Objects & Closures

Matt Might

University of Utah

matt.might.net

Midterm 2: April 18

Today's agenda

- How to compile objects
- How to compile lambda

dangling Pointers

dangling

Continuation-Passing C

Compiling threads to events through continuations

Gabriel Kerneis · **Juliusz Chroboczek**

Received: date / Accepted: date

Abstract In this paper, we introduce Continuation Passing C (CPC), a programming language for concurrent systems in which native and cooperative threads are unified and presented to the programmer as a single abstraction. The CPC compiler uses a compilation technique, based on the CPS transform, that yields efficient code and an extremely lightweight representation for contexts. We provide a complete proof of the correctness of our compilation scheme. We show in particular that lambda-lifting, a common compilation technique for functional languages, is also correct in an imperative language like C, under some conditions enforced by the CPC compiler. The current CPC compiler is mature enough to write substantial programs such as Hekate, a highly concurrent BitTorrent seeder. Our benchmark results show that CPC is as efficient, while significantly cheaper, as the most efficient thread libraries available.

Continuation-Passing C

Compiling threads to events through continuations

Gabriel Kerneis · **Juliusz Chroboczek**

Received: date / Accepted: date

Abstract In this paper, we introduce Continuation Passing C (CPC), a programming language for concurrent systems in which native and cooperative threads are unified and presented to the programmer as a single abstraction. The CPC compiler uses a compilation technique, based on the CPS transform, that yields efficient code and an extremely lightweight representation for contexts. We provide a complete proof of the correctness of our compilation scheme. We show in particular that lambda-lifting, a common compilation technique for functional languages, is also correct in an imperative language like C, under some conditions enforced by the CPC compiler. The current CPC compiler is mature enough to write substantial programs such as Heka[illegible], a highly concurrent BitTorrent seeder. Our benchmark results show that CPC is as efficient, [illegible] significantly cheaper, as the most efficient thread libraries available.

goto as call/ec?

lab_1: $block_1$

...

lab_2: $block_2$

...

```
(letrec
 ([lab_i (λ ()
           ((call/ec (λ (goto)
                       block_i
                       ...
                       (goto lab_{i+1})))))])
 (lab_1))
```

C++ and `finally`

```
fh = fopen(fname,“rw”)
try:
  doSomething(fh)
finally:
  fclose(fh)
```

```
class File_handle {
   FILE* p;
public:
   File_handle(const char* n, const char* a)
      { p = fopen(n,a); if (p==0) throw Open_error(errno); }
   File_handle(FILE* pp)
      { p = pp; if (p==0) throw Open_error(errno); }

   ~File_handle() { fclose(p); }
   operator FILE*() { return p; }

   // ...
};

void f(const char* fn)
{
   File_handle f(fn,"rw"); // open fn for reading and writing
   // use file through f
}
```

```
class Foo {
  public:
  Foo  () { printf("Foo created\n")   ; }
  ~Foo () { printf("Foo destroyed\n") ; }
} ;
```

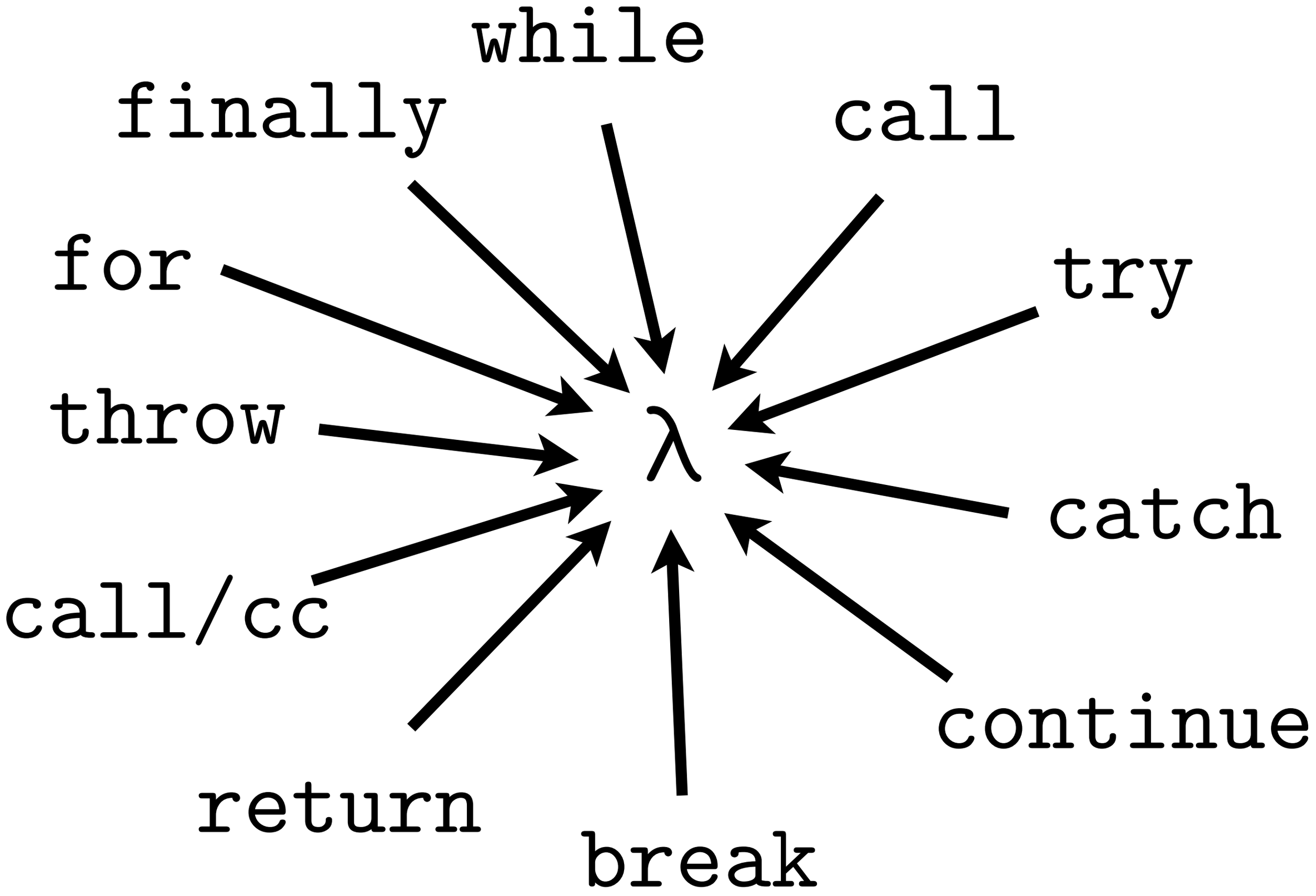
```
void f() {
  Foo x ;
  printf("I'm in f().\n") ;
}
```

