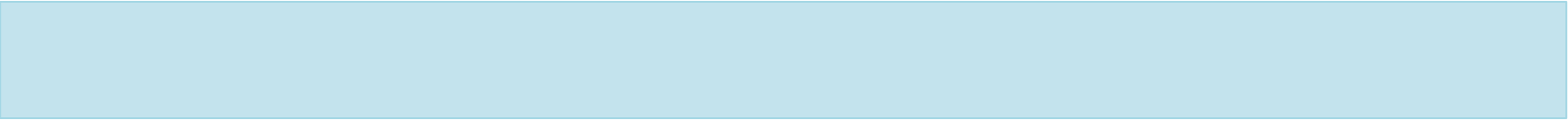
2

3

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)





MULTIPLE INHERITANCE

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

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

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

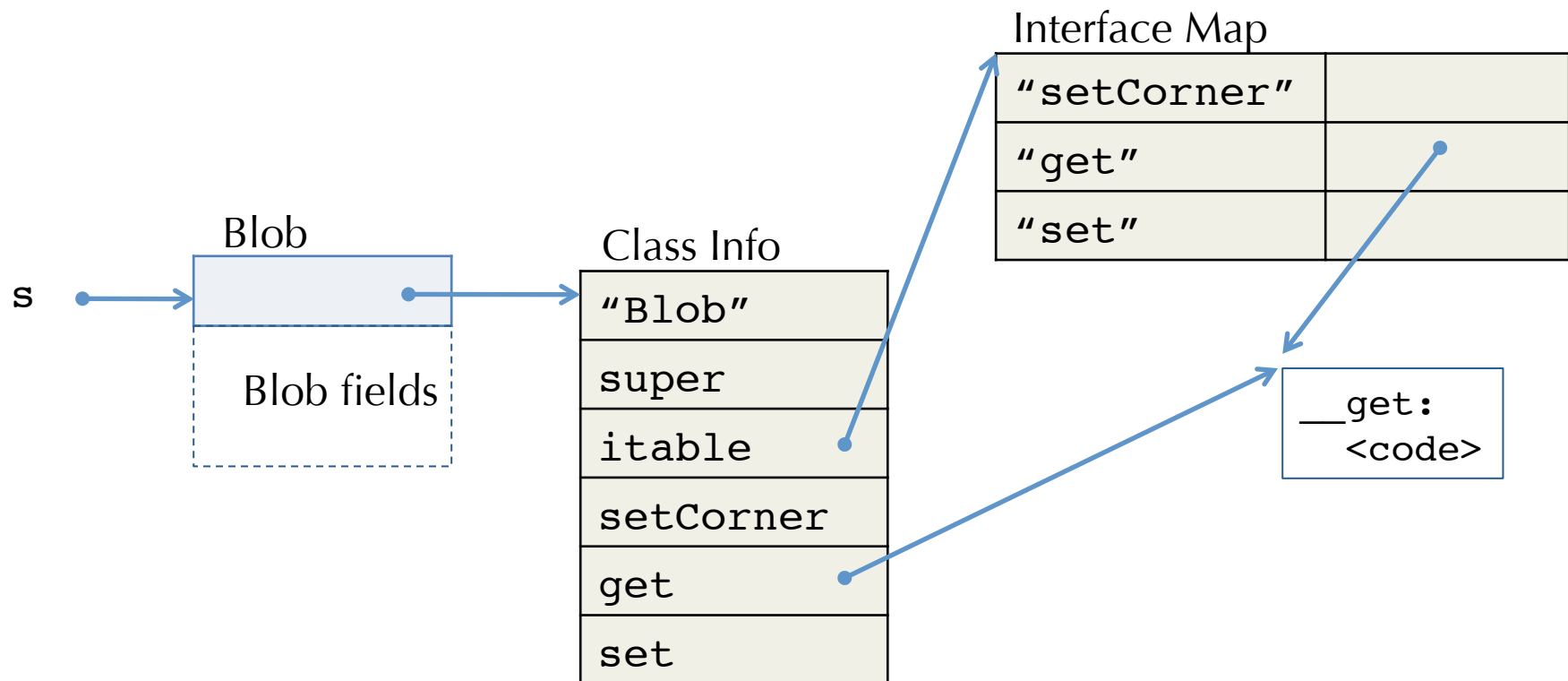
class Blob implements Shape, Color {	
void setCorner(int w, Point p) {...}	0?
float get(int rgb) {...}	0?
void set(int rgb, float value) {...}	1?
}	

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...

