Compiler Construction
Lent Term 2015
Lectures 10, 11 (of 16)

1. Slang.2 (Lecture 10)
   1. In lecture code walk of slang2_derive
2. Assorted topics (Lecture 11)
   1. Exceptions
   2. Objects
   3. Stacks vs. Register
   4. Simple optimisations
   5. Boxed and unboxed objects

Timothy G. Griffin
tgg22@cam.ac.uk
Computer Laboratory
University of Cambridge

Topic 1 : Exceptions (informal description)

`e handle f`

If expression e evaluates "normally" to value v, then v is the result of the entire expression.

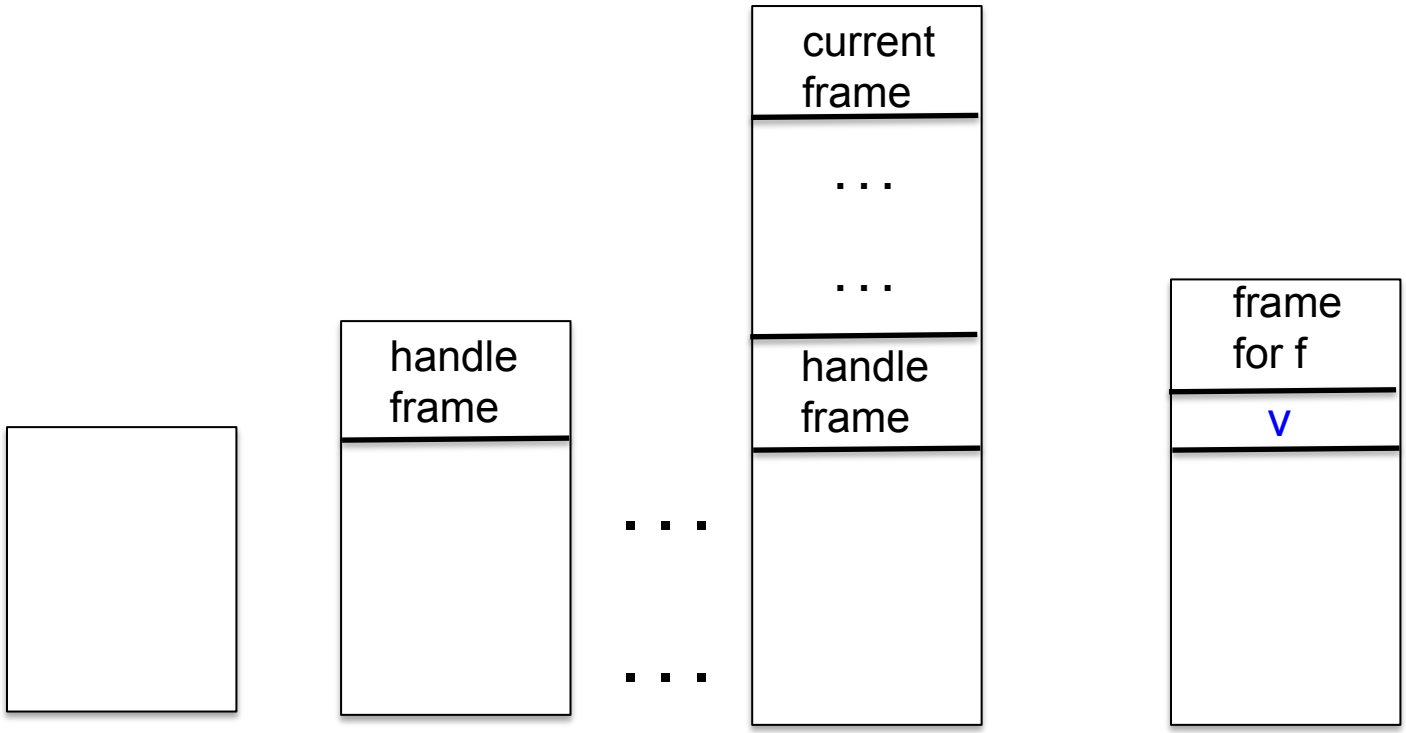
Otherwise, an exceptional value v' is "raised" in the evaluation of e, then result is (f v')

`raise e`

Evaluate expression e to value v, and then raise v as an exceptional value, which can only be "handled".

Implementation of exceptions may require a lot of language-specific consideration and care. Exceptions can interact in powerful and unexpected ways with other language features. Think of C++ and class destructors, for example.

