Compiling Scala to LLVM

Geoff Reedy

Strange Loop 2012

Why Scala on LLVM? – Native code

Deploy Scala where a JVM is

not available *not desired* *old and slow*

For example

Google Native Client *Embedded Systems*

Apple iOS

Why Scala on LLVM? – Fast startup

JVM startup dominates running time of short programs

→ Scala+JVM is not so great for scripting and utilities

Ahead-of-time compilation produces native binaries

→ Small utilities spend most time doing useful work

Why Scala on LLVM? – Efficient implementation

LLVM allows more efficient implementations of

traits *anonymous functions*
structural types *boxed values*

Why Scala on LLVM? – The rest

Language implementation research

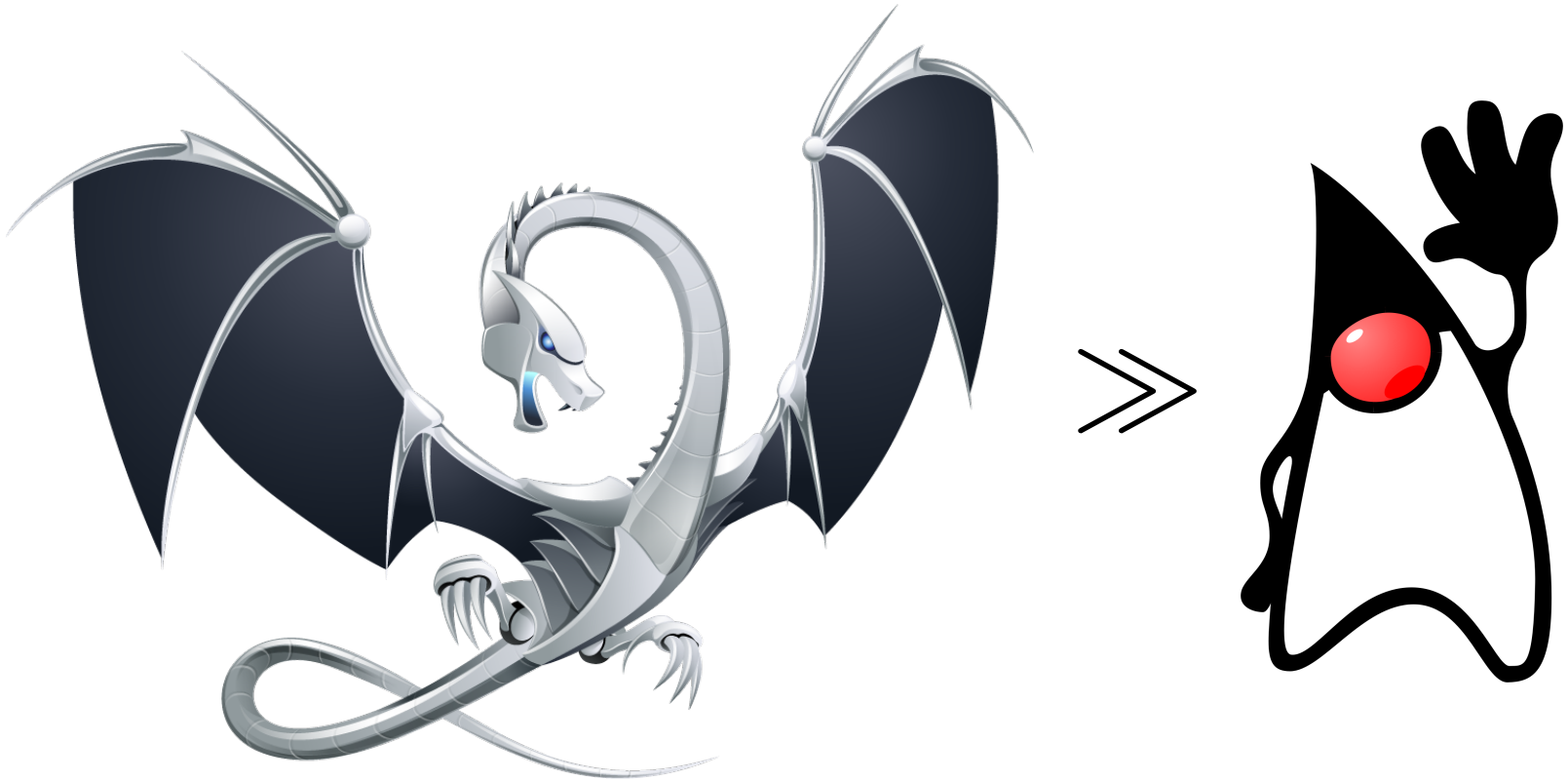
Scala+LLVM can be a place for innovation in language implementation issues

Multi-platform language

Scala already lets the programmer choose the right paradigm

Let them pick the right platform too

LLVM also has a sick wyvern logo



What is LLVM?

LLVM is not a VM. But it is...

- an abbreviation of Low Level Virtual Machine
- a universal assembly language
- a framework for program optimization and analysis
- an ahead of time compiler
- a just in time compile r
- a way to get fast native code without writing your own code generation

LLVM IR Sample

Figure : Factorial Function

```
define i32 @factorial(i32 %n) {
entry:
  %iszero = icmp eq i32 %n, 0
  br i1 %iszero, label %return1, label %recurse
return1:
  ret i32 1
recurse:
  %nminus1 = add i32 %n, -1
  %factnminusone =
    call i32 @factorial(i32 %nminus1)
  %factn = mul i32 %n, %factnminusone
  ret i32 %factn
}
```

LLVM analysis and optimization

LLVM is more than just an assembler

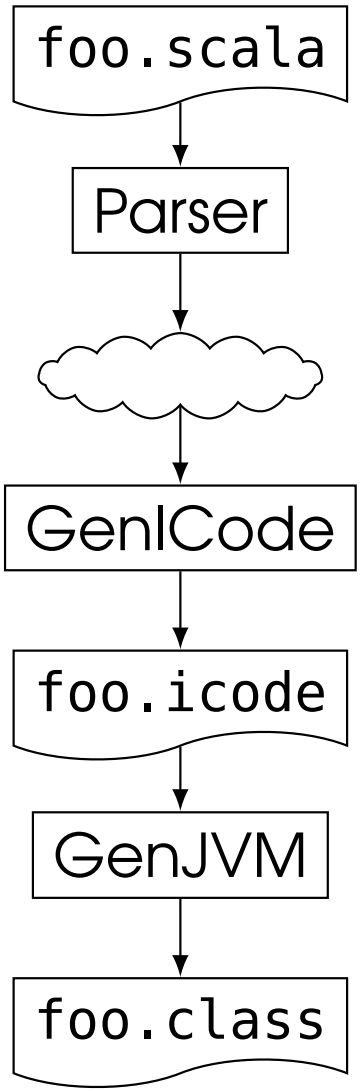
Analyses

Alias Analysis Liveness Analysis Def-Use Analysis
Memory Dependence Analysis and more...

