Implementation of High Level Languages

Advanced Compiler Techniques

2005

Erik Stenman

EPFL

Overview

- In this second part of the course we will talk about how to implement:
  - Objects and inheritance.
  - FPLs: higher order functions, laziness.
  - Concurrency: processes, message passing.
  - Automatic memory management. (GC)
  - Virtual Machines.
  - Just in time compilation.

Implementation of High Level Languages

- We will look at some simple ways to implement concepts in HLL.
- We will look at some more complex and more efficient implementations of these concepts.
- We will also look at some general optimization techniques that can be used with great advantage in HLL.

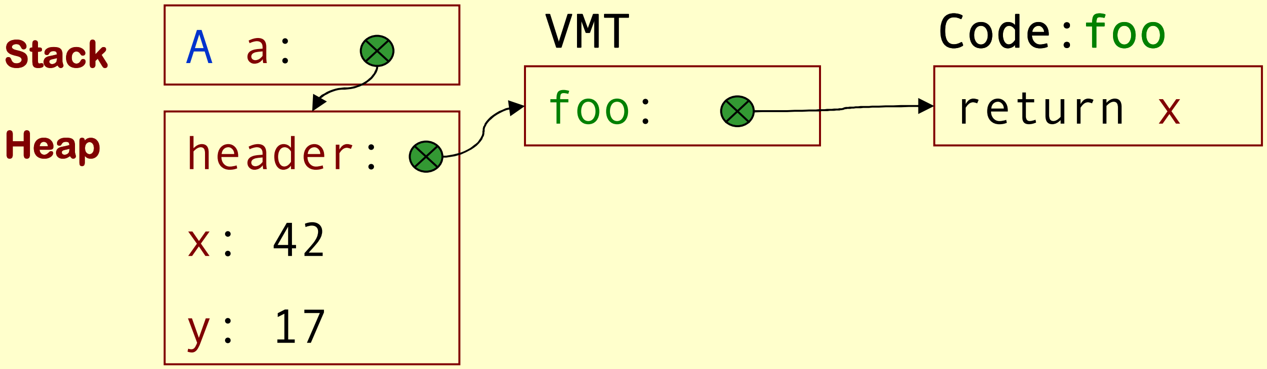
Implementation of Object Oriented Languages

- In class based OO languages each object belongs to a class that defines the fields, methods, and the type of the object.

```
class A {
  int x = 42;
  int y = 17;

  int foo() {
     return x;
  }
}
```

```
A a = new A;
a.foo();
```



Stack

A a:

Heap

header:

x: 42

y: 17

VMT

foo:

Code:foo

return x

Implementation of Object Oriented Languages

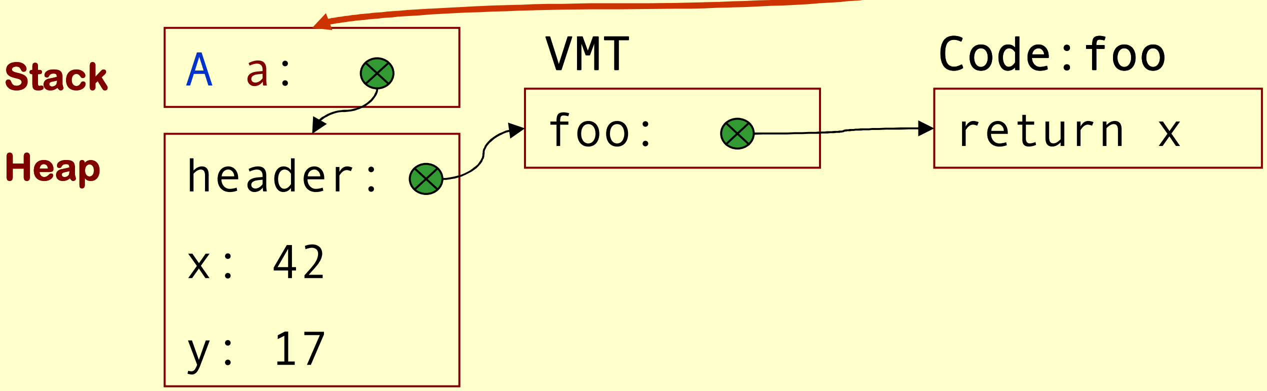
- In class based OO languages each object belongs to a class that defines the fields, methods, and the type of the object.

```
class A {
  int x = 42;
  int y = 17;

  int foo() {
     return x;
  }
}
```

```
A a = new A;
a.foo();
```

Reference to object: many/object.



Implementation of Object Oriented Languages

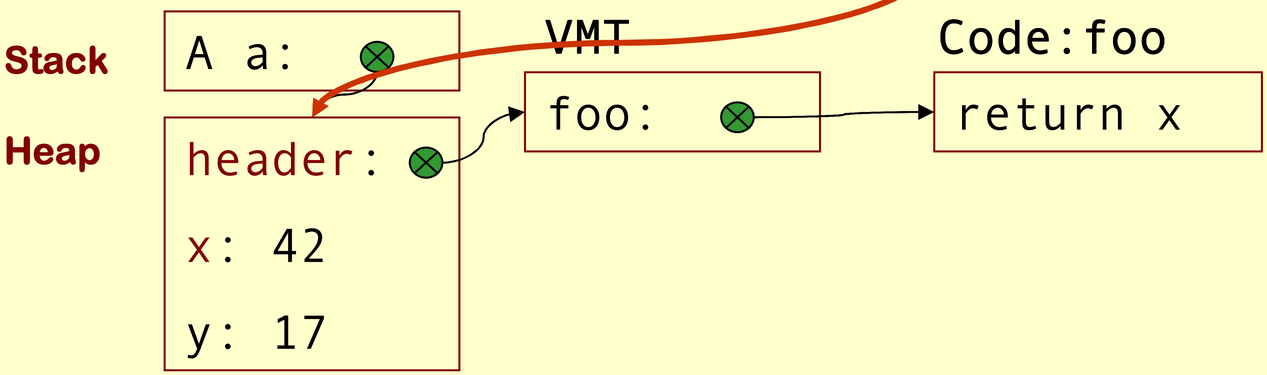
- In class based OO languages each object belongs to a class that defines the fields, methods, and the type of the object.

```
class A {
   int x = 42;
   int y = 17;

   int foo() {
      return x;
   }
}
```

```
A a = new A;
a.foo();
```

Representation of object: 1/object.



Implementation of Object Oriented Languages

- In class based OO languages each object belongs to a class that defines the fields, methods, and the type of the object.

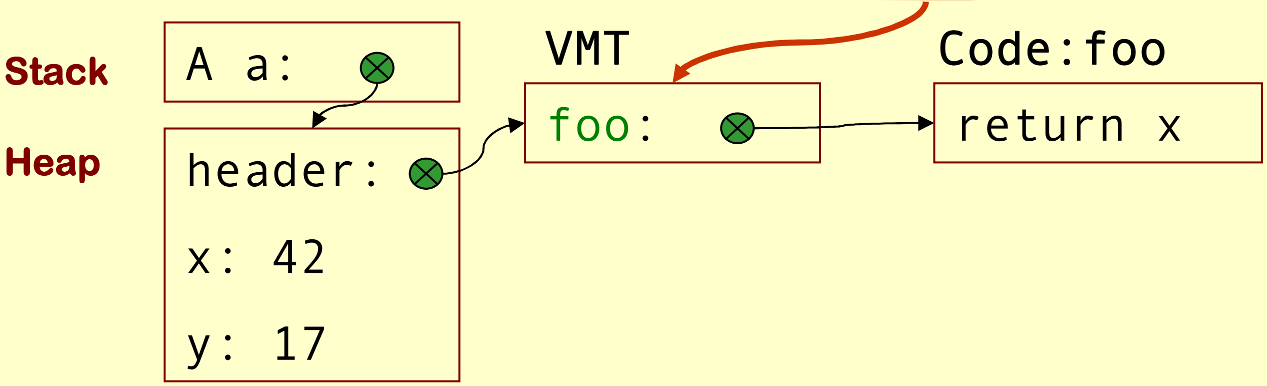
```
class A {
  int x = 42;
  int y = 17;

  int foo() {
     return x;
  }
}
```

```
A a = new A;
a.foo();
```

Virtual Method Table:

1/class.