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 in quotes are not strings; in practice they are some kind of unique identifier.



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: `Shape s = new Blob(); s.get();`

Call site 434 

Table in data seg.

```
cacheClass434:
    "Blob"
cacheCode434:
```
- Compiler knows that `s` is a `Shape`
 - Suppose `%rax` holds object pointer

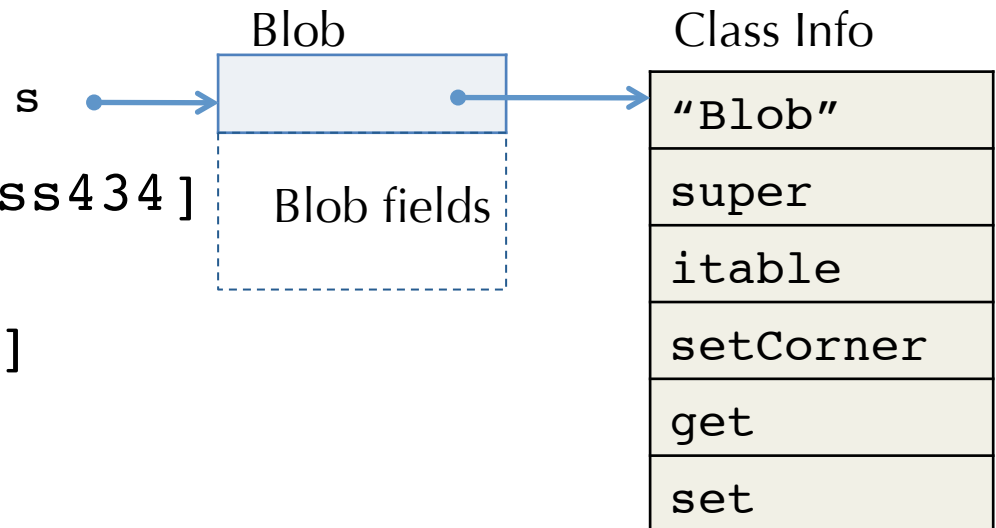
- Cached interface dispatch:

```
// set up parameters
```

```
movq [%rax], tmp
cmpq tmp, [cacheClass434]
Jnz __miss434
callq [cacheCode434]
```