Optimizations

Constant Propagation Loop Unrolling Function Inlining
Dead Code Elimination Peephole Optimizations
Partial Specialization Link-time Optimization and more...

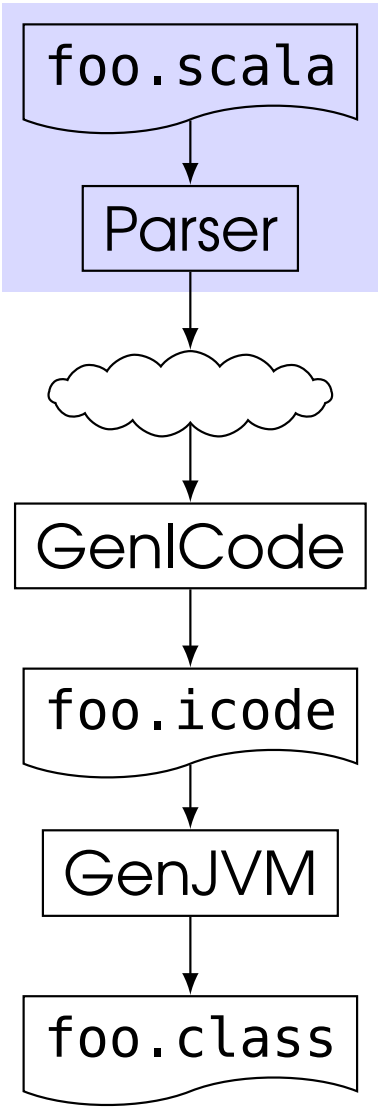
Compiler phases



The Scala compiler is organized as a pipeline of phases.

1. Source code is parsed into syntax trees
2. Syntax trees are typed, transformed, lifted, lowered, desugared
3. ICode is generated from the syntax trees
4. Java bytecode is generated from ICode

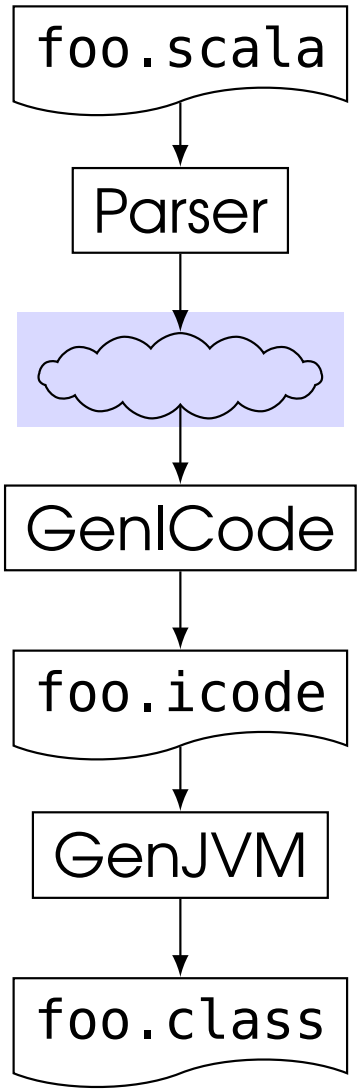
Compiler phases



The Scala compiler is organized as a pipeline of phases.

1. Source code is parsed into syntax trees
2. Syntax trees are typed, transformed, lifted, lowered, desugared
3. ICode is generated from the syntax trees
4. Java bytecode is generated from ICode

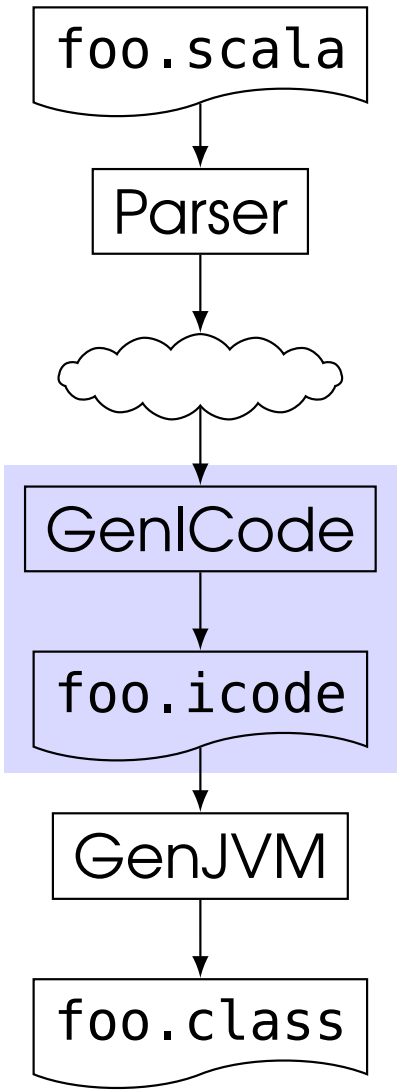
Compiler phases



The Scala compiler is organized as a pipeline of phases.