Implementation of Object Oriented Languages

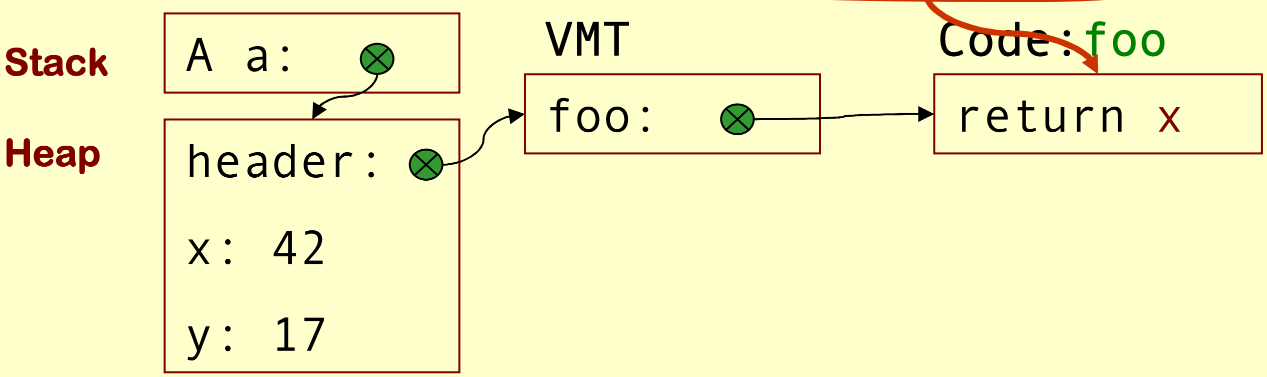
- In class based OO languages each object belongs to a class that defines the fields, methods, and the type of the object.

```
class A {
  int x = 42;
  int y = 17;

  int foo() {
     return x;
  }
}
```

```
A a = new A;
a.foo();
```

Code for functions (foo): max 1/class.



Stack

A a:

Heap

header:

x: 42

y: 17

VMT

foo:

Code:foo

return x

Implementation of Object Oriented Languages

- Object Oriented languages support inheritance.
- Inheritance complicates the answer to some questions:
  - Where is the value of a field stored?
  - Where is the code for a certain method?
  - What type will a value have at runtime?

Single Inheritance: Fields

- With single inheritance we can order the fields in such a way that all fields of a class are stored after fields of the superclass.
- This way we know at compile time the offset of each field.

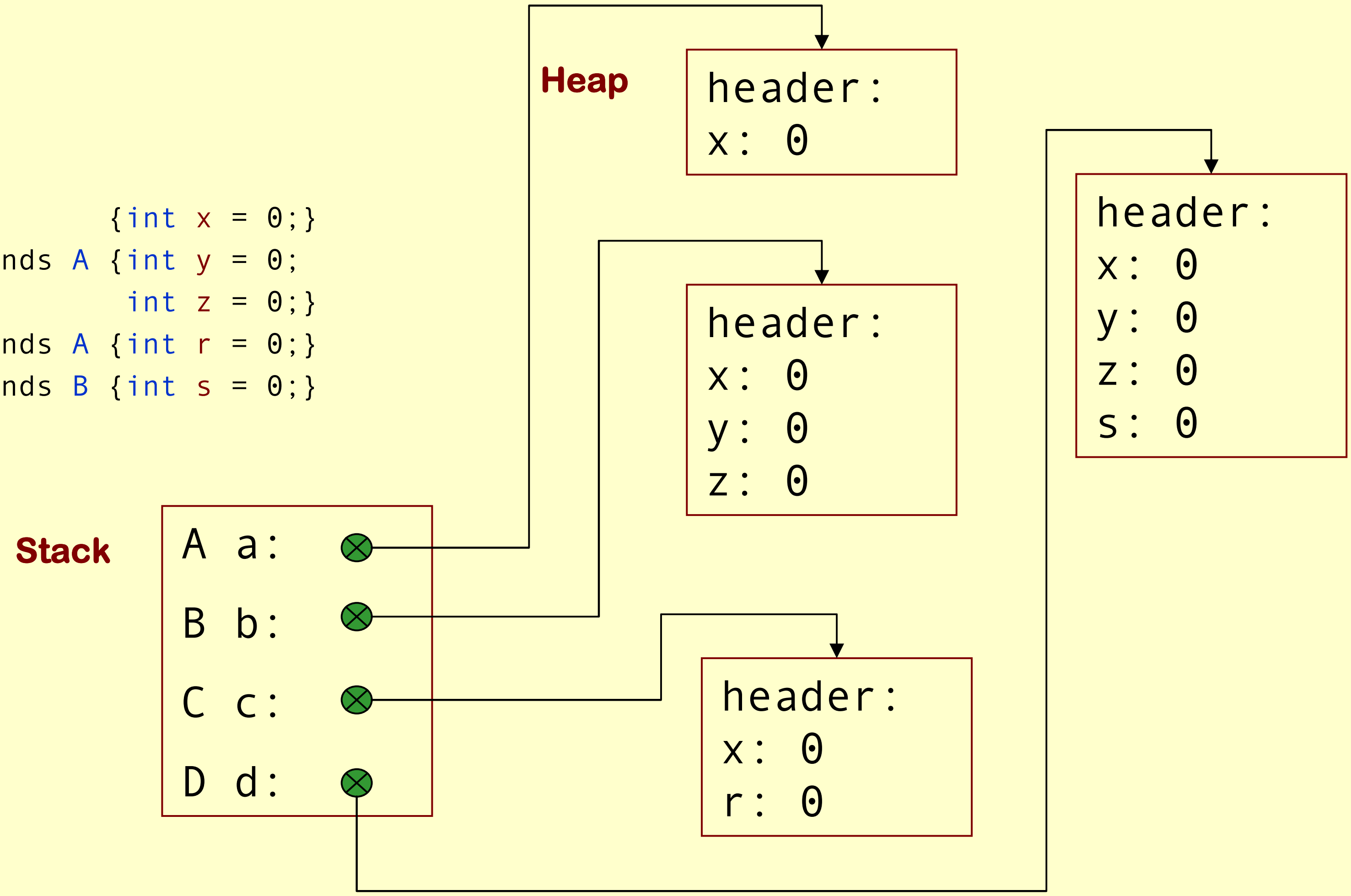
Single Inheritance: Fields

- Example:

```
class A           { int x = 0; }
class B extends A { int y = 0;
                    int z = 0; }
class C extends A { int r = 0; }
class D extends A { int s = 0; }
```

Single Inheritance: Fields

```
class A           {int x = 0;}
class B extends A {int y = 0;
                    int z = 0;}
class C extends A {int r = 0;}
class D extends B {int s = 0;}
```



Offsets:

(A,B,C,D).x: 1

(B,D).y: 2

(B,D).z: 3

(C).r: 2

(D).s: 4

Single Inheritance: Methods

- If we only have single inheritance we can handle methods in much the same way as fields.
- We store addresses to methods in the VMT instead of in the object.
- We copy all the addresses of the super classes to the VMT of the subclasses.
- If a method is overridden we use the address of the new definition instead of the definition in the superclass.