miss434:

```
// do the slow search
```



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

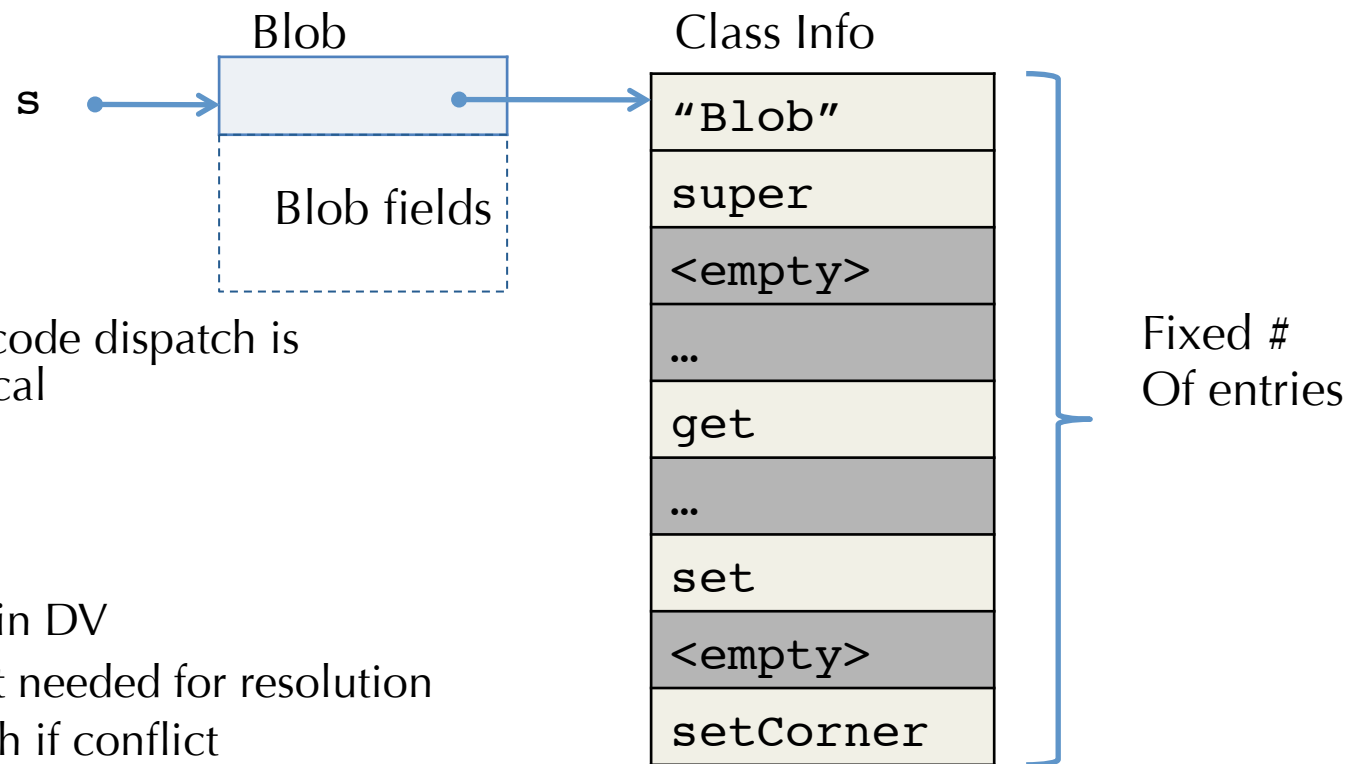
```
interface Shape {                                D.V.Index
    void setCorner(int w, Point p);             hash("setCorner") = 11
}
```

```
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)



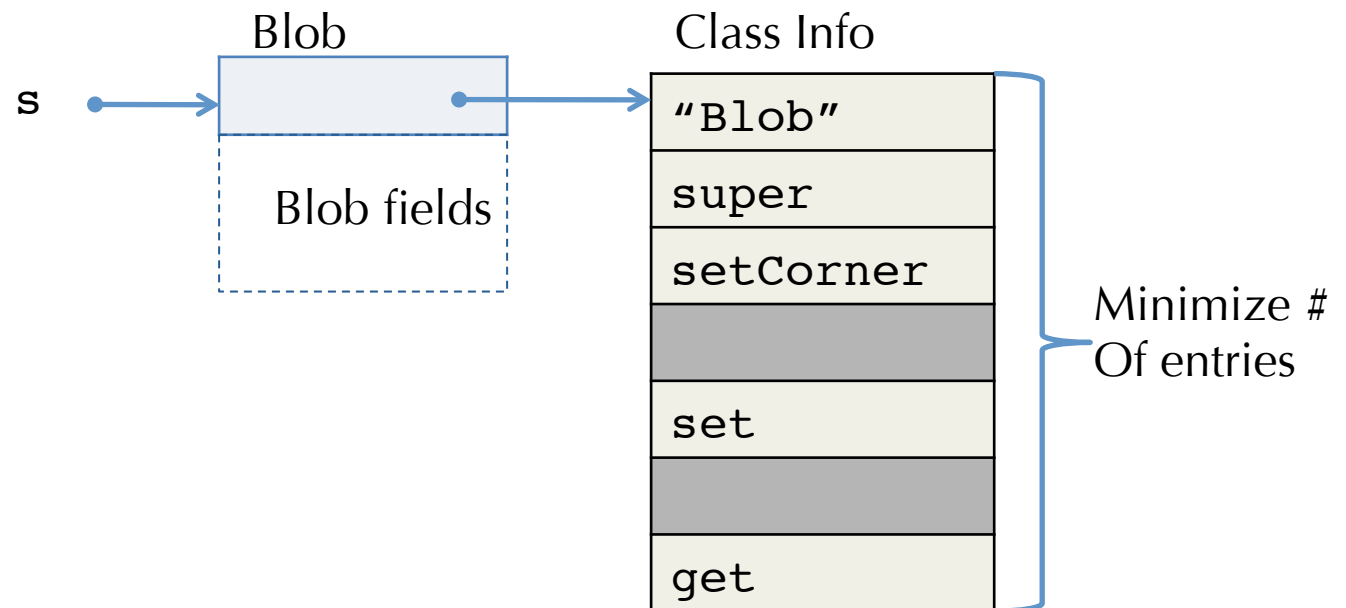
- Advantage:
 - Simple, basic code dispatch is (almost) identical
 - Reasonably efficient
- Disadvantage:
 - Wasted space in DV
 - Extra argument needed for resolution
 - Slower dispatch if conflict

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



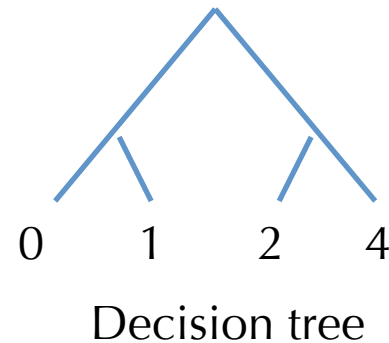
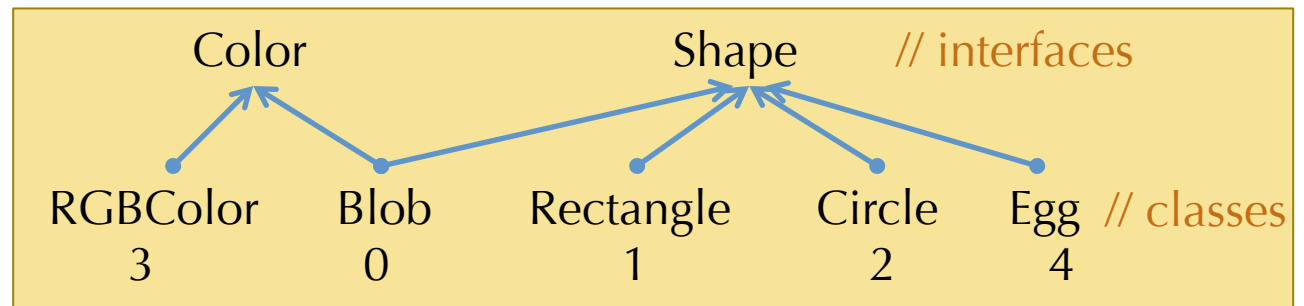
Option 2 variant 2: Binary Search Trees

- 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.