1. Source code is parsed into syntax trees
2. Syntax trees are typed, transformed, lifted, lowered, desugared
3. ICode is generated from the syntax trees
4. Java bytecode is generated from ICode

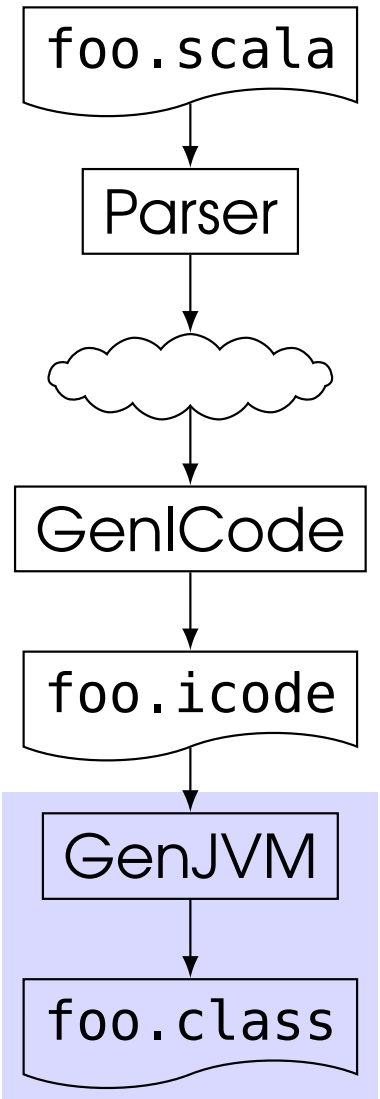
Compiler phases



The Scala compiler is organized as a pipeline of phases.

1. Source code is parsed into syntax trees
2. Syntax trees are typed, transformed, lifted, lowered, desugared
3. ICode is generated from the syntax trees
4. Java bytecode is generated from ICode

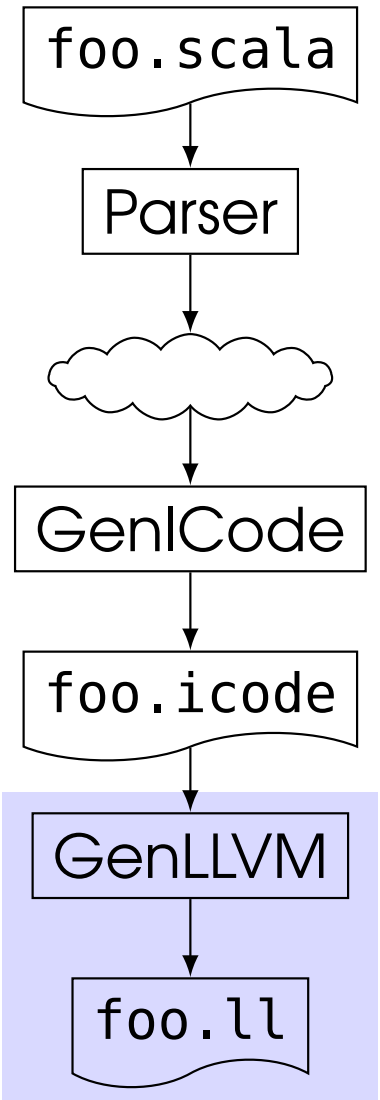
Compiler phases



The Scala compiler is organized as a pipeline of phases.

1. Source code is parsed into syntax trees
2. Syntax trees are typed, transformed, lifted, lowered, desugared
3. ICode is generated from the syntax trees
4. Java bytecode is generated from ICode

Compiler phases



The Scala compiler is organized as a pipeline of phases.

1. Source code is parsed into syntax trees
2. Syntax trees are typed, transformed, lifted, lowered, desugared
3. ICode is generated from the syntax trees
4. LLVM IR is generated from ICode

ICode

ICode is the compiler's internal intermediate representation. Unlike LLVM IR, it is stack based.

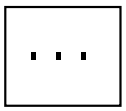
```
def fact(n: Int): Int = {
  if (n == 0) 1 else n * fact(n-1)
}

def fact(n: Int (INT)): Int {
  locals: value n; startBlock: 1; blocks: [1,2,3,4]

    1: LOAD_LOCAL(value n)        CONSTANT(1)
       CONSTANT(0)                CALL_PRIMITIVE(Arithmetic(SUB,INT))
       CJUMP (INT)EQ ? 2 : 3      CALL_METHOD fact.fact (dynamic)
    2: CONSTANT(1)                CALL_PRIMITIVE(Arithmetic(MUL,INT))
       JUMP 4                     JUMP 4
    3: LOAD_LOCAL(value n)     4: RETURN(INT)
       THIS(fact)
       LOAD_LOCAL(value n)

}
```

Translating ICode to LLVM

ICode fragment:

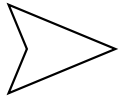
```
LOAD_LOCAL(value n)

CONSTANT(1)

CALL_PRIMITIVE(Arithmetic(SUB,INT))
```

Stack map:



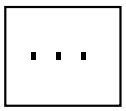
Translating ICode to LLVM

ICode fragment:



```
LOAD_LOCAL(value n)
CONSTANT(1)
CALL_PRIMITIVE(Arithmetic(SUB,INT))
```

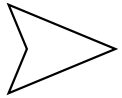
Stack map:




```
%n = load i32* %local.n
```

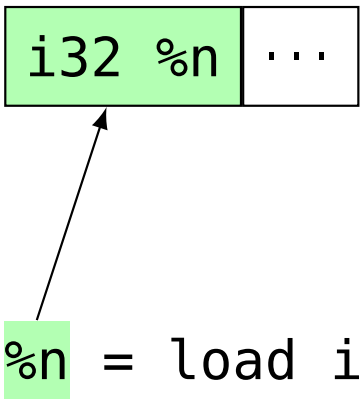
Translating ICode to LLVM

ICode fragment:



```
LOAD_LOCAL(value n)
CONSTANT(1)
CALL_PRIMITIVE(Arithmetic(SUB,INT))
```

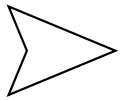
Stack map:



Translating ICode to LLVM

ICode fragment:

```
LOAD_LOCAL(value n)
CONSTANT(1)
CALL_PRIMITIVE(Arithmetic(SUB,INT))
```



Stack map:

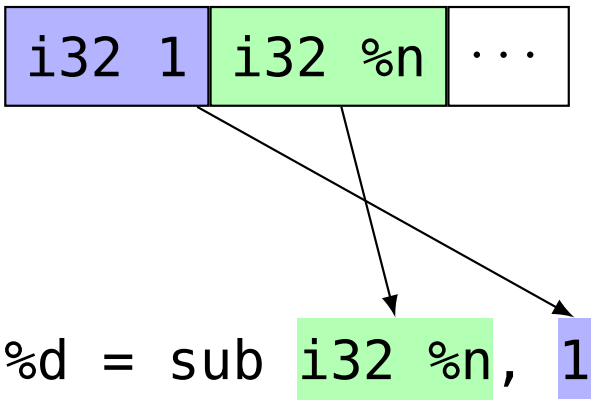
`i32 1`	`i32 %n`	···

Translating ICode to LLVM

ICode fragment:

```
LOAD_LOCAL(value n)
CONSTANT(1)
CALL_PRIMITIVE(Arithmetic(SUB,INT))
```

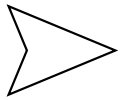
Stack map:



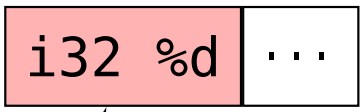
Translating ICode to LLVM

ICode fragment:

```
LOAD_LOCAL(value n)
CONSTANT(1)
CALL_PRIMITIVE(Arithmetic(SUB,INT))
```



Stack map:



```
%d = sub i32 %n, 1
```

References

```
x.xyzzy()
```

References

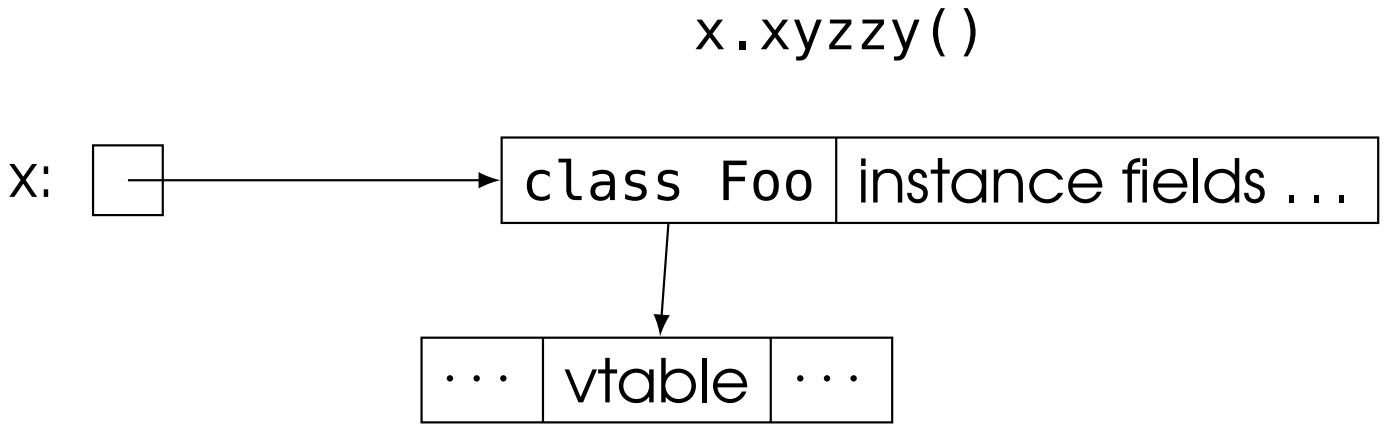
```
x.xyzzy()
```

x: 