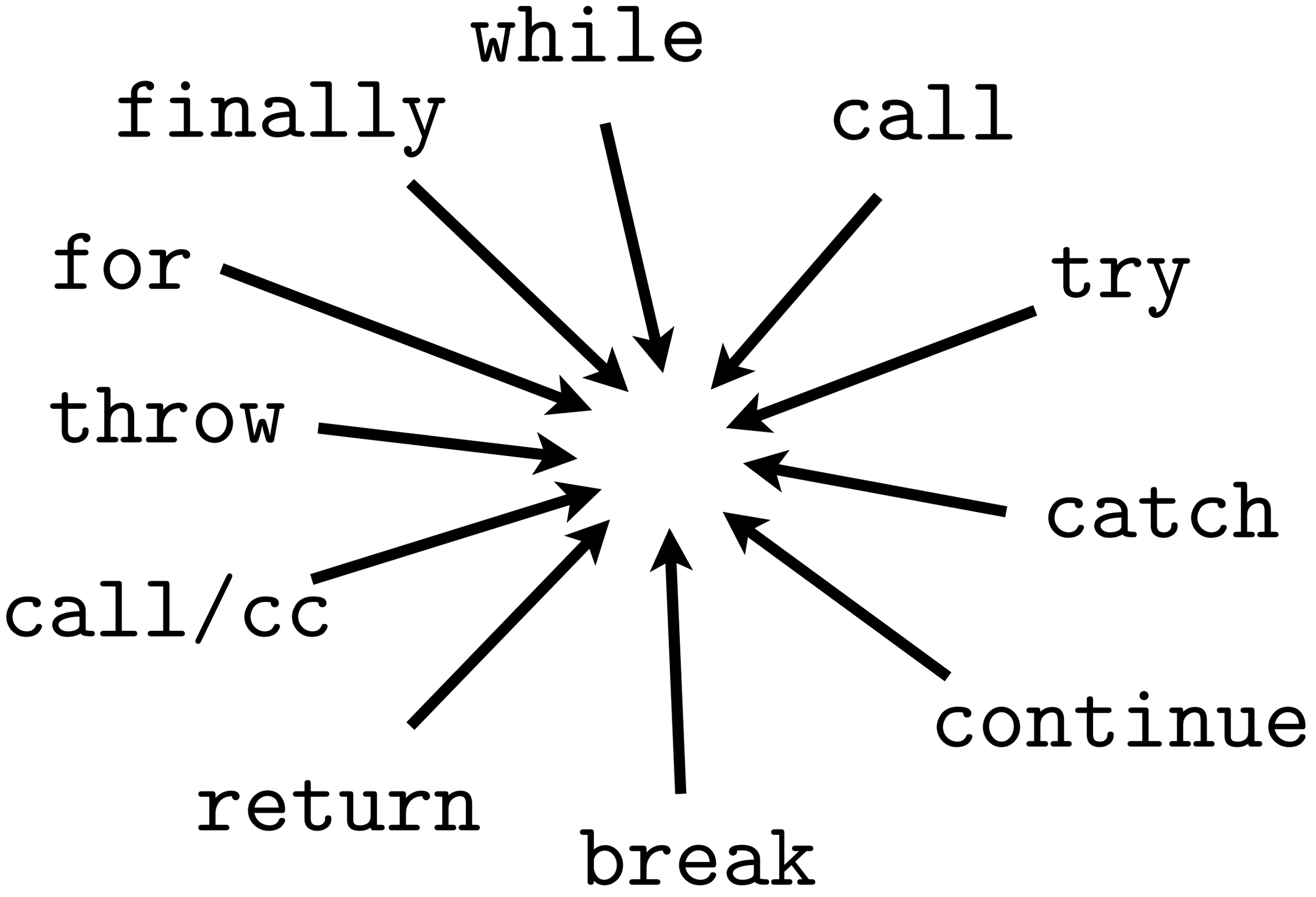
```
class Foo {
  public:
  Foo  () { printf("Foo created\n")   ; }
  ~Foo () { printf("Foo destroyed\n") ; }
} ;
```

```
void g() { throw "I'm an exception." ; }

void f() {
  Foo x ;
  printf("I'm in f().\n") ;
  g() ;
}
```

```
int f() {
  printf ("I'm starting f().\n") ;
  while (true) {
    Foo x ;
    printf("I'm in the loop.\n") ;
    return 10 ;
  }
  printf ("I'm ending f().\n") ;
}
```