```
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
```



Search Tree Tradeoffs

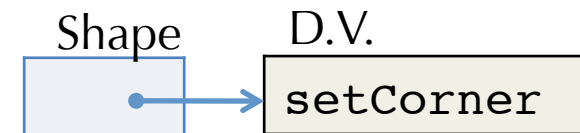
- Binary decision trees work well if the distribution of classes that may appear at a call site is skewed.
 - Branch prediction hardware eliminates the branch stall of ~10 cycles (on X86)
- Can use profiling to find the common paths for each call site individually
 - Put the common case at the top of 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.

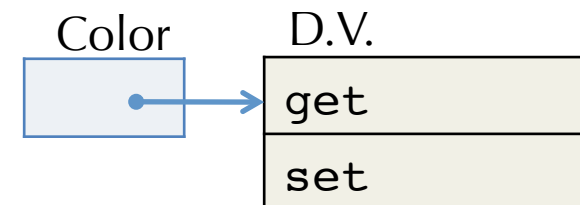
```
interface Shape {  
    void setCorner(int w, Point p);  
}
```

D.V. Index
0

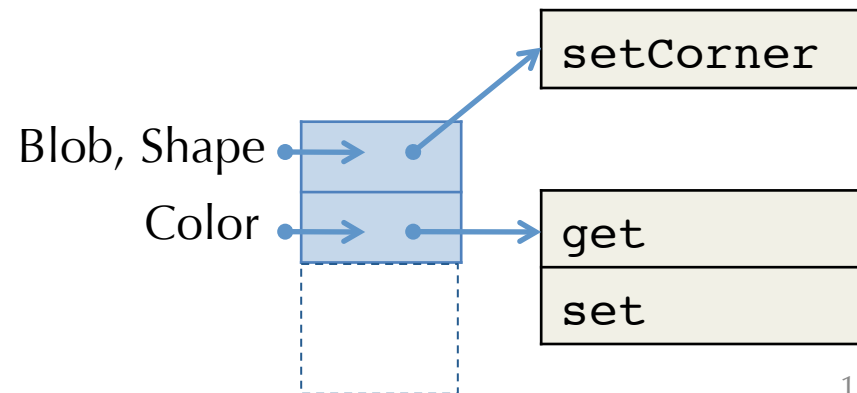