Viewed from the call stack



Call stack just before evaluating code for

`e handle f`

Push a special frame for the handle

"`raise v`" is encountered while evaluating a function body associated with top-most frame

"Unwind" call stack. Depending on language, this may involve some "clean up" to free resources.

Possible pseudo-code implementation

```
e handle f
```

```
let fun _h27 () =
  build special “handle frame”
  save address of f in frame;
  … code for e …
  return value of e
in _h27 () end
```

```
raise e
```

```
… code for e …
save v, the value of e;
unwind stack until first
fp found pointing at a handle frame;
Replace handle frame with frame
for call to (extracted) f using
v as argument.
```

Topic 2 : Objects (with single inheritance)

```
let start := 10

   class Vehicle extends Object {
      var position := start
      method move(int x) = {position := position + x}
   }
   class Car extends Vehicle {
      var passengers := 0
      method await(v : Vehicle) =
         if (v.position < position)
         then v.move(position - v.position)
         else self.move(10)
   }
   class Truck extends Vehicle {
      method move(int x) =
         if x <= 55 then position := position +x
   }
   var t := new Truck
   var c := new Car
   var v : Vehicle := c
in
   c.passengers := 2;
   c.move(60);
   v.move(70);
   c.await(t)
end
```

method override

subtyping allows a Truck or Car to be viewed and used as a Vehicle

Object Implementation?

- how do we access object fields?
  - both inherited fields and fields for the current object?
- how do we access method code?
  - if the current class does not define a particular method, where do we go to get the inherited method code?
  - how do we handle method override?
- How do we implement subtyping ("object polymorphism")?
  - If B is derived from A, then need to be able to treat a pointer to a B-object as if it were an A-object.

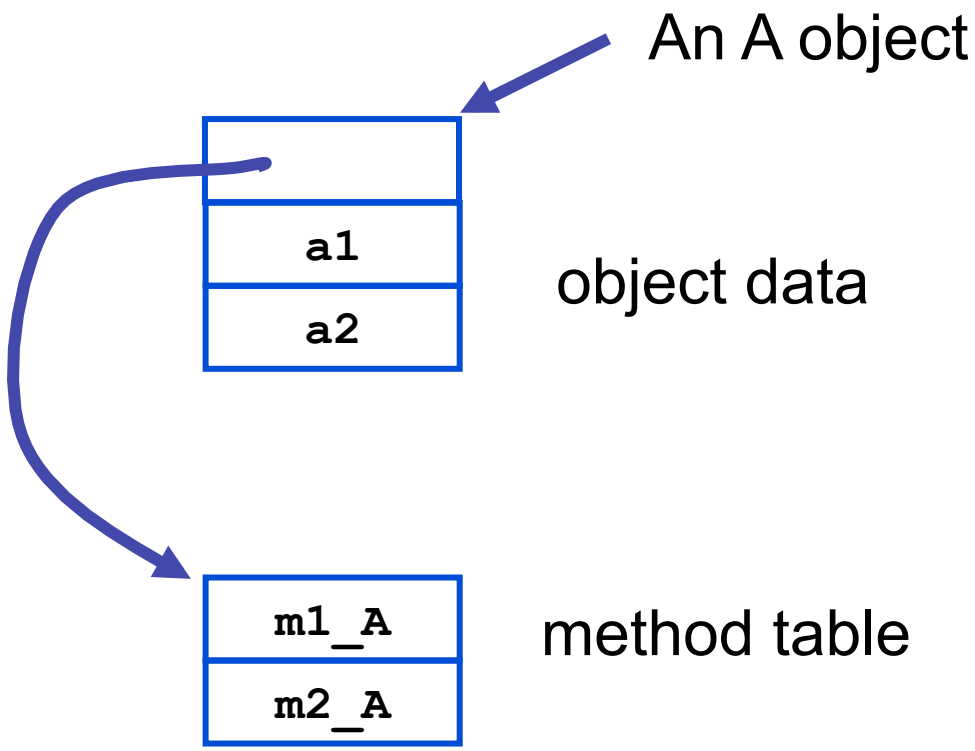
Another OO Feature

- Protection mechanisms
  - to encapsulate local state within an object, Java has “private” “protected” and “public” qualifiers
    - private methods/fields can’t be called/used outside of the class in which they are defined
  - This is really a scope/visibility issue! Front-end during semantic analysis (type checking and so on), the compiler maintains this information in the symbol table for each class and enforces visibility rules.