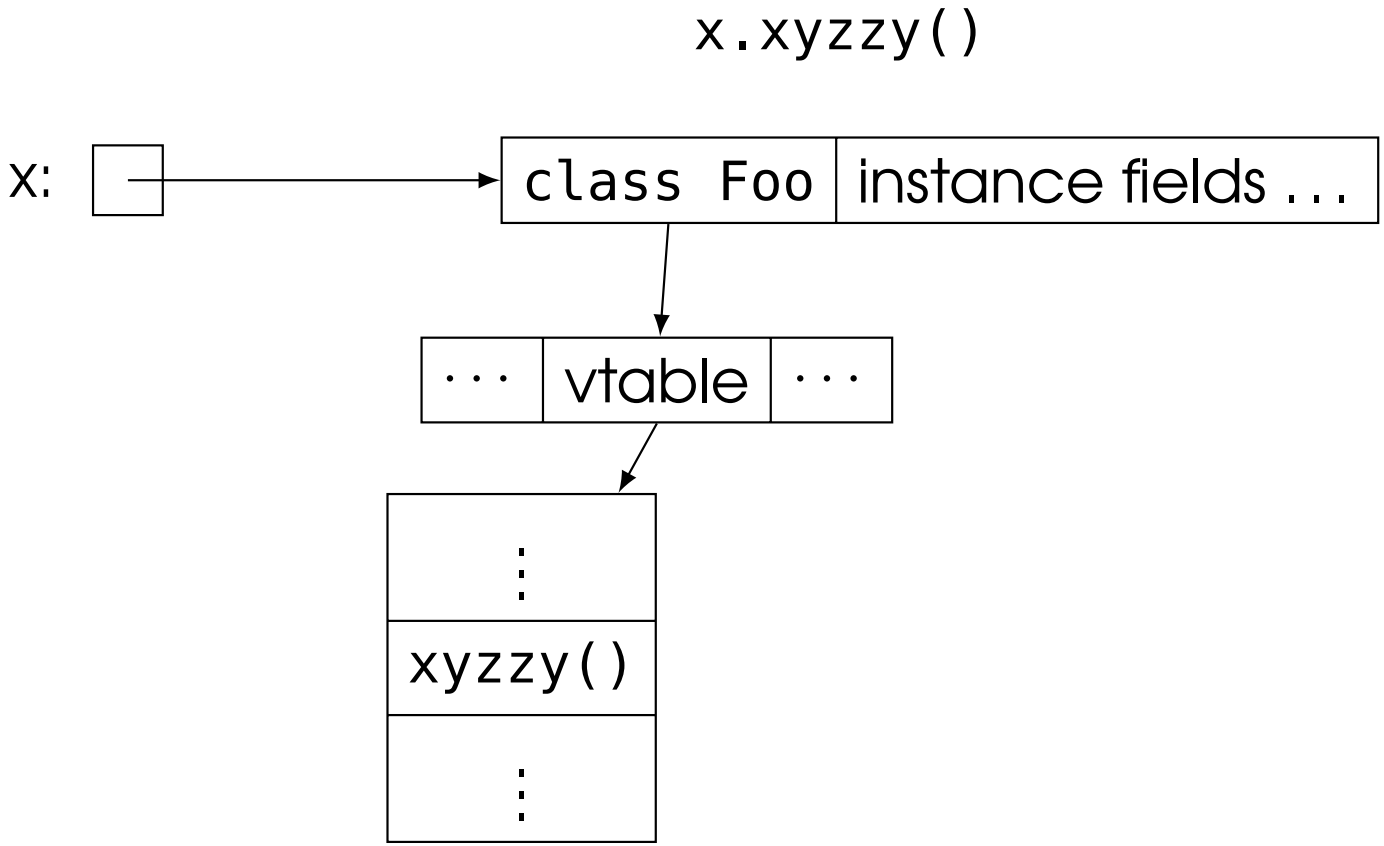
References

```
x.xyzzy()
```

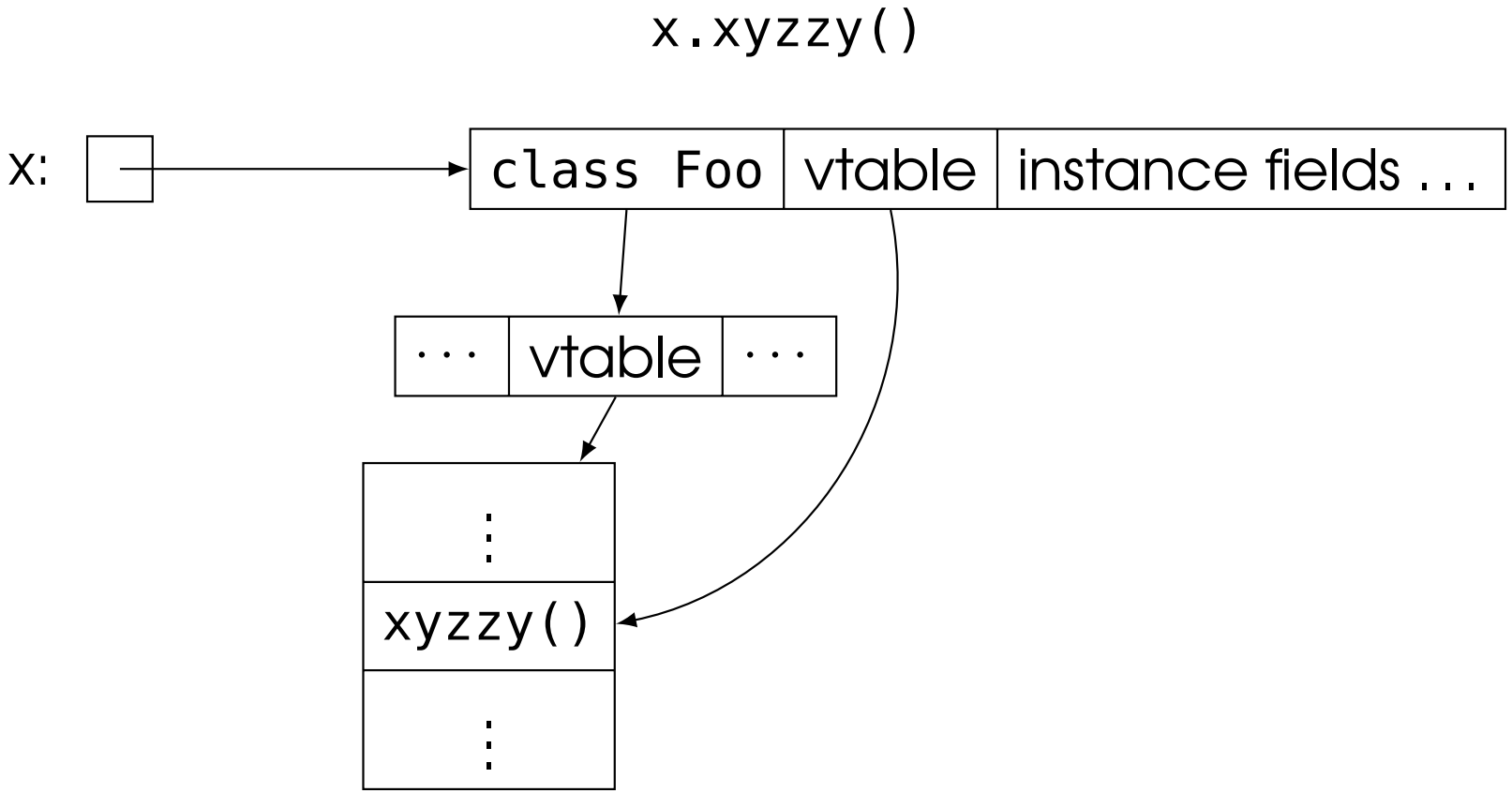
References



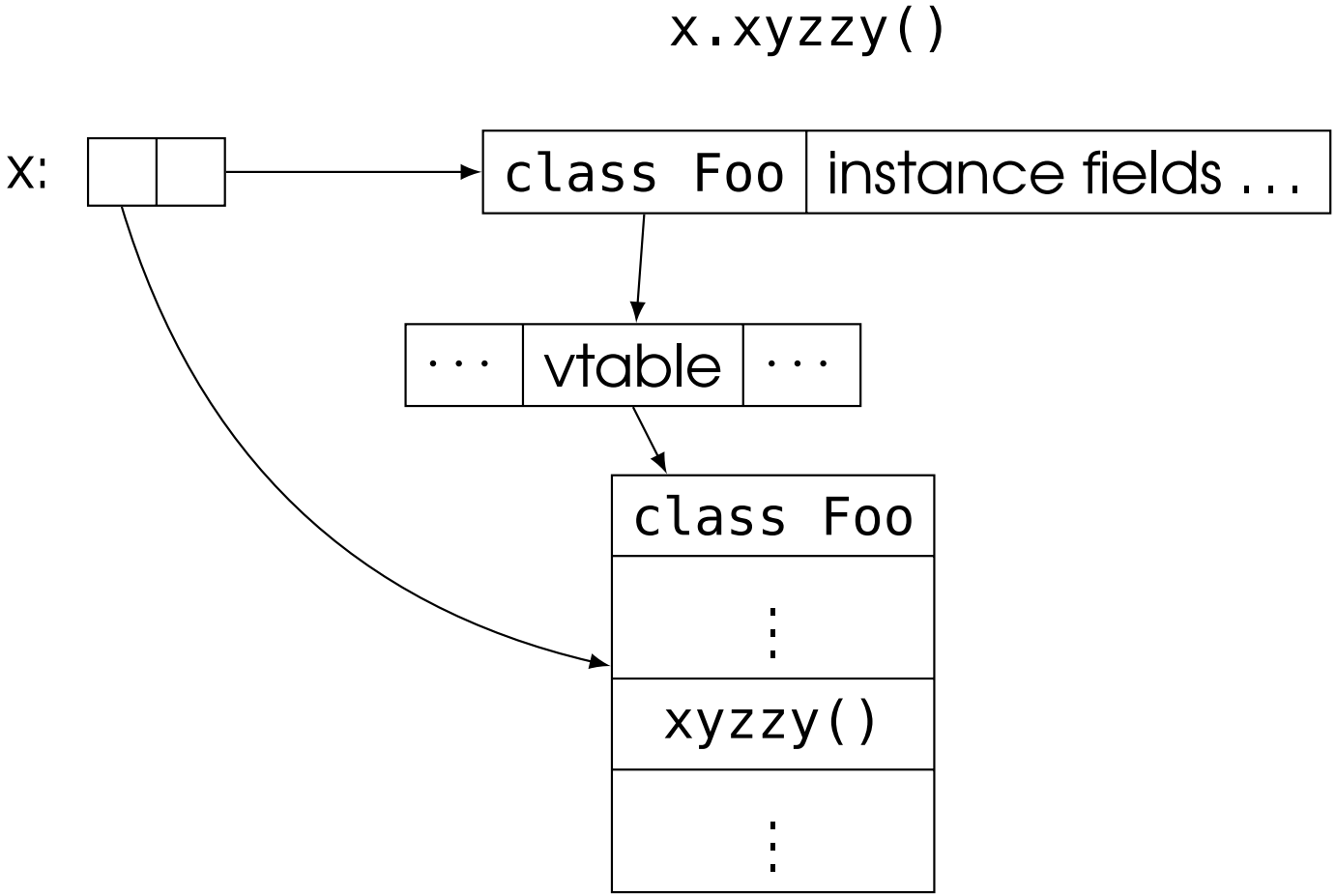