Implementation of OO: Single inheritance

Single Inheritance: Methods

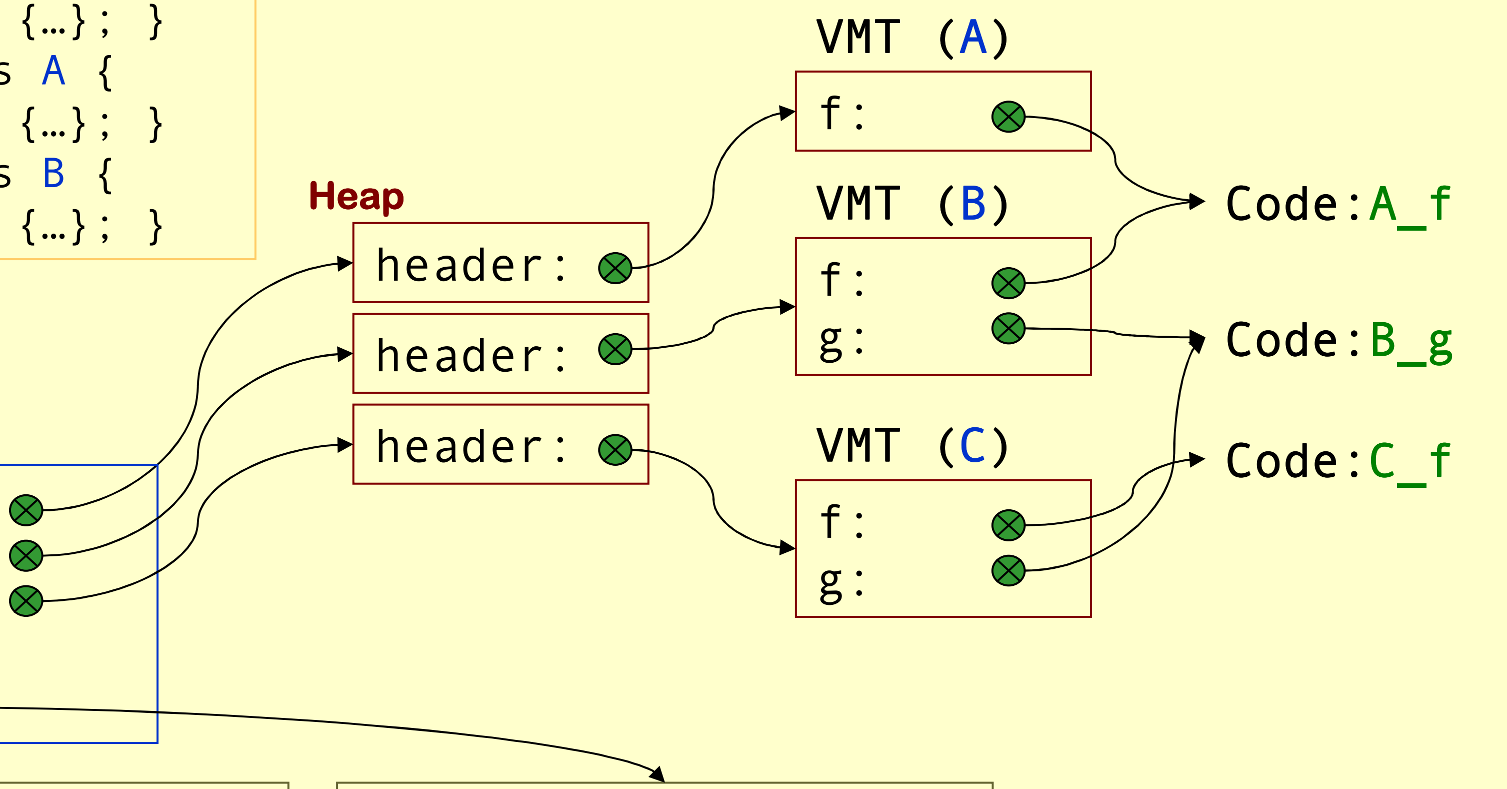
- Example:

```
class A           { int f {…}; }
class B extends A { int g {…}; }
class C extends B { int f {…}; }
```

Single Inheritance: Methods

```
class A {int f {…}; }
class B extends A {
          int g {…}; }
class C extends B {
          int f {…}; }
```

```
A a = new A;
B b = new B;
C c = new C;
b.g();
c.f();
```



```
LD r1,SP(8) ; Get b
LD r2,r1(0) ; Get &VMT(B)
LD r3,r2(4) ; Get &B_g
call r3     ; Call B_g
```

```
LD r1,SP(4) ; Get c
LD r2,r1(0) ; Get &VMT(C)
LD r3,r2(0) ; Get &C_f
call r3     ; Call C_f
```

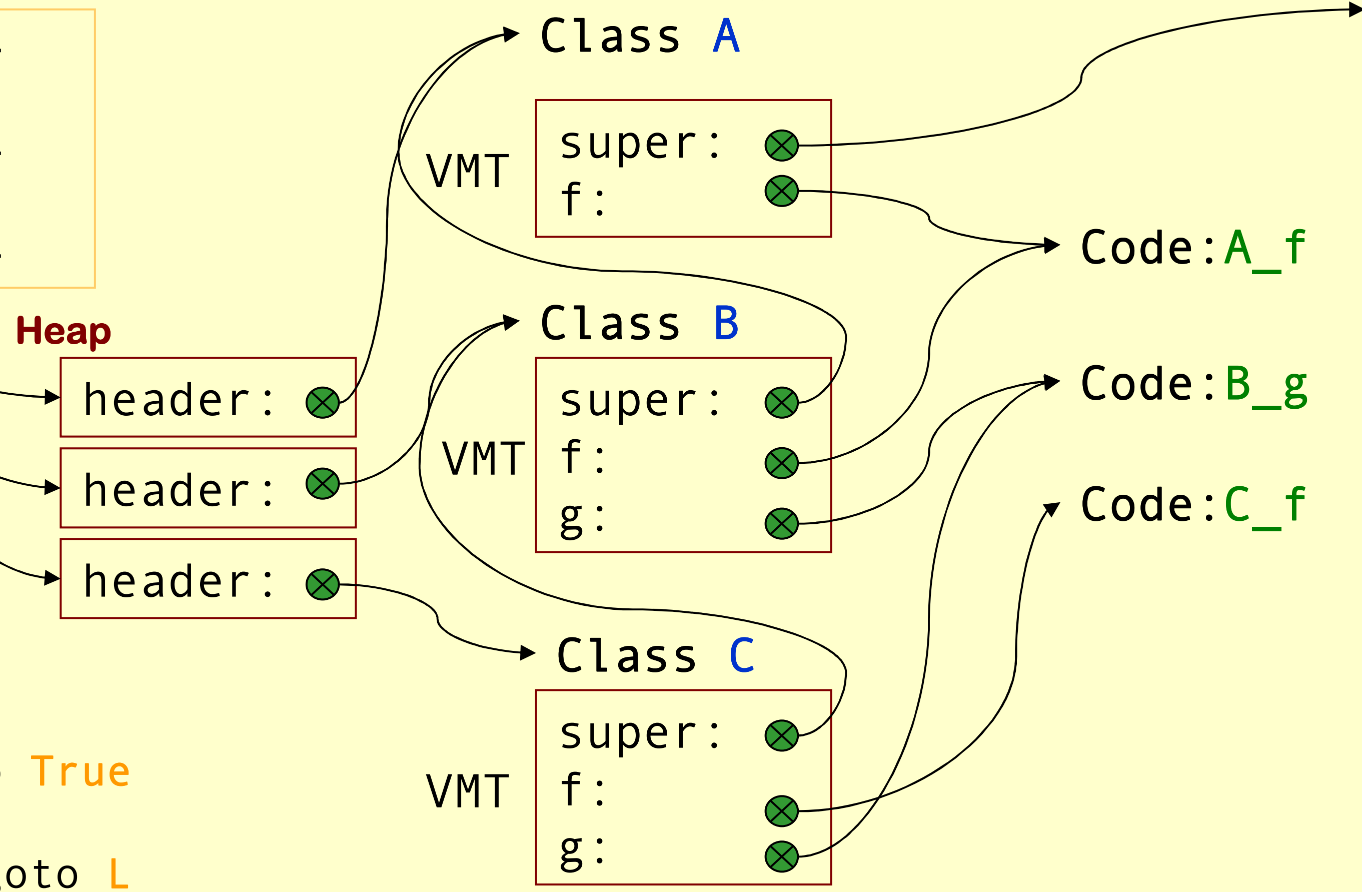
Single Inheritance: Testing Class Membership

- Many OO languages allow you to test class membership of an object.
- In Java there is “`o instanceof C`”.
- An object is a member of all its superclasses.
- We need to be able to find the superclass of a class. Let us extend our implementation with class descriptors.