λ

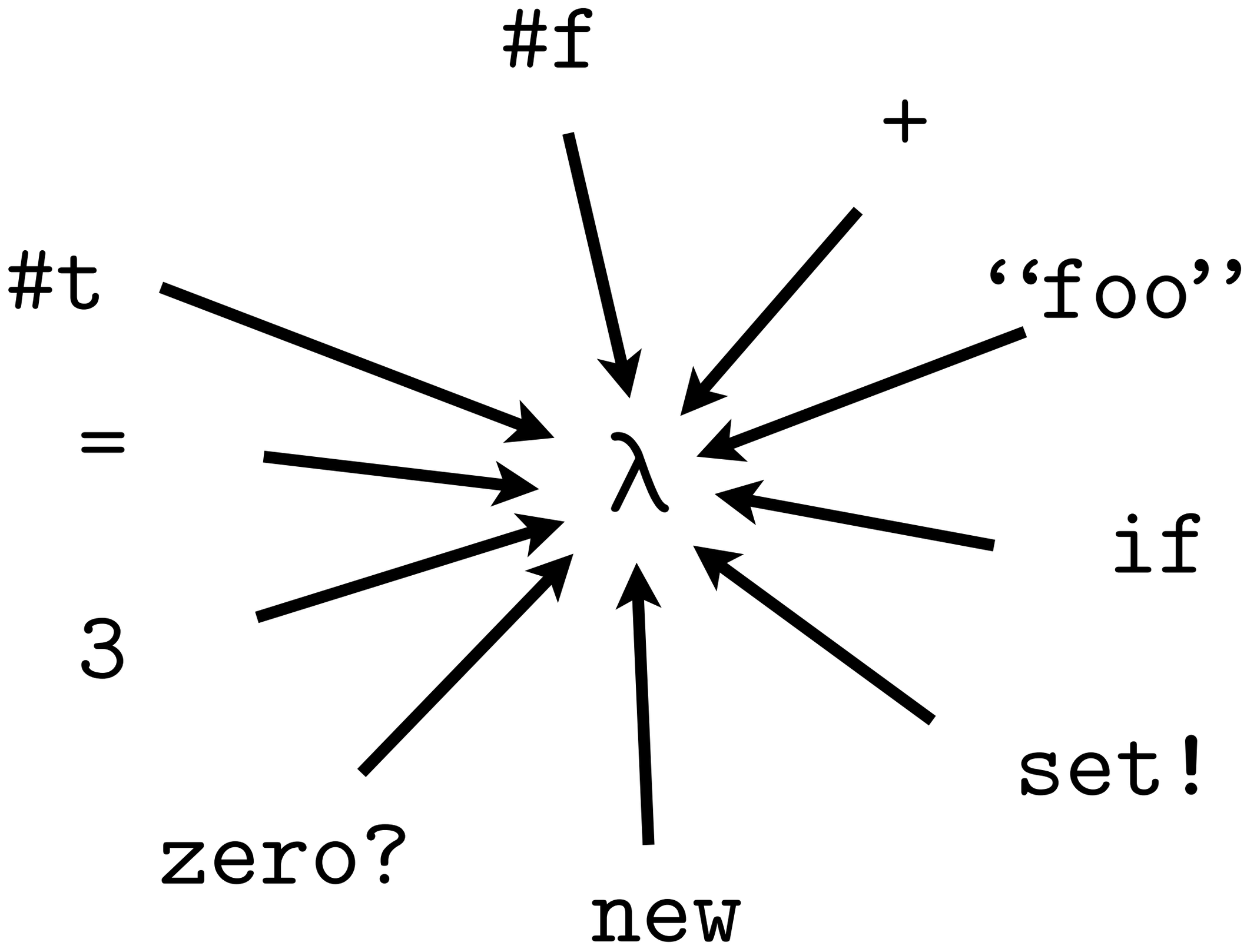

while
finally
call
for
try
throw
λ
catch
call/cc
continue
return
break

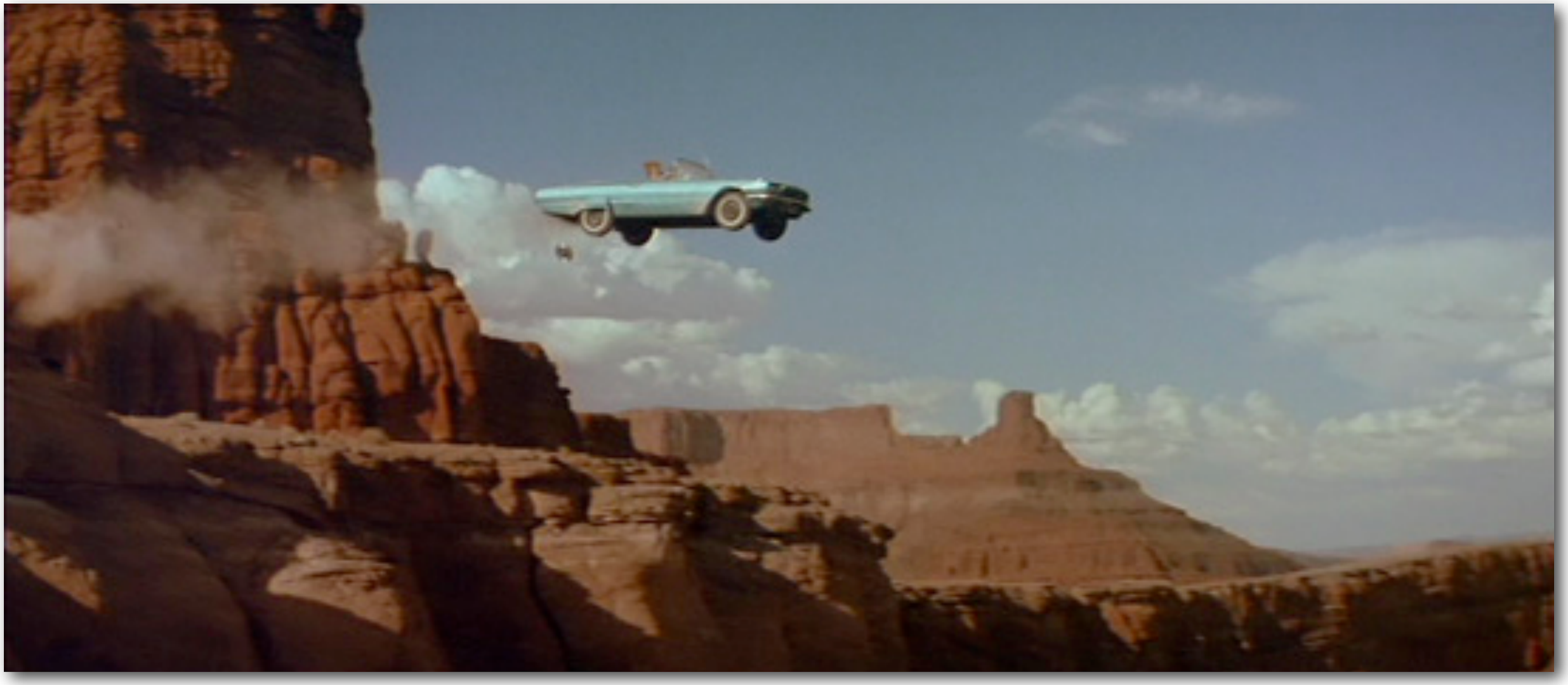

while
finally
call
for
try
throw
catch
call/cc
continue
return
break

What's left of Python?

- Basic values
- Primitives
- Conditionals
- Mutation
- Classes/objects
- Lambda terms

λ


#f
+
#t
“foo”
=
λ
3
if
set!
zero?
new



Inefficient.

Ridiculous.

Problem: `new` and λ

```
Foo* foo = new Foo() ;
foo->a = 3 ;
foo->b = 4 ;
```

```
new     Foo
movq    %rax, -8(%rbp)
movq    -8(%rbp), %rax
movl    $3, (%rax)
movq    -8(%rbp), %rax
movl    $4, 4(%rax)
movl    $0, %eax
```

```
o->methodCall() ;
```

```
call o,methodCall
```

```
(set-then! f (λ (x) (+ a x)))
(f 3))
```

```
movl    λ addl a, %eax
          return
        _f
movl    $3, %edi
movl    $0, %eax
call    _f
```

How to kill `new` and λ

λ: Closure conversion.

Turn `new` into λ?

We could, but...

```
o->methodCall() ;
```

Structs & Procedures

Object elimination

What *is* an object?

(*class*, *struct*)

What's a class?