Object representation

C++

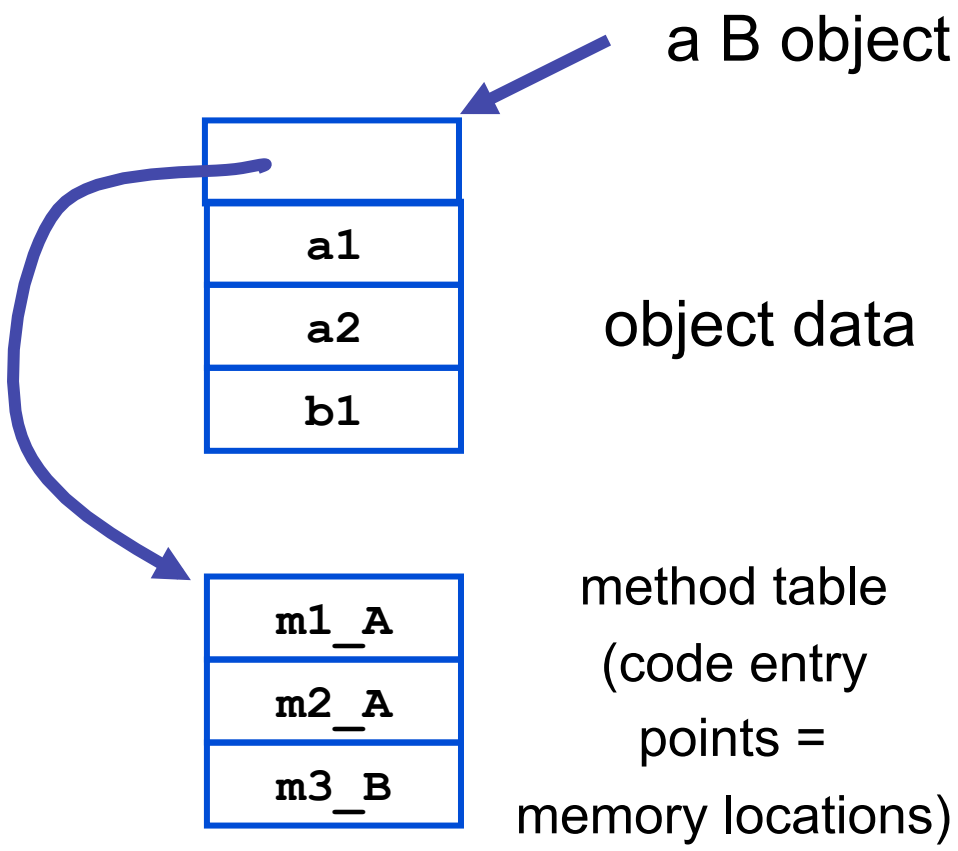
```
class A {
public:
    int a1, a2;

    void m1(int i) {
        a1 = i;
    }
    void m2(int i) {
        a2 = a1 + i;
    }
}
```



NB: a compiler typically generates methods with an extra argument representing the object (self) and used to access object data.

Inheritance ("pointer polymorphism")