Single Inheritance: Class Membership

```
class A {int f {…}; }
class B extends A {
          int g {…}; }
class C extends B {
          int f {…}; }
```

```
A a = new A;
B b = new B;
C c = new C;
 c instanceof A;
```



Now we can do
`c instanceof A` as:

```
      t = c.header
L:    if t == A goto True
      t = t.super
      if t != nil  goto L
      res = false
      goto End
True: res = true
End:
```

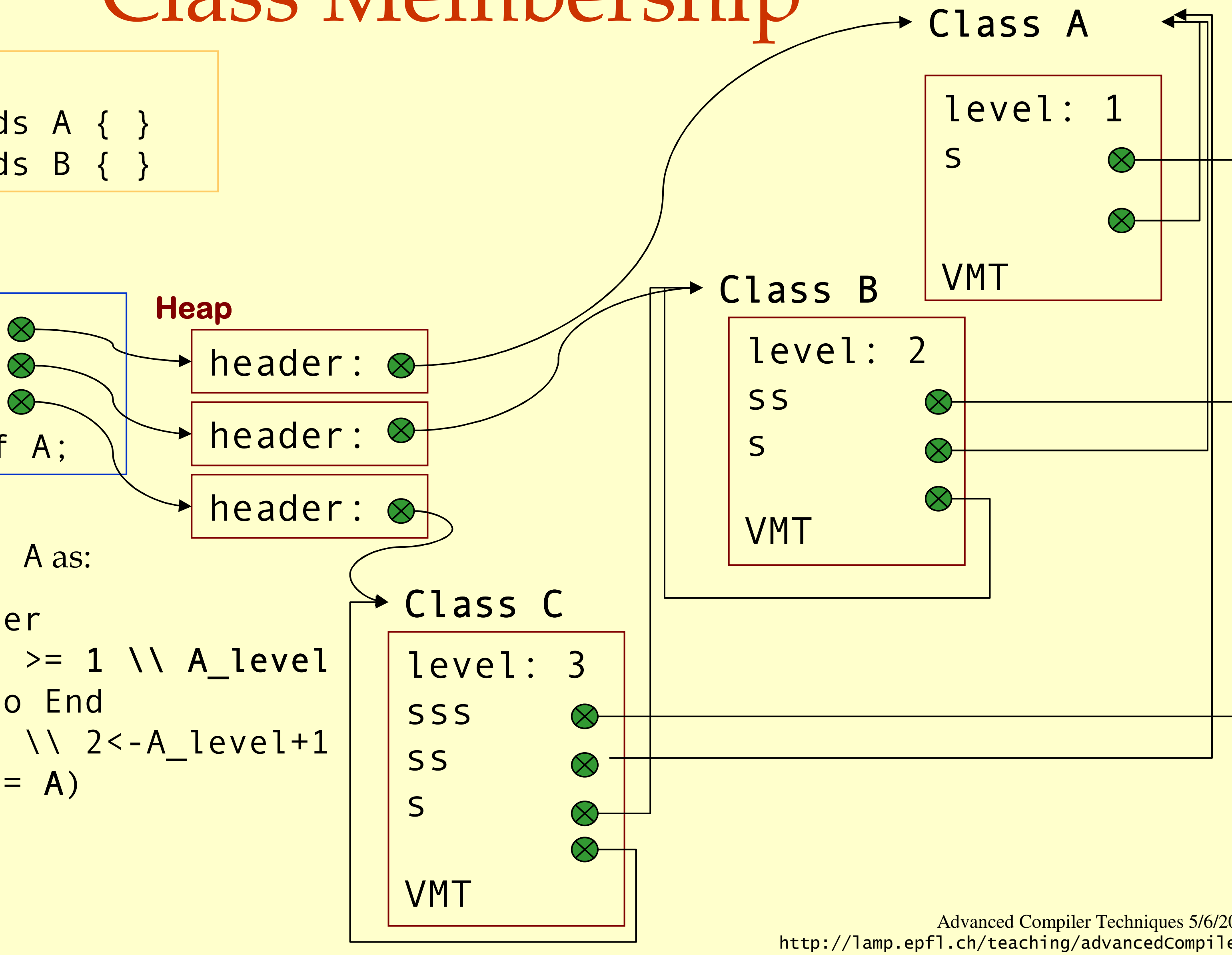
Single Inheritance: Testing Class Membership

- Searching through the class hierarchy is inefficient.
- We can trade space for speed.
- Let each class descriptor have a *display* of all superclasses. I.E., a direct link to each superclass.

Single Inheritance: Class Membership

```
class A {}
class B extends A { }
class C extends B { }
```

```
A a = new A;
B b = new B;
C c = new C;
 c instance of A;
```



Heap

header:

header:

header:

Class A

level: 1

s

VMT

Class B

level: 2

ss

s

VMT

Class C

level: 3

sss

ss

s

VMT

Now we can do
`c instance of A` as:

```
  t1 = c.header
  res = t1[0] >= 1 \\ A_level
  if !res goto End
  t2 = t1[2]  \\ 2<-A_level+1
  res = (t2 == A)
End:
```

Multiple Inheritance

- In languages with multiple inheritance, i.e., where it is possible to extend several parent classes with a class, all the operations we have seen become more difficult.
- Java’s hybrid approach with interfaces complicates these issues in the same way as multiple inheritance.

Multiple Inheritance: Graph Coloring

- One way to handle the layout of fields would be to use graph coloring. (This can also be used for methods.)
- All identical fields would have to occupy the same offset in the object.
- For some objects there would be holes in the array of fields. To reduce the wasted space the fields can be compacted in the object by storing the offsets in the class descriptor.

Multiple Inheritance: Graph Coloring

```
class A             {int x = 0;}
class B             {int y = 0;
                     int z = 0;}
class C extends A,B {int r = 0;}
```

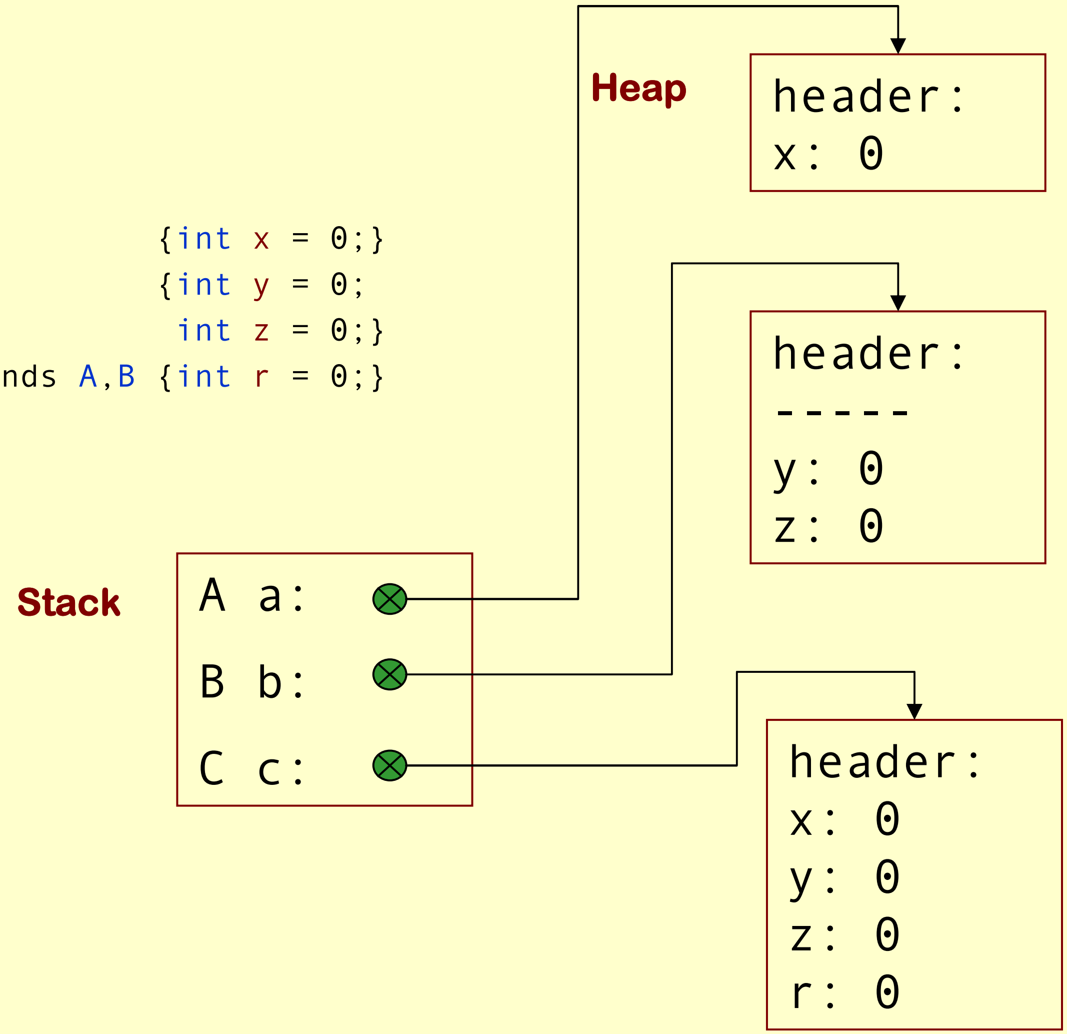
Offsets:

(A,C).x: 1

(B,C).y: 2

(B,C).z: 3

(C).r: 4



Stack

```
A a:
B b:
C c:
```

Heap

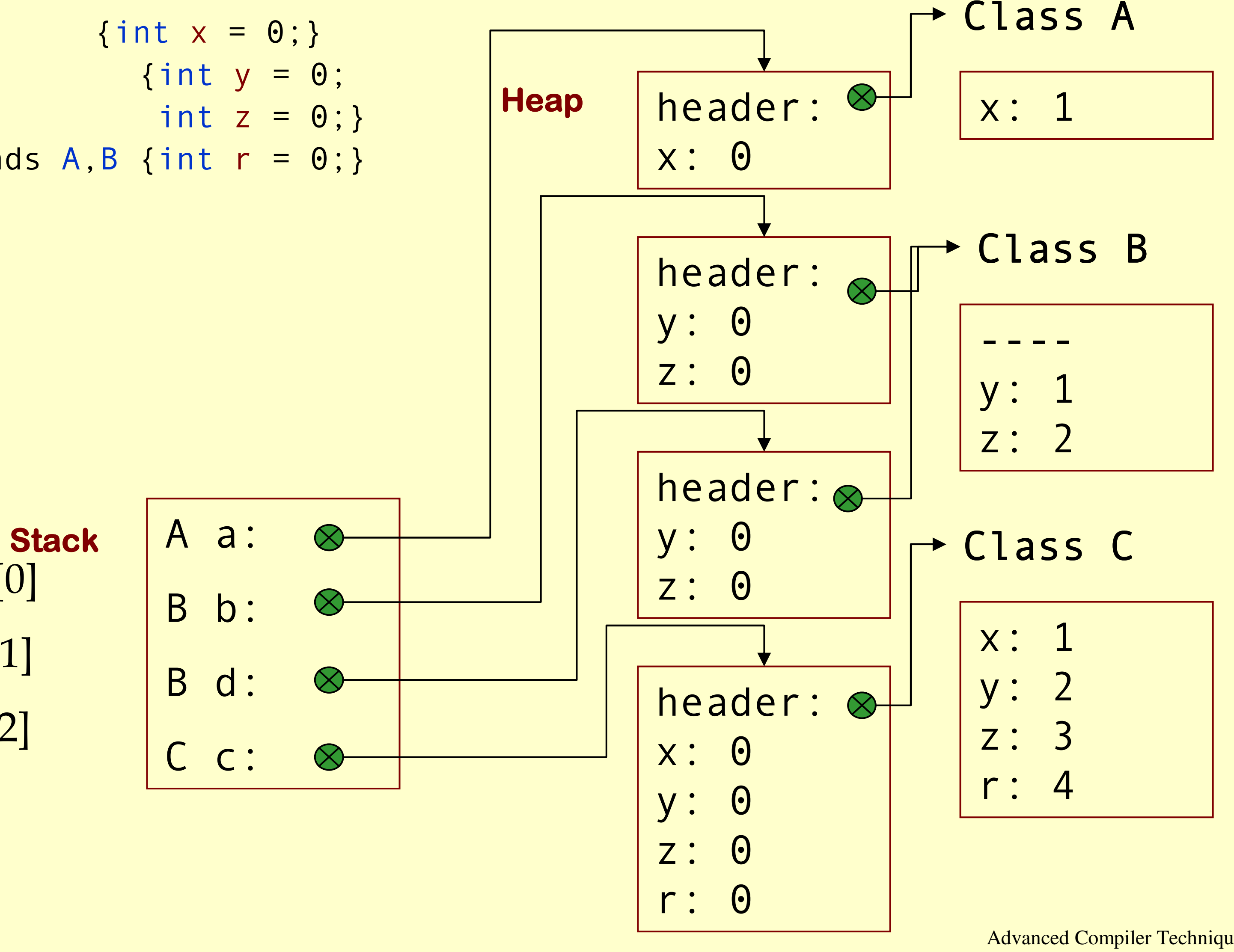
```
header:
x: 0
```

```
header:
-----
y: 0
z: 0
```

```
header:
x: 0
y: 0
z: 0
r: 0
```

Multiple Inheritance: Graph Coloring

```
class A           {int x = 0;}
class B             {int y = 0;
                     int z = 0;}
class C extends A,B {int r = 0;}
A a = new A;
B b = new B;
B d = new B;
C c = new C;
```



Offsets:

(A,C).x: header[0]

(B,C).y: header[1]

(B,C).z: header[2]

(C).r: header[3]

Multiple Inheritance: Graph Coloring

- One problem with global graph coloring is that it is global: you need the whole program – must be done at link time.
- If dynamic linking is possible this approach becomes even harder.

Multiple Inheritance: Hashing

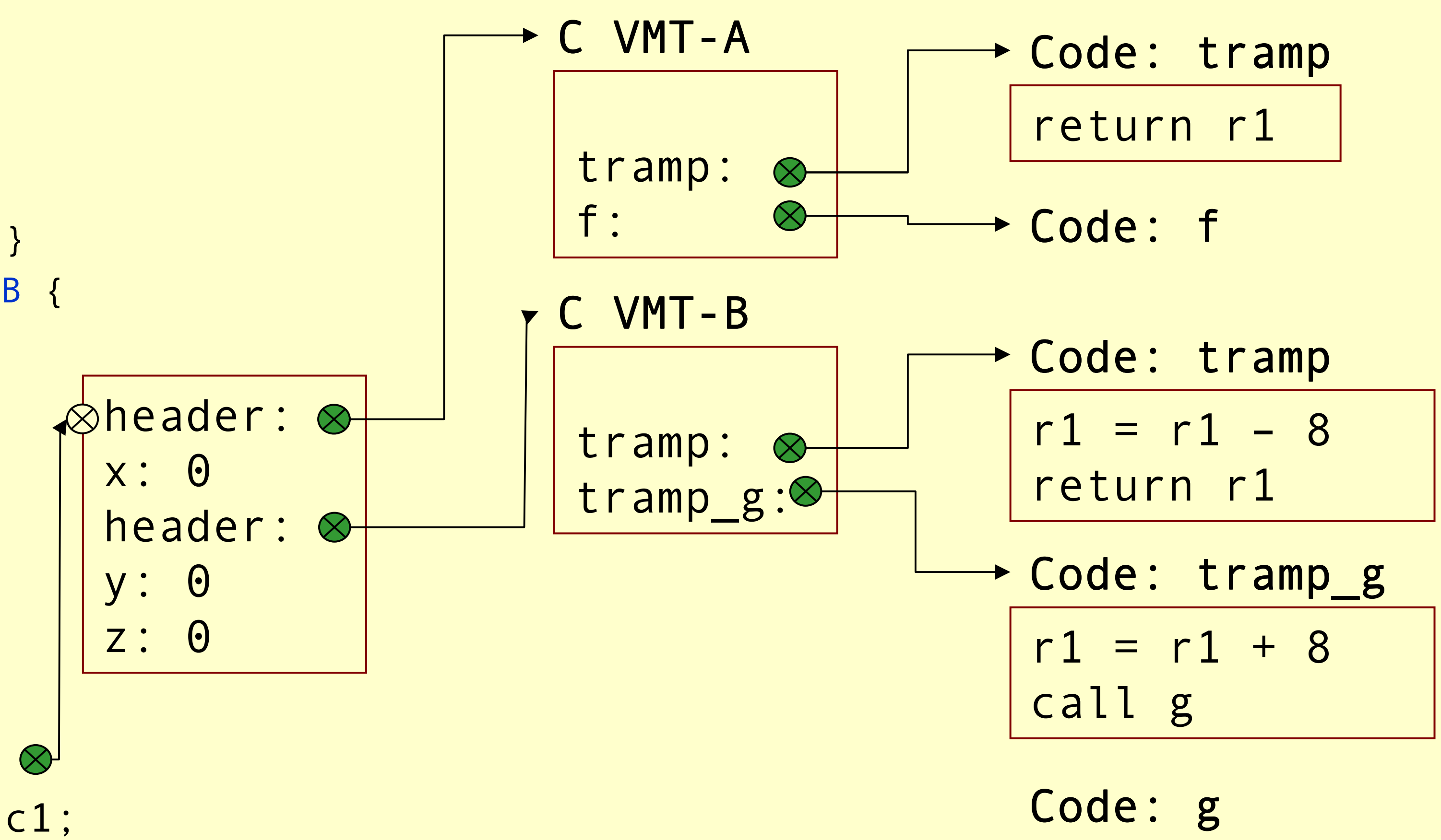
- Second approach: Hashing.
- Instead of a global compile- or link time solution we can calculate a hash value for each name at compile time.
- At runtime we use the hash value as an offset into a hash table in the class descriptor.
- This hash table contains the offset to fields in the object. (This also works for method addresses.)
- This can be costly if there are many collisions in the hash table.