It depends.

```
o->methodCall()
```

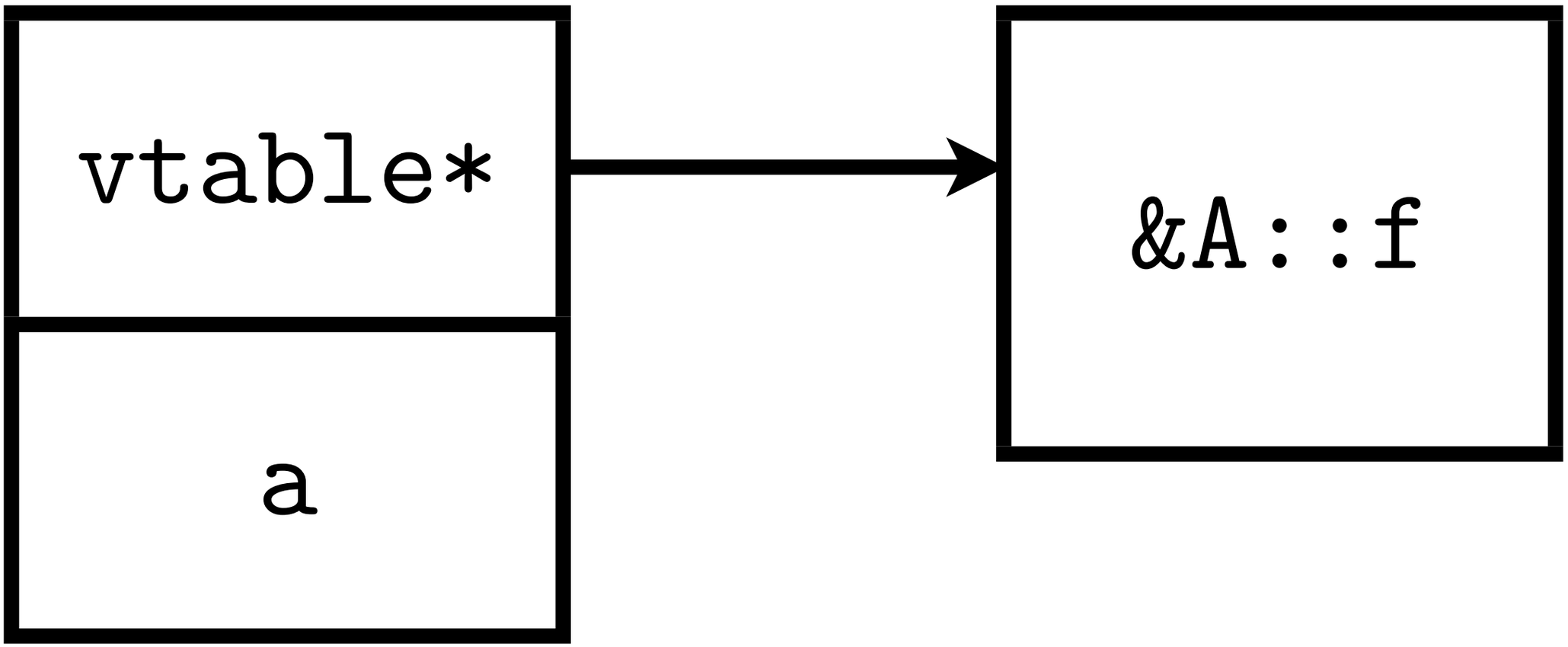
```
o.methodCall()
```

Dispatch strategies

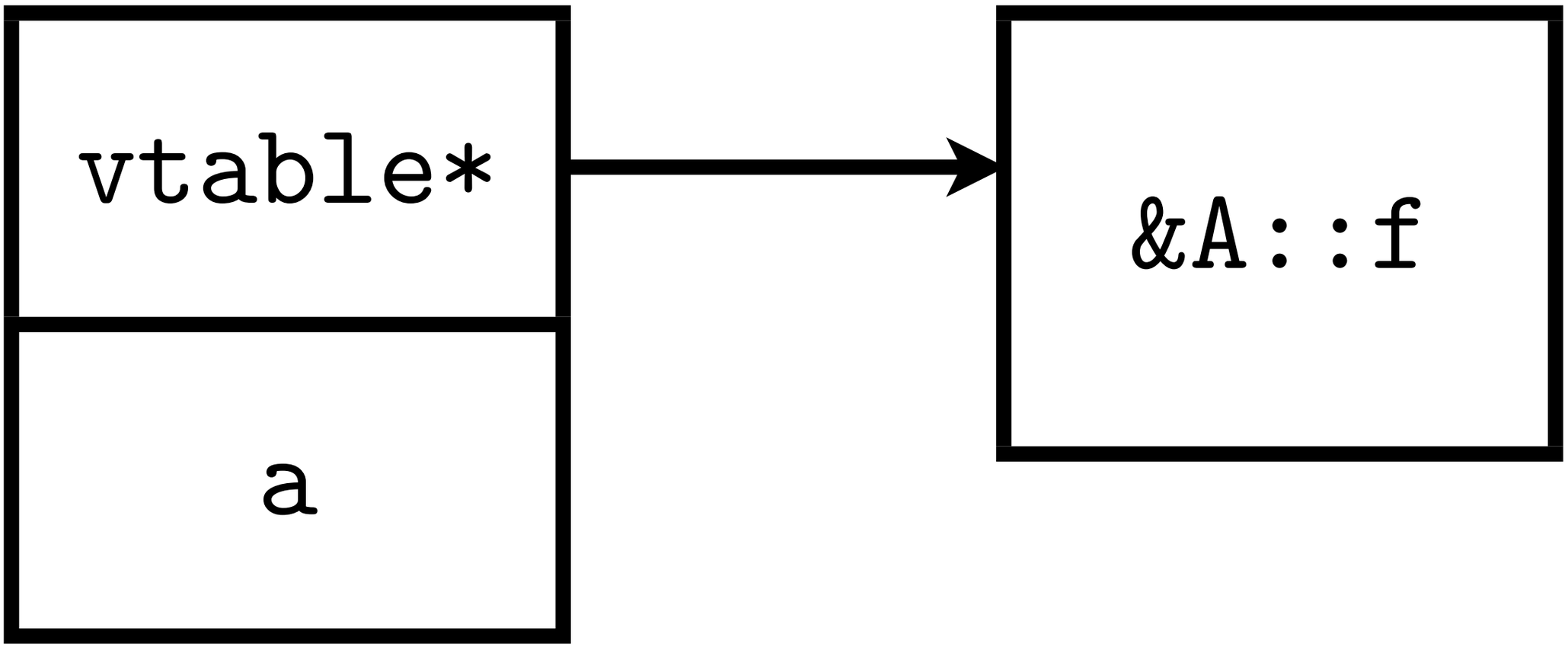
- Per-object hash table
- Per-class hash table
- Virtual method table