```
interface Color {  
    float get(int rgb);  
    void set(int rgb, float value);  
}
```

0
1



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

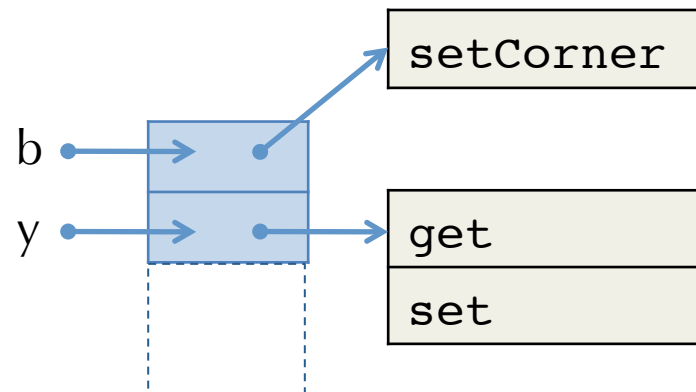


Multiple Dispatch Vectors

- 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



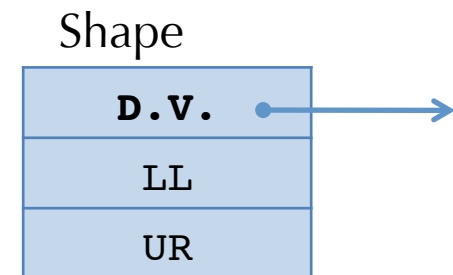
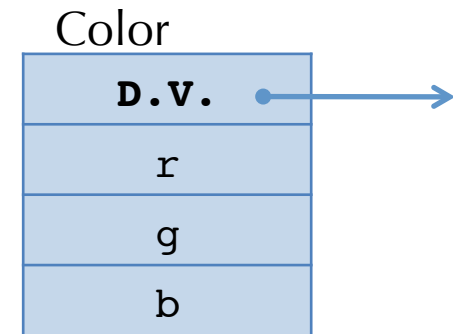
Multiple D.V. Summary

- 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.

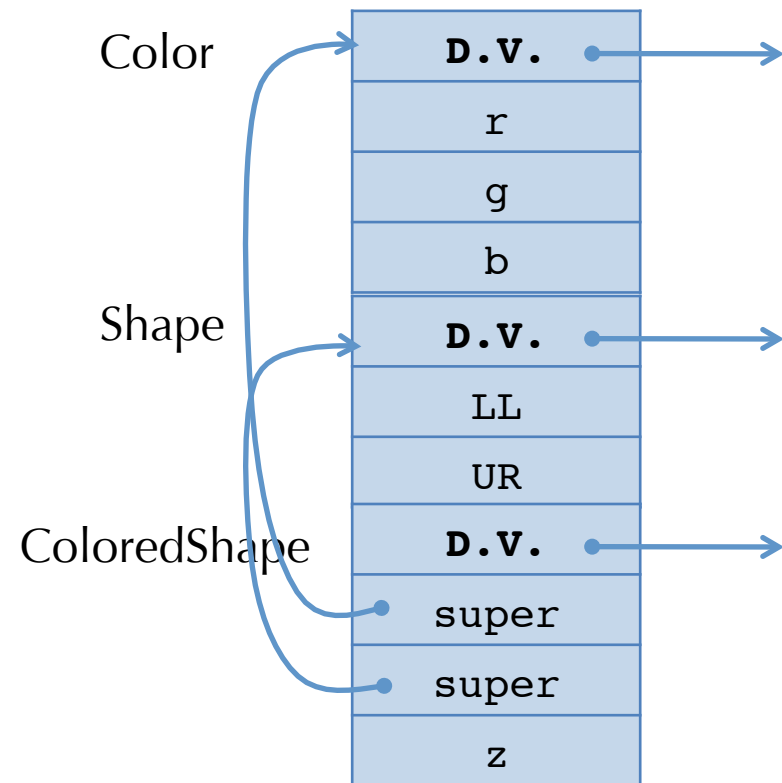
```
class Color {  
    float r, g, b; /* offsets: 4,8,12 */  
}  
class Shape {  
    Point LL, UR; /* offsets: 4, 8 */  
}  
class ColoredShape extends  
Color, Shape {  
    int z;  
}
```



ColoredShape ??

C++ approach:

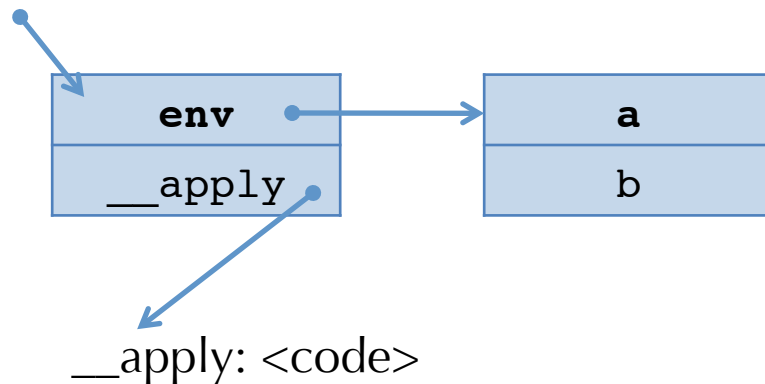
- Add pointers to the superclass fields
 - Need to have multiple dispatch vectors anyway (to deal with methods)
- Extra indirection needed to access superclass fields
- Used even if there is a single superclass
 - Uniformity



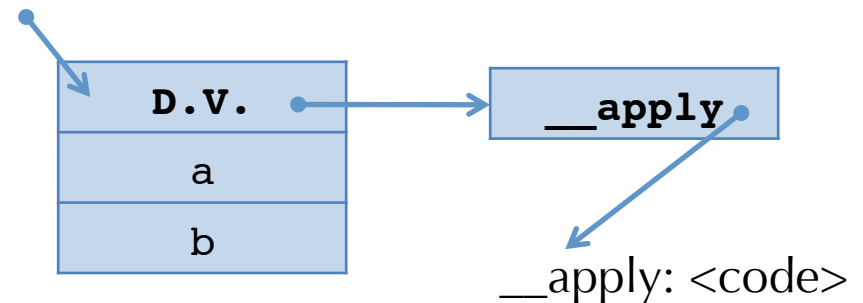
Observe: Closure $\approx$ Single-method Object

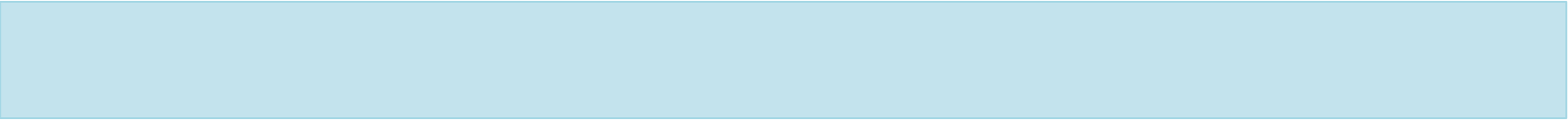
- Free variables $\approx$ Fields
- Environment pointer $\approx$ “this” parameter
- Closure for function: $\approx$ Instance of this class:

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



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





A high-level tour of a variety of optimizations.

OPTIMIZATIONS

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 / C program:

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

- See opt.c, opt-oat.oat

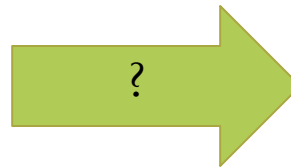
Unoptimized vs. Optimized Output