Multiple Inheritance: Trampolines

- Third approach: Trampoline functions.
- We give each object several headers, one for each extended class.
- We add trampoline functions that changes the view of the object from one class to another in an efficient way.

Multiple Inheritance: Trampolines

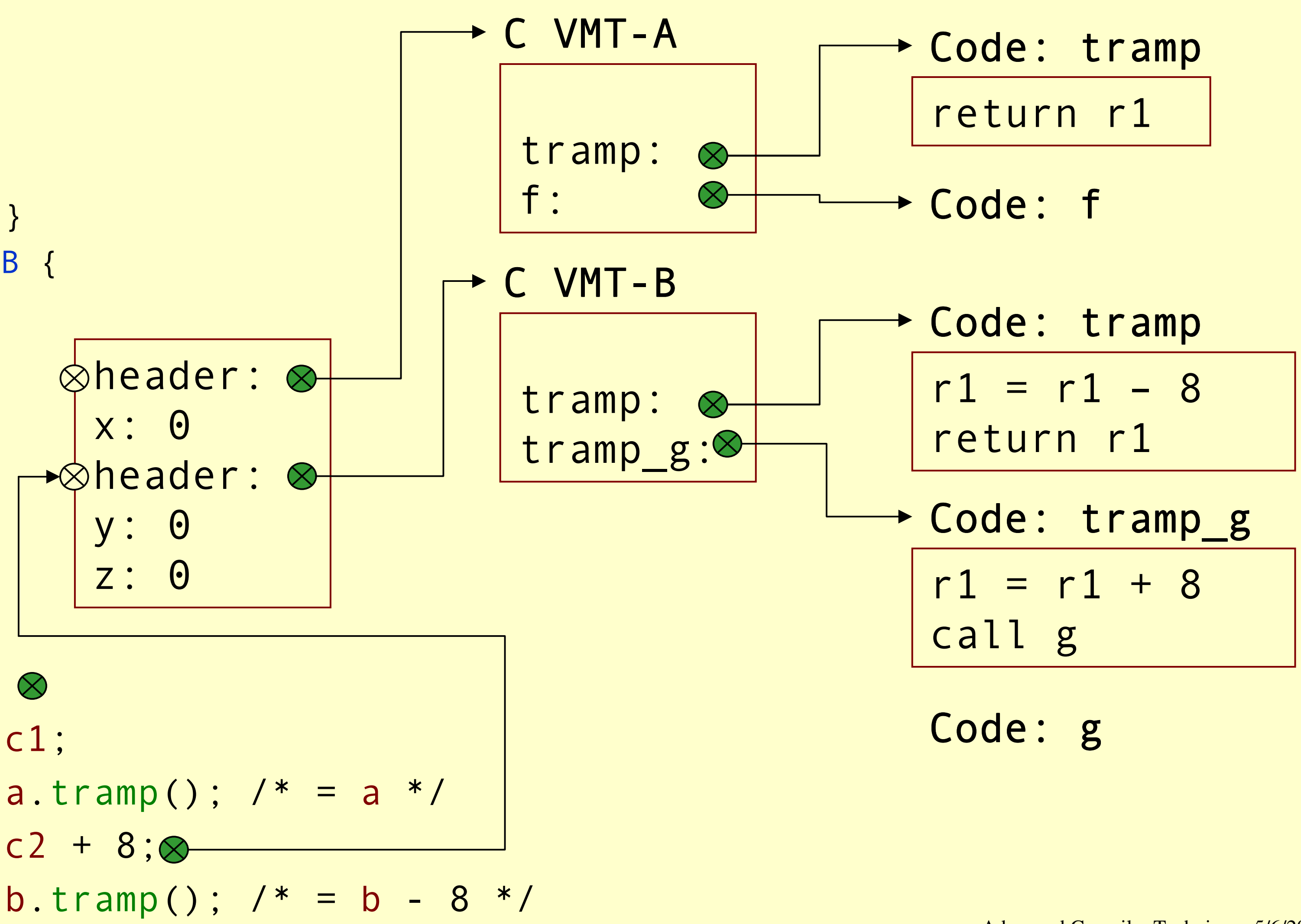
```
class A  {
    int x = 0;
    int f() {…}}
class B  {
    int y = 0;
    int g() {… y …}}
class C extends A,B {
    int z = 0;}
C c1 = new C();
A a  = (A) c1;
C c2 = (C) a;
B b  = (B) c2;
C c3 = (C) b;
```



```
c1 = 
a  = c1;
c2 = a.tramp(); /* = a */
b  = c2+8;
c3 = b.tramp(); /* = b-8 */
```

Multiple Inheritance: Trampolines

```
class A  {
    int x = 0;
    int f() {…}}
class B  {
    int y = 0;
    int g() {… y …}}
class C extends A,B {
    int z = 0;}
C c1 = new C();
A a  = (A) c1;
C c2 = (C) a;
B b  = (B) c2;
C c3 = (C) b;
```



C VMT-A

tramp:
f:

Code: tramp

```
return r1
```

Code: f

C VMT-B

tramp:
tramp_g:

Code: tramp

```
r1 = r1 - 8
return r1
```

Code: tramp_g

```
r1 = r1 + 8
call g
```

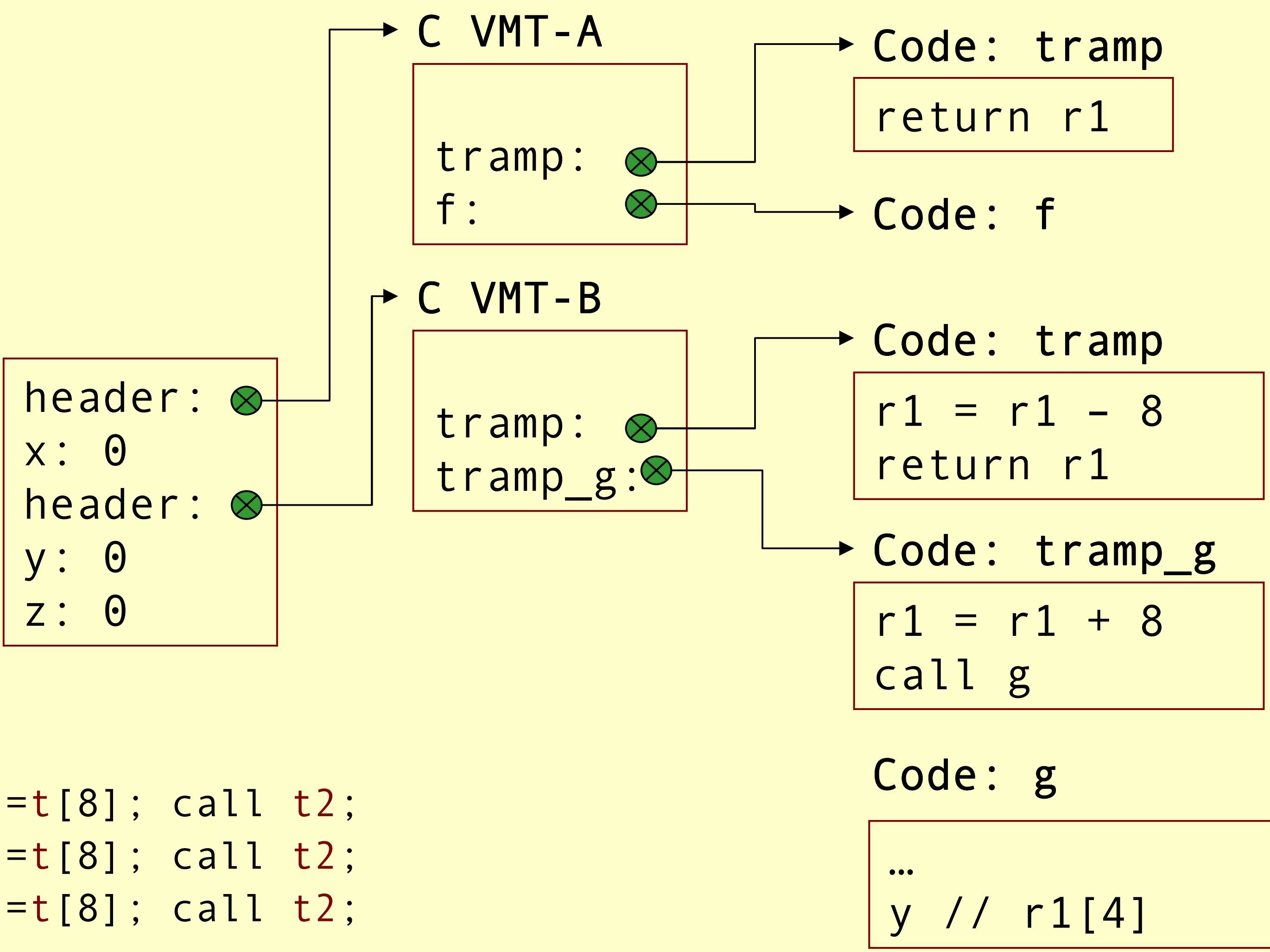
Code: g

header:
x: 0
header:
y: 0
z: 0

```
c1 = 
a  = c1;
c2 = a.tramp(); /* = a */
b  = c2 + 8;
c3 = b.tramp(); /* = b - 8 */
```

Multiple Inheritance: Trampolines

```
class A  {
   int x = 0;
   int f() {…}}
class B  {
   int y = 0;
   inf g() {… y …}}
class C extends A,B {
   int z = 0;}
C c1 = new C();
A a  = (A) c1;
C c2 = (C) a;
B b  = (B) c2;
C c3 = (C) b;
c1.z;   // c1[16]
c1.x;   // c1[4]
c1.y;   // c1[12]
c1.g(); // t=c[8]; t2=t[8]; call t2;
a.f();  // t=a[0]; t2=t[8]; call t2;
b.g();  // t=b[0]; t2=t[8]; call t2;
```



Optimizing OO-Programs

- In modern machines a jump to a known address is much faster than a jump to an address fetched from a table.
- Dynamic dispatch also makes inlining and interprocedural analysis harder.
- Possible solutions: Whole program optimization, link time optimization, JIT compilation, or runtime optimizations.
- When we have the whole program we can turn many dynamic properties into static properties.

Inline caching

- Many dynamic calls actually go to the same class all the time.
- For each call site remember the actual target of the last call.
- Next time jump directly to this location, and check if we end up in the right place.

Polymorphic Inline Caching

- If a call site is polymorphic inline caching can lead to degraded performance.
- Solution: Polymorphic inline caching, remember more than one target address.

Polymorphic Inline Caching

- Polymorphic inline caching can be implemented with an if then else search tree:

```
v.f()
  ⟹
  if c.header < C {
    if c.header < B A.f() else B.f()
 } else {
   if c.header < D C.f() else D.f()
 }
```

OO: Summary

- Implementing OO efficiently means implementing inheritance efficiently.
- There are several possible solution available and there is still research going on in this area.
- One of the most successful techniques for optimizing OO is to do it at runtime using JIT compilation – something we will look closer at later in the course.

Implementation of Functional Programming Languages

- There is no common agreement on exactly what a functional programming language is. But usually such a language should have at least one of the following concepts:
  - No statements – only functions (or expressions).
  - Higher order functions.
  - Pureness (no side effects).
  - Laziness.
  - Automatic memory management (Garbage collection.)

Higher Order Functions

- In Eins (and in C) you have “second”-order functions.
  - That is, functions are also values in the language: you can take their addresses and pass them around and apply them.

    ```
    def apply(f: (Int) => Int, x: Int): Int = f(x);
    ```

  - These functions can be represented with just a function pointer, i.e., the address of the function.
- Functions that take functions as arguments are called *higher order functions*.
- For a language to have interesting higher order functions you need to be able to create new functions at runtime. E.g., in Scala you can write:

  ```
  val f:(Int => Int) = x => x + 1;
  ```

Higher Order Functions

- To get **really** interesting functions at runtime you need to be able to capture the *free variables* of the function.
  - A free variable is a variable that is not bound by the definition of the function. (`y` is free in `x => x+y`.)

    ```
    def f(y:Int):(Int => Int)  =  x => x + y;
    ```
- In order to do this we need ***closures***.
- A closure is a data structure that contains a function pointer and a way to access all free variables of the body of the function.

Higher Order Functions

- In an OO language a closure can be implemented as an object with a single method and several instance variables.

```
def f(y:Int):(Int => Int) = x => x + y;
f(42)(17)
```

⟹

```
   class F {
        int y;
        public F(int y) { this.y = y; }
        public int apply(int x) {
          return x+y;
        }
   }

   public F f(int y) = new F(y);
   f(42).apply(17);
```

Implementation of FPL

Higher Order Functions

- This is more or less the way Scala implements functions.
- To make it more general we can make all closures implement the `Function` interface:

```
public interface Function1 {
  public abstract java.lang.Object apply(java.lang.Object a0);
}
```

- We also need to take care of local (mutable) variables that are captured by the function. This can be done by turning them into references.

Implementation of FPL

Higher Order Functions

```
def f(y:Int):(Int => Int) = {
   var z = y*2;
   val f = x=>x+z;
   z = z +1;
   f;
}
```

⇒

```
   class F {
    IntRef y;
    public F(IntRef y) {
      this.y = y; }
    public int apply(int x) {
          return x+y.v;
        }
   }
```

```
class IntRef {
   int v;
   public IntRef(int i) {v=i;}
   public set(int i) {v=i;}
}

public F f(int y) =  {
   IntRef z = new IntRef(y*2);
   F f = new F(z);
   z.set(z.v + 1);
   return f;
}
```

Pure Functional Languages

- In a *pure functional language* there are no *side effects*.
- This includes no updates of variables. That is, variables are immutable.
  - Variables are, like variables in mathematics, just names for values.
  - If we say `x = 42;` then we give the value 42 a new name: x, from now on x and 42 are interchangeable.
- With a pure functional language it is possible to do *equational reasoning*.