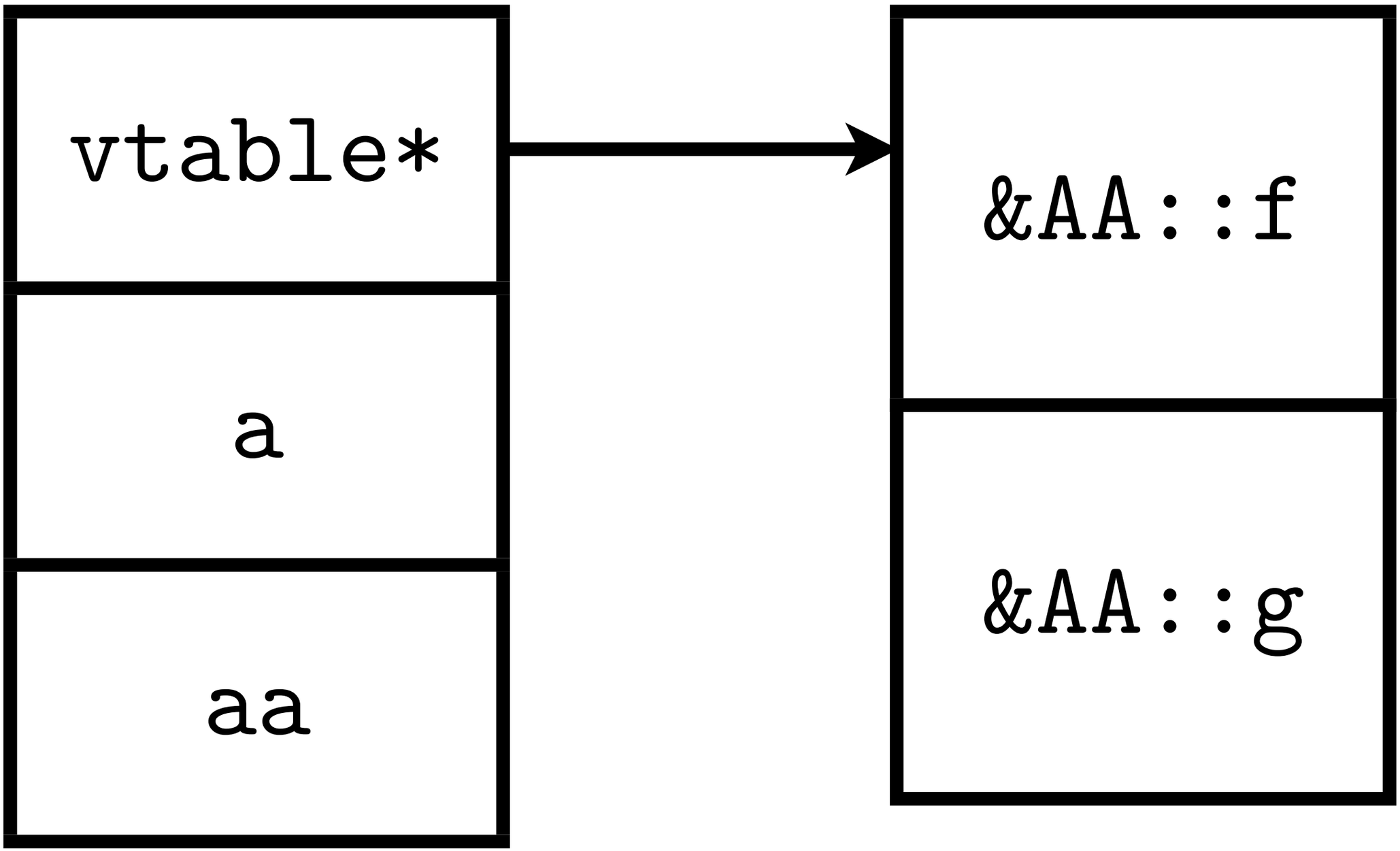
```
class A(B1,...Bn):
  ...

a = A()
a.f()
```

```
class A {
 public:
 virtual int f() ;
  int a ;
} ;
```


