Lecture 17

COMPILER DESIGN
• HW5: OAT v. 2.0
  – Records, function pointers, type checking, array-bounds checks, etc.
  – **Due:** Thursday, November 28\textsuperscript{th} at 23:59

• Final Exam
  – Scheduled for Friday, January 31\textsuperscript{st}, 9-11 AM
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
  ```
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) {...}
}
General Approaches

• Can’t directly identify methods by position anymore

• Option 1: 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

• Option 2: 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 3: Give up separate compilation
  – Use “sparse” dispatch vectors, or binary decision trees
  – Must know then entire class hierarchy

• Note: many variations on these themes
  – Different Java compilers pick different approaches to options 2 & 3
Option 1: 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 {
    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) {...}
}
```

Zhendong Su    Compiler Design
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
```

Zhendong Su    Compiler Design
Multiple D.V. Summary

- Benefit: Efficient dispatch, similar cost as for single 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;
}
```
vtable for C++ Multiple Inheritance

class A {
    public:
        int x;
        virtual void f();
};
class B {
    public:
        int y;
        virtual void g();
        virtual void f();
};
class C: public A, public B {
    public:
        int z;
        virtual void f();
};

C *pc = new C;
B *pb = pc;
A *pa = pc;

Three pointers to the same object, but different static types
Objects & Classes

- Offset d in vtbl is used in call to pb->f, since C::f may refer to A data that is above the pointer pb.
- Call to pc->g can proceed through C-as-B vtbl.
Multiple Inheritance “Diamond”

- Is interface or implementation inherited twice?
- What if definitions conflict?
Diamond Inheritance in C++

- Standard base classes
  - D members appear twice in C

- Virtual base classes
  
  class A : public virtual D { ... }
  
  - Avoid duplication of base class members
  - Require additional pointers so that D part of A, B parts of object can be shared

Multiple inheritance is complicated in C++ because of its desire to maintain efficient lookup
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) {...}
}
Option 2: 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 identifiers
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

• 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 2 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 { 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 3 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 3 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
```

Decision tree

---

Zhendong Su  Compiler Design
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)
Observe: Closure ≈ 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
    }
}
```

---

Zhendong Su    Compiler Design
CLASSES & OBJECTS IN LLVM
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)

class A {
    A (int x) { super(); int x = x; }
    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

%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 }
Method Arguments

• 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 as sending “messages” to “receivers”

A method in a class...

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

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

```java
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`
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_ptr}

Use \text{Getelementptr} to extract the \text{vtable} pointer from \text{obj_ptr}

Load the \text{vtable} pointer

Use \text{Getelementptr} to extract the address of the function pointer from the \text{vtable}

- using the information about C in H

Load the function pointer

Call through the function pointer, passing \text{\textquote{obj_ptr}} for this:

\text{call (cmp_typ t) m(obj_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 \text{“formal” type}
X86 Code For Dynamic Dispatch

• Suppose \( b : B \)
• What code for \( b.bar(3) \)?
  – \( bar \) has index 1
  – Offset = \( 8 \times 1 \)

\[
\begin{align*}
\text{movq } & [b], \%rax \\
\text{movq } & [%rax], \%rbx \\
\text{movq } & [rbx+8], \%rcx & \quad \text{// D.V. + offset} \\
\text{movq } & \%rax, \%rdi & \quad \text{// “this” pointer} \\
\text{movq } & 3, \%rsi & \quad \text{// Method argument} \\
\text{call } & \%ecx & \quad \text{// Indirect call}
\end{align*}
\]
• 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
Compiling Static Methods

• Java supports 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:

  ```
  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
A high-level tour of a variety of 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 program:

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

• See opt.c, opt-oat.oat
Hand optimized code:

```
_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 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

- 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

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

```plaintext
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 type, perform the operation statically.

\[
\text{int } x = (2 + 3) \times y \rightarrow \text{int } x = 5 \times y \\
\text{b & false} \rightarrow \text{false}
\]

• Performed at every stage of optimization…
• Why?
  – Constant expressions can be created by translation or earlier optimizations
• Example: \text{A[2]} might be compiled to:
  \[
  \text{MEM[MEM[A] + 2 \times 4]} \rightarrow \text{MEM[MEM[A] + 8]}
  \]
Constant Folding Conditionals

if (true) S  \rightarrow S
if (false) S  \rightarrow ;
if (true) S else S’  \rightarrow S
if (false) S else S’  \rightarrow S’
while (false) S  \rightarrow ;
if (2 > 3) S  \rightarrow ;
Algebraic Simplification

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

• Identities:
  – $a * 1 \rightarrow a \quad a * 0 \rightarrow 0$
  – $a + 0 \rightarrow a \quad a - 0 \rightarrow a$
  – $b | \ false \rightarrow b \quad b & \ true \rightarrow b$

• 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)
  – $a * 4 \rightarrow a << 2$
  – $a * 7 \rightarrow (a << 3) - a$
  – $a / 32767 \rightarrow (a >> 15) + (a >> 30)$

• 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?

Zhendong Su    Compiler Design
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;  \Rightarrow \text{int y = 5 * 2;  }  \Rightarrow \text{int y = 10;  }  \Rightarrow 
  int z = a[y]; \quad \text{int z = a[y]; \quad int z = a[y]; \quad 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:

\[
x = y; \quad \Rightarrow \quad x = y;
\]

\[
\text{if} \ (x > 1) \ {\{} \quad \Rightarrow \quad \text{if} \ (y > 1) \ {\{} \\
\quad x = x * f(x - 1); \quad \Rightarrow \quad x = y * f(y - 1);
\text{\}} \quad \quad \quad \text{\}}
\]

• 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.

\[
x = y \times y \quad // \ x \text{ is dead!}
\]

\[
\ldots \quad // \ x \text{ never used} \quad \rightarrow \quad \ldots
\]

\[
x = z \times z
\]

\[
x = z \times z
\]

• 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:
  ```c
  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; }
  }
  ```

Fallback

- 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

Zhendong Su  Compiler Design
Code Specialization

• Idea: create specialized versions of a function that is called from different places with different arguments.
• Example: specialize function \( f \) in:

```java
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:

```c
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 following optimization that shares the expression `a[i]` is unsafe... why?

```c
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;
}
```