Lazy Evaluation

- With *lazy evaluation*, an expression is not evaluated unless its value is demanded by some other part of the computation.
- In contrast, strict languages (Java, ML, C, Erlang) evaluate each expression as the control flow reaches it.

Call-by-Name Evaluation

- Most languages pass function arguments using call-by-value:
  - i.e. all arguments are evaluated before a function is called.
  - e.g. in the expression `f(g(x+y))`, first `(x+y)` is evaluated then the function g is called before the function f is called.
  - If the function f doesn't use its argument then the evaluation of g and of `x+y` is done in vane.

Call-by-Name Evaluation

- *Call-by-name* evaluation avoids this problem by not evaluating the arguments, instead a *thunk* is created for each argument.
- A *thunk* is a function that can be called to compute the value on demand.

  `f(g(x+y))` is translated to
  `f(()=>g(()=>x+y))`
- Any use of the argument in f is replaced by an application of the function:

  `f(x) = x;` is translated to
  `f(x) = x();`

Call-by-Name Evaluation

- Scala provides call-by-name with explicit `def` parameters.
- A problem with call-by-name is that a thunk may be executed many times.

  `f(x) = x+x;` is translated to

  `f(x) = x()+x();`

Call-by-Need

- With *call-by-need* each thunk is only evaluated once.
- This is implemented by giving each thunk a *memo slot* that stores the evaluated value; each evaluation of the thunk first checks the memo slot: if it is empty the expression is evaluated and stored in the slot, otherwise the value in the slot is returned.

Call-by-Need

Conceptually a thunk for x+y can be implemented as:

```
class Thunk {
    res = null;
    apply() = {
        if res == null then res = x+y
        else res
    }
}
```

Call-by-need

- A thunk can also be implemented just as two words <thunk_function, memo_slot>
- When the thunk is evaluated both fields are updated: the memo slot with the value and the function with a new function that returns the value.

Optimization of FP

- Functional programs tend to use many small functions. Modern hardware is optimized for imperative programs with few large functions, i.e., function calls are relatively expensive.
- Hence it can be profitable to reduce the number of function calls and increase the size of functions.
- This can be done by *inline expansion*.

Inline Expansion

- Inline expansion or *inlining* is an optimization where a function call is replaced by the body of the function.
- If this is done in a stage in the compiler where all independent names are replaced by unique symbols then the process is quite straightforward. Otherwise the formal parameters need to be renamed (α-converted).

Inline Expansion

- If inline expansion is applied indiscriminately, the size of the program explodes.
- To limit the code growth we can:

1. Expand only frequent call sites.
2. Expand only small functions.
3. Expand functions called only once, and perform *dead function elimination*.

Inline Expansion

- If we inline a recursive function just as any other function we would probably end up with a call to the original function. Either directly after the first iteration or after a while.

Inline Expansion

```
def loop(int x, int max, int y) =
   if (x > max) y
   else loop(x + 1, y * y);

def f(int z) =
   loop(1,10,z);
```

→

```
def f(int z) = {
   val x = 1; val max = 10; val y = z;
   if (x > max) y
   else loop(x + 1, max, y * y);
}
```

Inline Expansion

- To remedy this we can bring the definition of the recursion with us in the inlining by splitting the function into a *prelude* and a *loop header*.

Implementation of FPL

Inline Expansion

```
def loop(int x, int max, int y) =
    if (x > max) y
    else loop(x + 1, y * y);

def f(int z) =
    loop(1,10,z);
→
def f(int z) = {
    val x = 1; val max = 10; val y = z;
    val loop =
        (int xX, int maxX, int yX) =>
            if (xX > maxX) yX
            else loop(xX + 1, maxX, yX * yX);
    if (x > max) y
    else loop(x + 1, max, y * y);
  }
```

Loop-Invariant Hoisting

- We can avoid passing around values that are the same in each recursive call by using *loop-invariant hoisting*.
- Just let the constant value become a free variable.
- In our example lift max from an argument to a free variable.

Loop-Invariant Hoisting

```
def loop(int x, int max, int y) =
   if (x > max) y
   else loop(x + 1, y * y);

def f(int z) =
   loop(1,10,z);


→

def f(int z) = {
   val x = 1; val max = 10; val y = z;
   val loop =
      (int xX, int yX) =>
         if (xX > max) yX
         else loop(xX + 1, yX * yX);
   if (x > max) y
   else loop(x + 1, y * y);
  }
```

Inline Expansion

- Inline expansion in itself can be useful since the overhead for a function call and return is removed, but **the real benefit** comes from applying standard optimizations on the inline expanded program.
- Constant propagation and folding, dead code and unreachable code elimination all work better when the scope (of a function) is increased.

Inline Expansion & constant prop

```
def loop(int x, int max, int y) =
    if (x > max) y
    else loop(x + 1, y * y);

def f(int z) =
    loop(1,10,z);


→
def f(int z) = {
    val x = 1; val max = 10; val y = z;
    val loop =
        (int xX, int yX) =>
            if (xX > 10) yX
            else loop(xX + 1, yX * yX);
    if (x > 10) y
    else loop(x + 1, y * y);
  }
```

Inline Expansion & copy prop

```
def loop(int x, int max, int y) =
    if (x > max) y
    else loop(x + 1, y * y);

def f(int z) =
    loop(1,10,z);
```

→

```
def f(int z) = {
    val x = 1; val max = 10; val y = z;
    val loop =
        (int xX, int yX) =>
           if (xX > 10) yX
           else loop(xX + 1, yX * yX);
    if (1 > 10) z
    else loop(1 + 1, z * z);
  }
```

Inline Expansion
& constant folding

```
def loop(int x, int max, int y) =
    if (x > max) y
    else loop(x + 1, y * y);

def f(int z) =
    loop(1,10,z);
```

→

```
def f(int z) = {
    val loop =
        (int xX, int yX) =>
            if (xX > 10) yX
            else loop(xX + 1, yX * yX);
    loop(2, z * z);
  }
```

Efficient Tail Calls

- A function call f(x) within a body of a function g is in a *tail position* if calling f is the last thing g will do before returning.
- We can save stack space and execution time by turning the call to f into a jump to f.
- For some languages, like Erlang and Scheme, proper tail calls is not an optimization but a **requirement**.

Tail Calls

- A tail call can be transformed from a call to a jump as follows:
  1. Move actual parameters into argument registers (and stack positions).
  2. Restore callee-save registers.
  3. Pop the stack frame of the calling function.
  4. Jump to the callee.
- If both the caller and the callee have few arguments so that they all fit in argument registers then step 1 might be eliminated by a coalescing register allocator, and step 2 and 3 might also be unnecessary: the tail call becomes just a jump.

Equational Reasoning

- In a pure language we can perform *β-substitution*.
  - That is, replacing a call to a function with a version of the body of the function where each occurrence of the formal parameter is replaced by the argument.
  - ((x) => x + x)(42)   β→ 42 + 42
- Basically: we can perform function calls at compile time.

Optimization of Lazy FP

- A lazy language allows us to do some optimizations that would not be safe in a strict language:
  - Invariant hoisting.
  - Dead code removal (of function calls).
  - Strictness Analysis.

Implementation of FPL

Optimization of Lazy FP

- Invariant hoisting:

```
def f(i) = {
      def g(j) = h(i) * j;
      g
 }
---
def f(i) = {
    val h = h(i);
      def g(j) = h * j;
      g
 }
```

- If h(n) loops infinitely but the result of f(n) is never called a strict language would loop in the call to f(n).

Optimization of Lazy FP

- Dead code removal:

```
def f(i:int): int = {
   var d = g(x);
   i + 2;
 }
```

- In an imperative language g(x) can not be removed, there might be side effects.
- In a strict pure language removing g(x) might turn a non-terminating computation into a terminating one.

Optimization of Lazy FP

- The overhead of thunk creation and evaluation is quite high, so they should only be used when needed.
- If a function f(x) is certain to evaluate its argument x, there is no need to create a thunk for x.
- We can use a ***strictness analysis*** to find out which arguments should be evaluated at the call site and which should be passed as thunks.
- In general exact strictness analysis is **not computable** – a conservative approximation must be used, i.e., assume that arguments who can not be proved strict are non-strict.