References



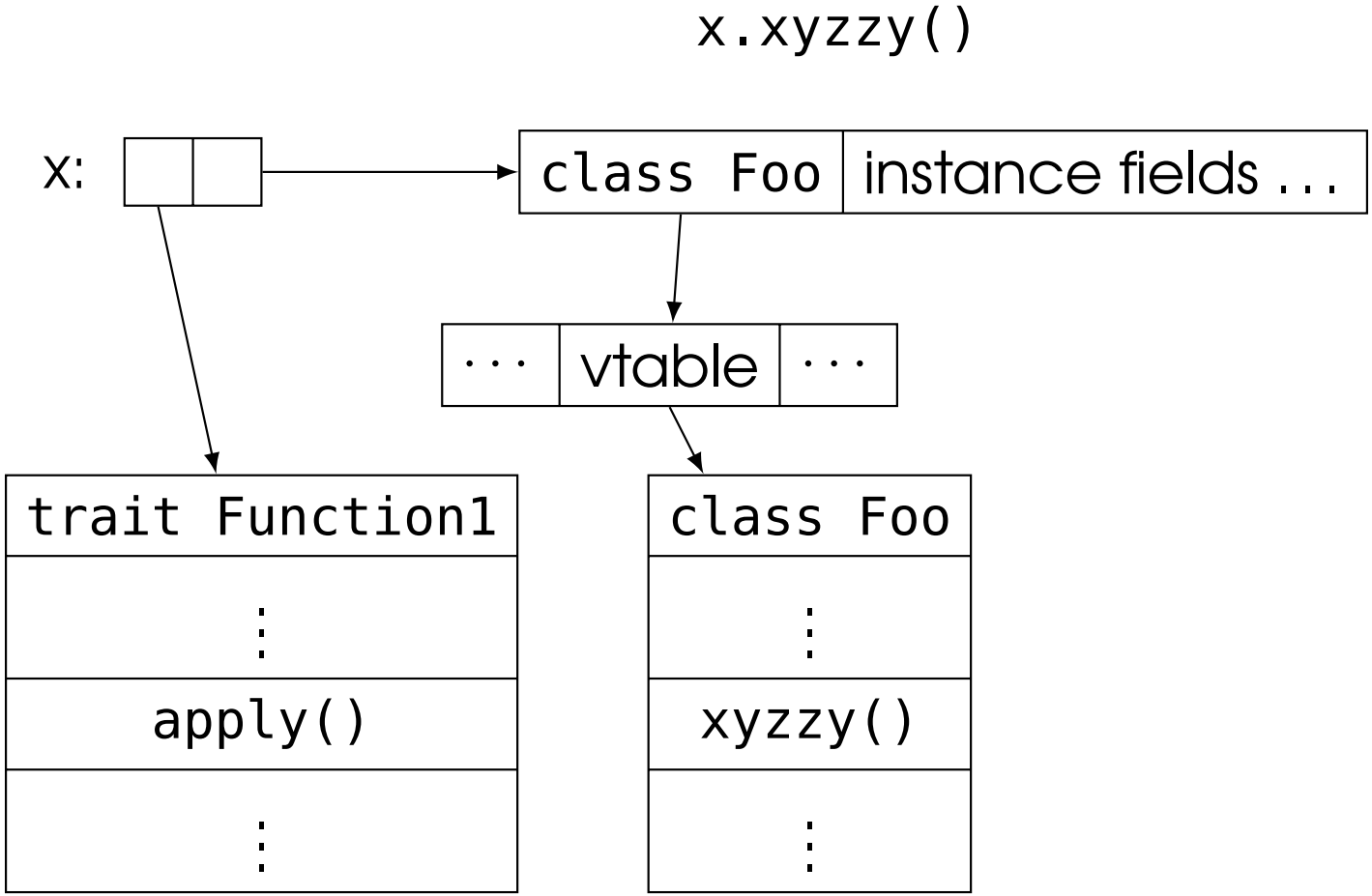
References



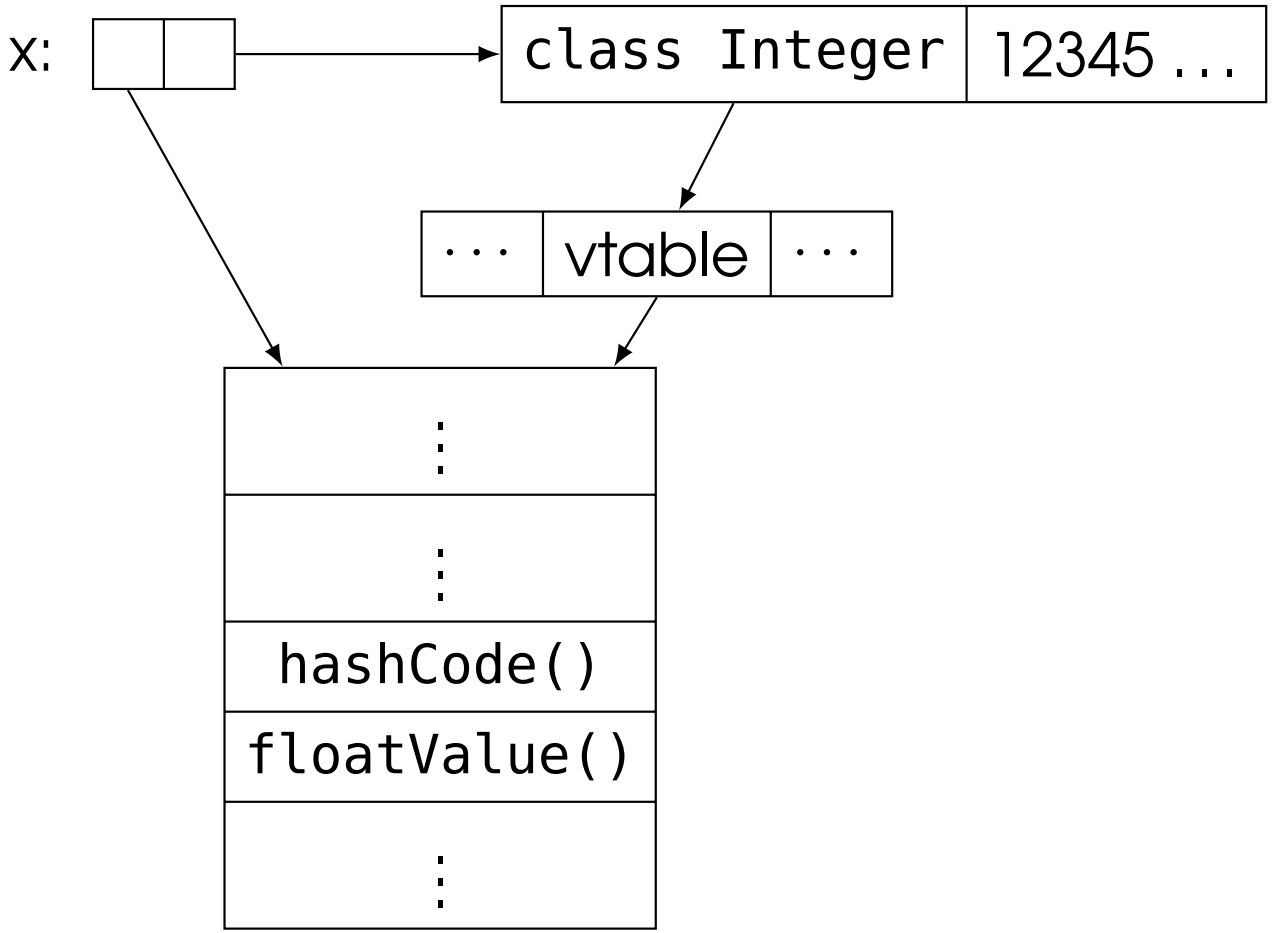
References