```
.globl _foo
_foo:
    pushl %ebp
    movl %esp, %ebp
    subl $64, %esp

    __fresh2:
        leal -64(%ebp), %eax
        movl %eax, -48(%ebp)
        movl 8(%ebp), %eax
        movl %eax, %ecx
        movl -48(%ebp), %eax
        movl %ecx, (%eax)
        movl $3, %eax
        movl %eax, -44(%ebp)
        movl $5, %eax
        movl %eax, %ecx
        addl %ecx, -44(%ebp)
        leal -60(%ebp), %eax
        movl %eax, -40(%ebp)
        movl -44(%ebp), %eax
        movl %eax, %ecx
        movl -40(%ebp), %eax
        movl %ecx, (%eax)
        movl (%eax), %ecx
        movl %ecx, -36(%ebp)
        movl -48(%ebp), %eax
        movl (%eax), %ecx
        movl %ecx, -32(%ebp)
        movl -36(%ebp), %eax
        movl %eax, -28(%ebp)
        movl -32(%ebp), %eax
        movl %eax, %ecx
        movl -28(%ebp), %eax
        imull %ecx, %eax
        movl %eax, -28(%ebp)
        leal -56(%ebp), %eax
        movl %eax, -24(%ebp)
        movl -28(%ebp), %eax
        movl %eax, %ecx
        movl -24(%ebp), %eax
        movl %ecx, (%eax)
        movl -24(%ebp), %eax
        movl (%eax), %ecx
        movl %ecx, -20(%ebp)
        movl -20(%ebp), %eax
        movl %eax, -16(%ebp)
        movl $0, %eax
        movl %eax, %ecx
        subl %ecx, -16(%ebp)
        leal -52(%ebp), %eax
        movl %eax, -12(%ebp)
        movl -16(%ebp), %eax
        movl %eax, %ecx
        movl -12(%ebp), %eax
        movl %ecx, (%eax)
        movl -12(%ebp), %eax
        movl (%eax), %ecx
        movl %ecx, -8(%ebp)
        movl -8(%ebp), %eax
        movl %eax, -4(%ebp)
        movl $4, %eax
        movl %eax, %ecx
        movl -4(%ebp), %eax
        imull %ecx, %eax
        movl %eax, -4(%ebp)
        movl -4(%ebp), %eax
        movl %ebp, %esp
        popl %ebp
        ret
```

Hand optimized code:

```
_foo:
    shlq    $5, %rdi
    movq    %rdi, %rax
    ret
```



- Function foo may be inlined by the compiler, so it can be implemented by just one instruction!

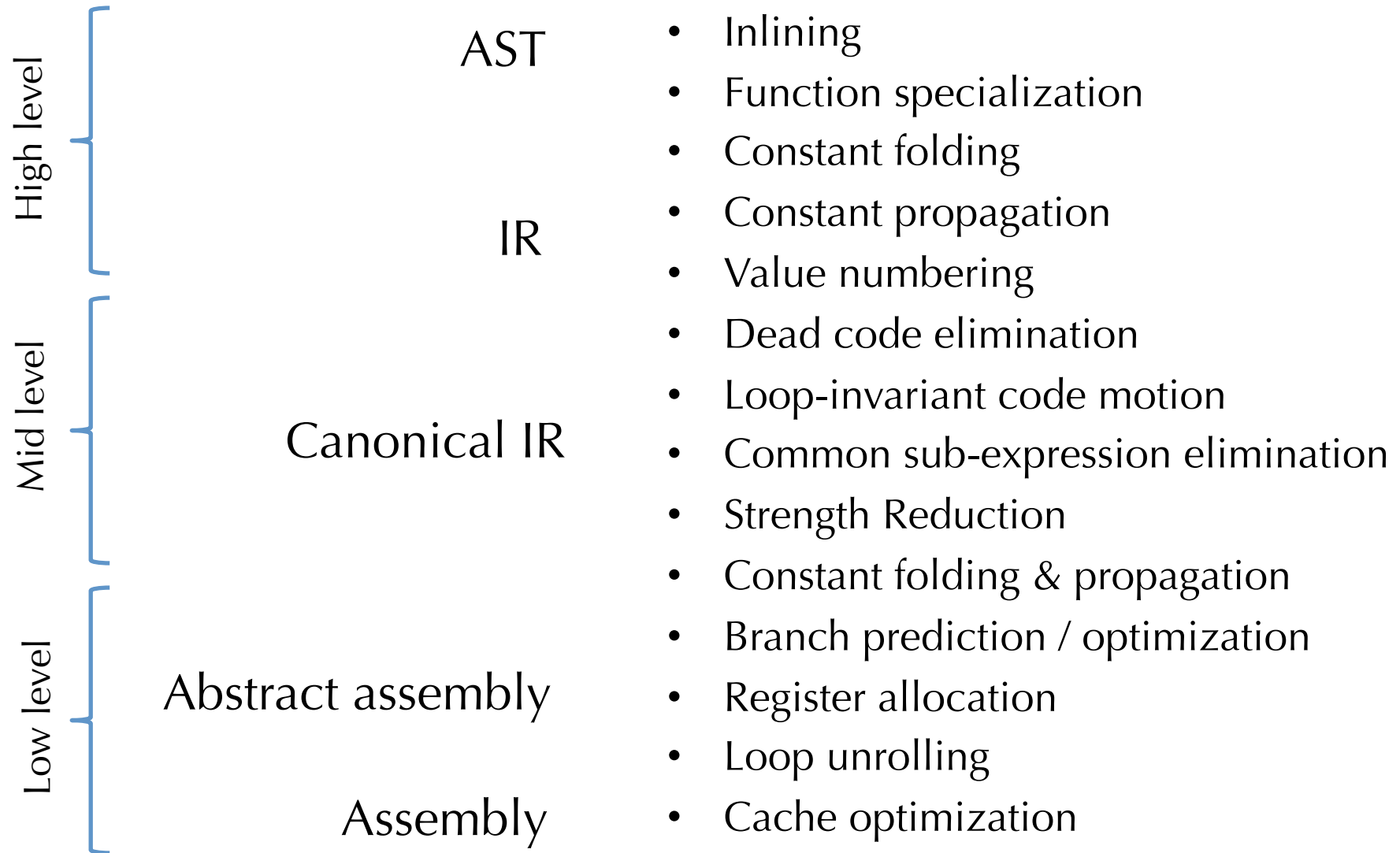
Why do we need optimizations?

- 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 can'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



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 codes space slows down whole program a tiny bit but speeds up the loop
 - 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 than 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?

Constant Folding

- Idea: If operands are known at compile time, 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:
 - $a * 1 \rightarrow a$ $a * 0 \rightarrow 0$
 - $a + 0 \rightarrow a$ $a - 0 \rightarrow a$
 - $b \mid \text{false} \rightarrow b$ $b \& \text{true} \rightarrow b$
- Reassociation & commutativity:
 - $(a + 1) + 2 \rightarrow a + (1 + 2) \rightarrow a + 3$
 - $(2 + a) + 4 \rightarrow (a + 2) + 4 \rightarrow a + (2 + 4) \rightarrow a + 6$
- Strength reduction: (replace expensive op with cheaper op)
 - $a * 4 \rightarrow a \ll 2$
 - $a * 7 \rightarrow (a \ll 3) - a$
 - $a / 32767 \rightarrow (a \gg 15) + (a \gg 30)$
- Note 1: must be careful with floating point (due to rounding)
- 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;  
if (x > 1) {  
    x = x * f(x - 1);  
}  
→  
x = y;  
if (y > 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 canonical 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.