vtable*
a
&A::f

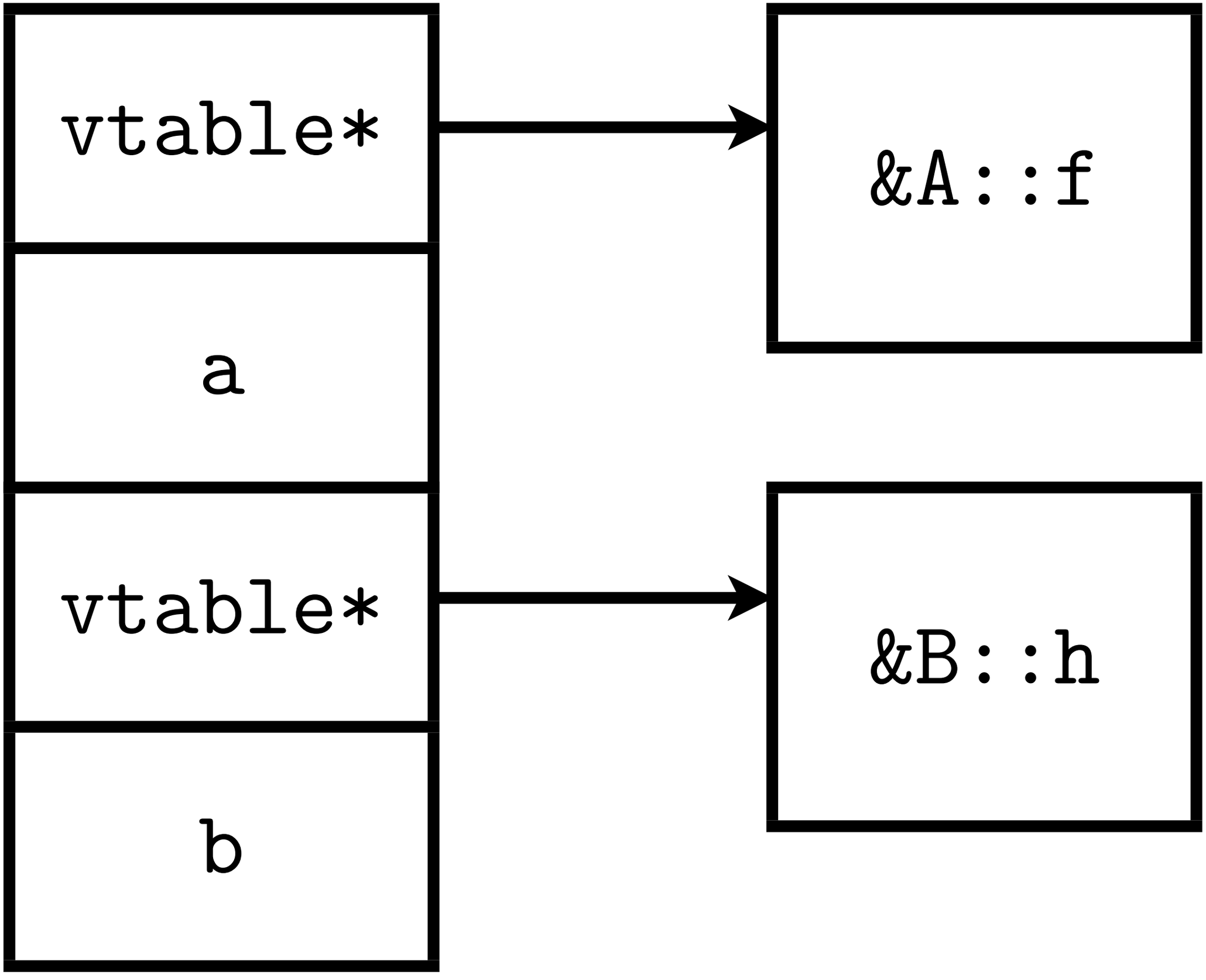
```
class AA : public A {
 public:
 virtual int g() ;
  virtual int f() ;
  int aa ;
} ;
```


vtable*
a
&A::f


vtable*
a
aa
&AA::f
&AA::g

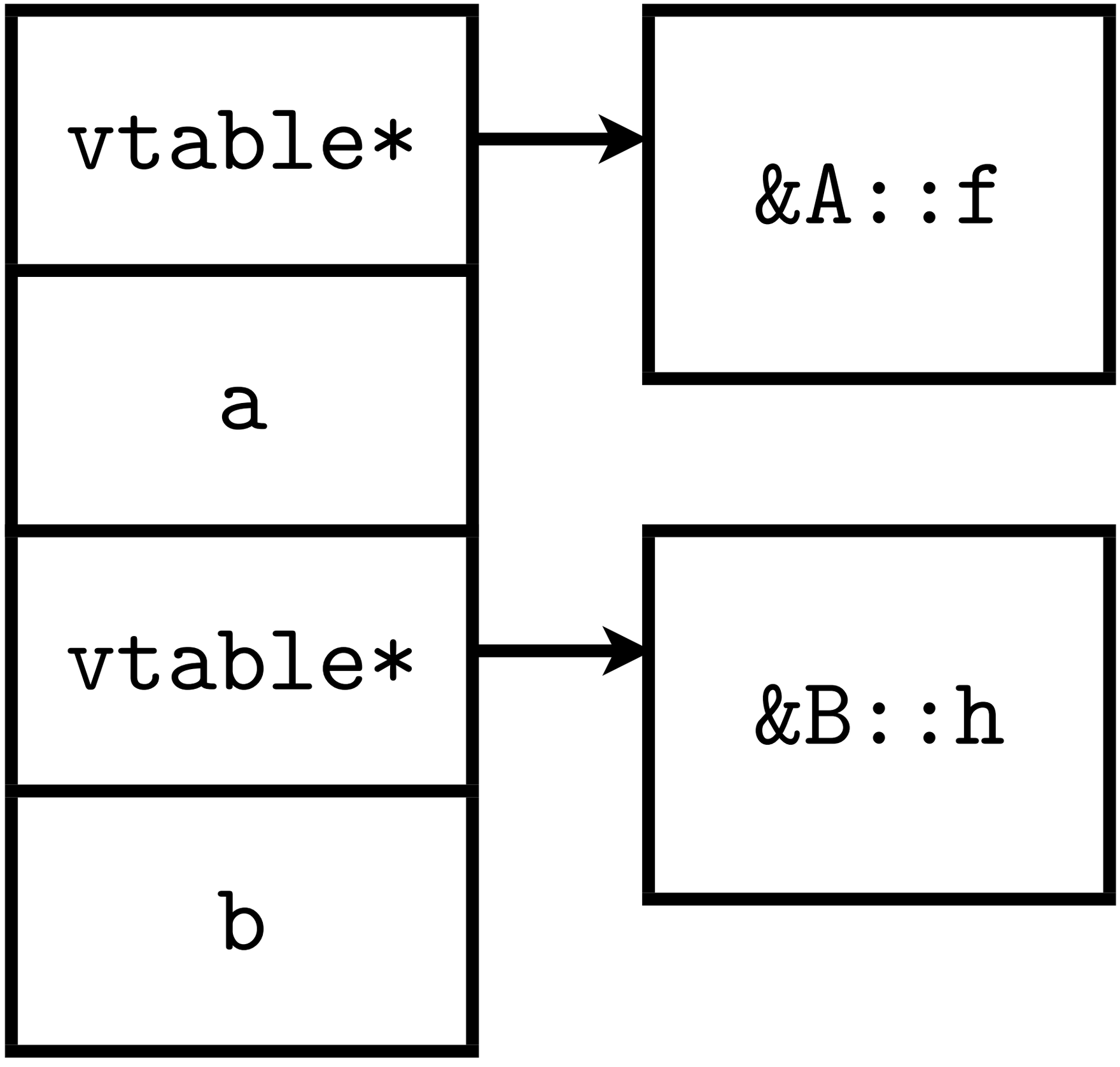
```
class B {
 public:
 virtual int h() ;
  int b ;
} ;
```

```
class AB :
 public A,
  public B {
} ;
```