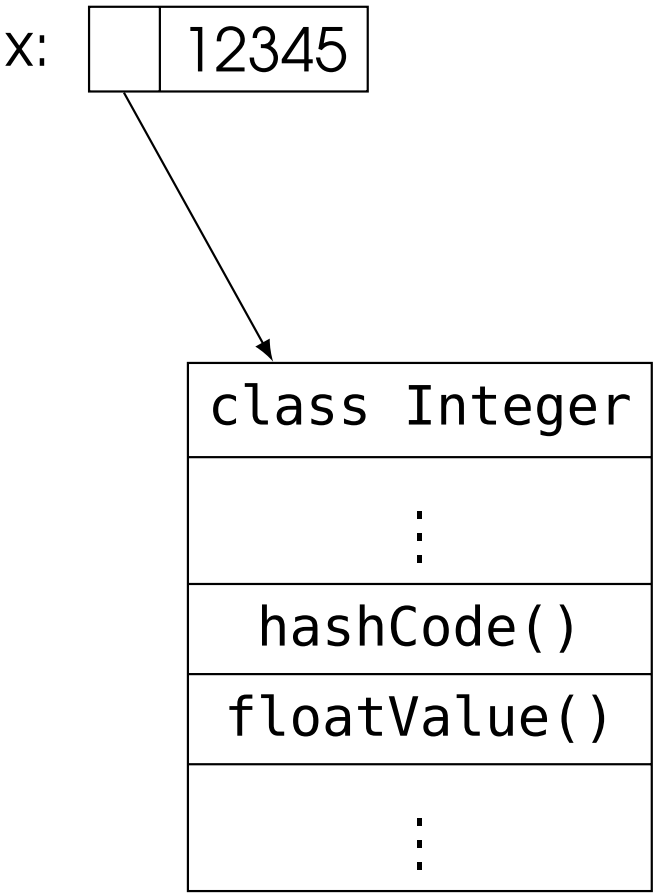
References


x.xyzzy()
x:
class Foo
instance fields ...
···
vtable
···
trait Function1
apply()
class Foo
xyzzy()

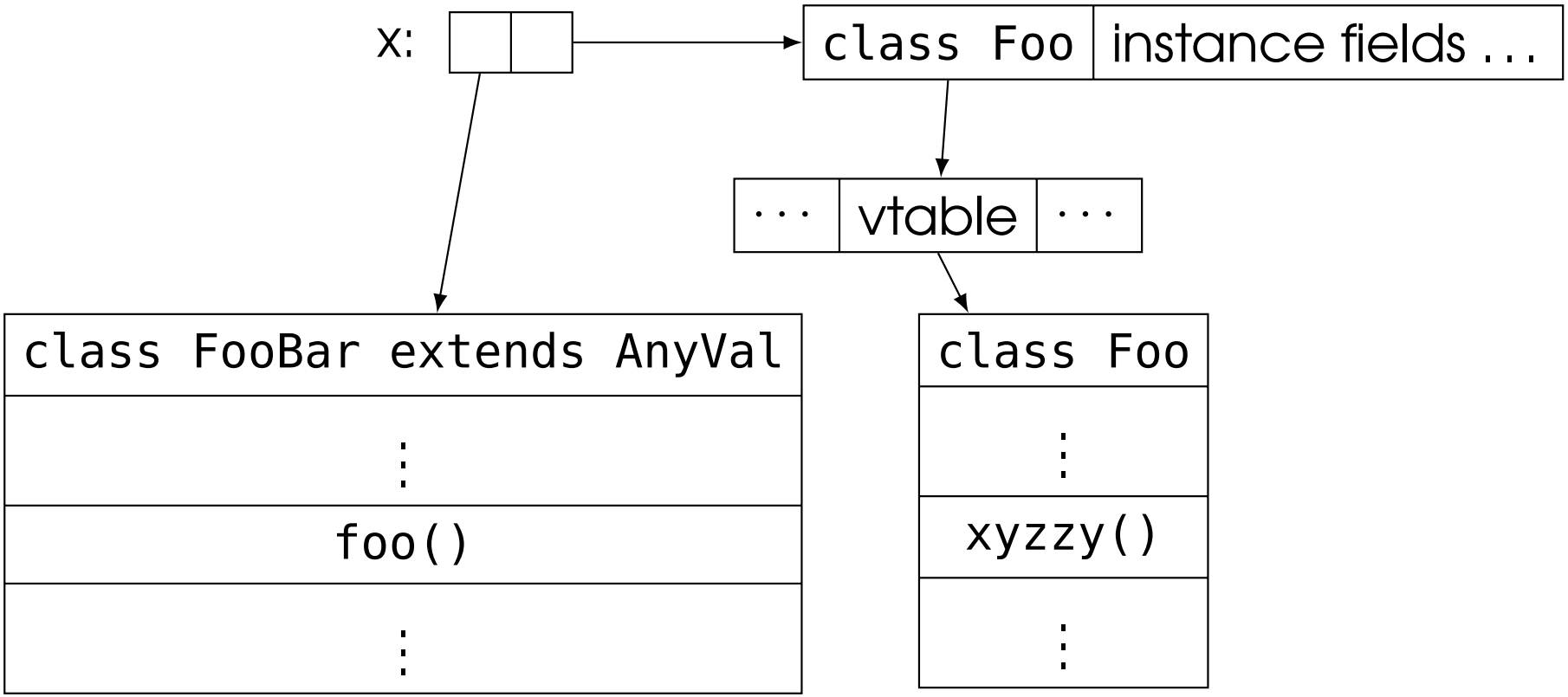
Disappearing Boxes



Disappearing Boxes



Disappearing Boxes – Value classes



Writing a (good) garbage collector is hard

Writing a (not-so-good) garbage collector is not as hard

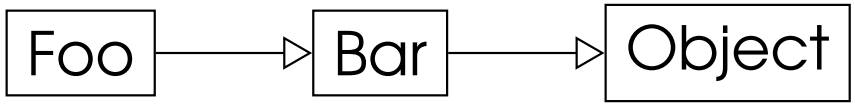
Memory Management – Mark Sweep

1. Add all roots to a worklist
2. Remove an object *o* from the worklist
3. Add all un-marked objects directly reachable from *o* to the worklist
4. Repeat 2 and 3 as long as there's something in the worklist
5. Sweep away (free) every un-marked object

Memory Management – Roots

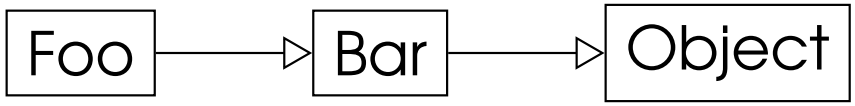
- Global objects – can be accessed by their name
- Local variables – can be accessed through the variable

Memory Management – Finding Pointers



Object.identity:Int
Bar.count:Int
Bar.parent:Bar
Bar.scale:Double
Foo.active:Boolean
Foo.name:String

Memory Management – Finding Pointers



Object.identity:Int
Bar.parent:Bar
Bar.count:Int
Bar.scale:Double
Foo.name:String
Foo.active:Boolean

Object	1/0
Bar	3/1
Foo	2/1

Foreign Function Interface

```
/* File I/O */

@foreign("apr_file_open_stdout")
def _apr_file_open_stdout(thefile:Ptr[Ptr[file_t]],
                          pool:Ptr[pool_t]): status_t = ???

@foreign("apr_file_putc")
def _apr_file_putc(ch:CChar, thefile:Ptr[file_t]): status_t = ???

object file_t {
  lazy val stdout = alloc.alloca { pptr: Ptr[Ptr[file_t]] =>
    new file_t(pptr.peek())
  }
}
class file_t(val self: Ptr[file_t]) {
  def putc(c: Byte): Unit {_apr_file_putc(c:CChar, self) }
}
```

Lightweight functions

LLVM has function pointers

$\Longrightarrow$ We don't need to build objects just to get something callable

Scala specific optimizations

LLVM can be extended with new analyses and optimizations

$\Longrightarrow$ Link time devirtualization! Whole program optimization!

Platform abstraction of Scala libraries

Much of Scala's library is tied to the JVM ☹

$\Longrightarrow$ Modularize the library; separate generic and implementation specific code