vtable*
a
vtable*
b
&A::f
&B::h

```
AB* ab = new AB() ;
A*   a  = ab ;
B*   b  = ab ;
//   a == b ???
```

a ab



b

```
ab = static_cast<AB*>(b);
```

```
ab = (AB*)(void*)(b);
```

Closure conversion

The issue?

```
lambda z: \
    lambda x: z + x
```

f

```
typedef int (*g_t)(int) ;
g_t f(int z) {
  return g ;
}
```

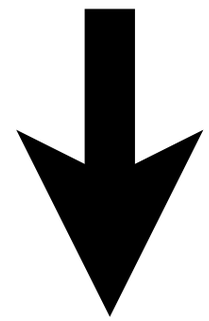
```
int g(int x) {
  return x + z ;
}

typedef int (*g_t)(int) ;

g_t f(int z) {
  return g ;
}
```

```
int g(int x) {
  return x + z;
}

typedef int (*g_t)(int) ;

g_t f(int z) {
  return g ;
}
```

```
f = lambda z: \
        lambda x: z + x

g = f(3)
```

```
(lambda x: x + z,[z => 3])
```

```
(lambda x: x + z,[z => 3])
```



```
(lambda ρ, x: x + ρ.z),
[z => 3])
```

```
lambda args:
```

$$\ldots\ fv_i\ \ldots\ldots$$

$$\texttt{lambda}_n\ \texttt{a}\mathit{rgs}\texttt{:}$$

$$\ldots\ \mathit{fv}_i\ \ldots\ldots$$

```
lambda_n ρ,args:
```

$$\ldots\ fv_i\ \ldots\ldots$$

$$\texttt{lambda}_n\ \rho\texttt{,a}\mathit{rgs}\texttt{:}$$

$$\ldots\ \rho \cdot \mathit{fv}_i\ \ldots$$

```
(lambda_n ρ,args:
  ... ρ.fv_i ...,
  Envn(fv_1,...,fv_m))
```

```
class Envn:
 def __init__(self,fv1,...):
  ...
  self.fvi = fvi
  ...
```

$$f(e_1, \ldots, e_n)$$

$$
\begin{aligned}
t &= f \\
g &= t[0] \\
\rho &= t[1] \\
g(\rho, e_1, \ldots, e_n)
\end{aligned}
$$

```
g = h()[0]
ρ = h()[1]
g(ρ, e_1, ..., e_n)
```

Language whose *objects* need *closure*-conversion?

```
class J {
    int a ;
    Object o ;
    public J () {
      o = new Object() {
        int b() {
          return a ;
        }
      } ;
    }
}
```

Questions?

```
(λ (args)
  . . .  fv_i  . . .)
```

$$(\lambda_n\ (args)\ \ldots\ fv_i\ \ldots)$$

$$(\lambda_n\ (\rho\ args)\ \ldots\ fv_i\ \ldots)$$

```
(λ_n (ρ args)
     ... (get ρ fv_i) ...)
```

```
(make-closure
   (λ_n (ρ args)
     ... (get-n ρ fv_i) ...)
(env-n fv_1 ...)))
```

```
struct clo_n {
 val_t (*lam)(env_n*,val_t,...) ;
 struct env_n* env ;
} ;
```

```
struct env_n {
  val_t fv1 ;
    ...
  val_t fvm ;
} ;
```

```
union clo_t {
  struct {
    val_t (*lam)() ;
    void* env ;
  } any ;
  struct clo_i clo_i ; ...
} ;
```