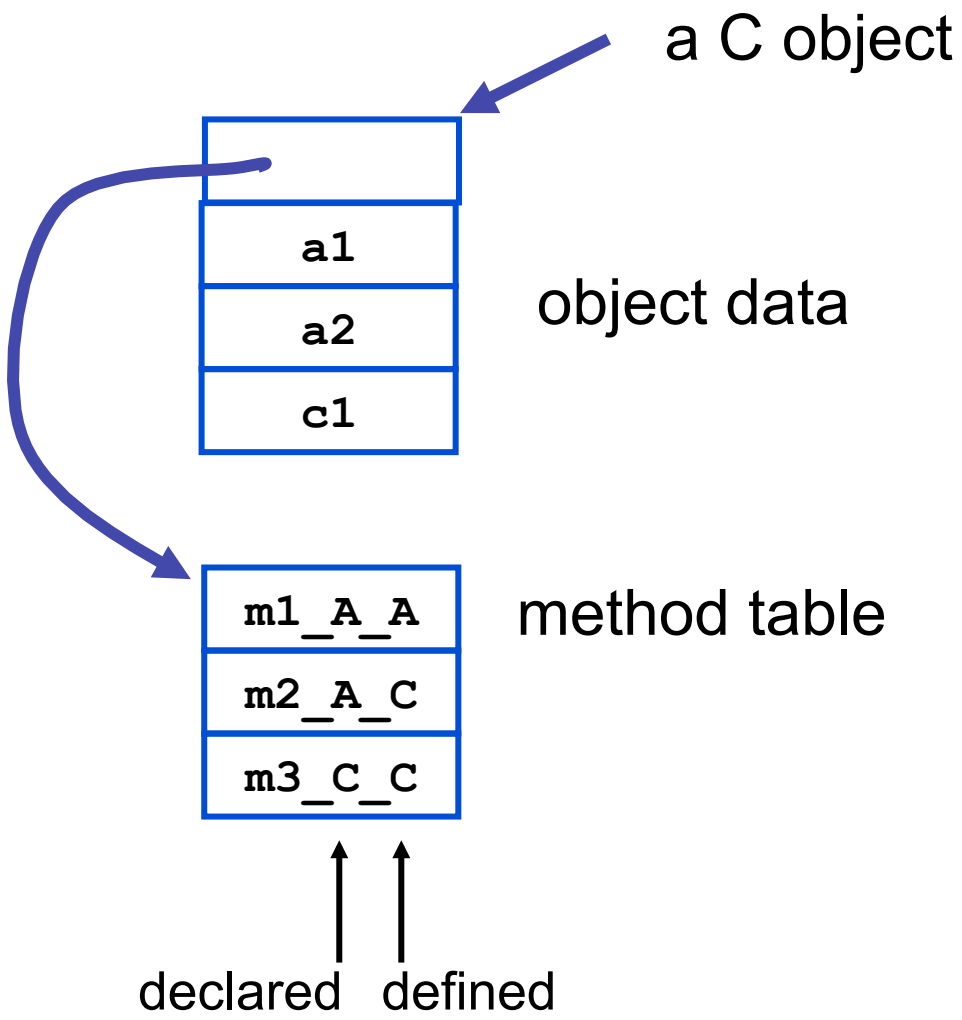
```
class B : public A {
public:
    int b1;

    void m3(void) {
        b1 = a1 + a2;
    }
}
```



Note that a pointer to a B object can be treated as if it were a pointer to an A object!

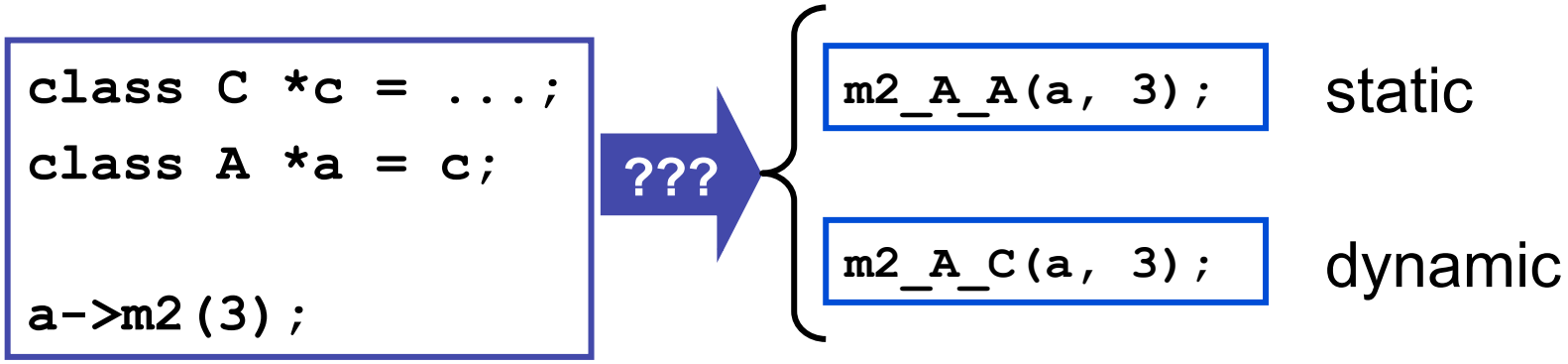
Method overriding

```
class C : public A {
public:
    int c1;

    void m3(void) {
        b1 = a1 + a2;
    }
    void m2(int i) {
        a2 = c1 + i;
    }
}
```



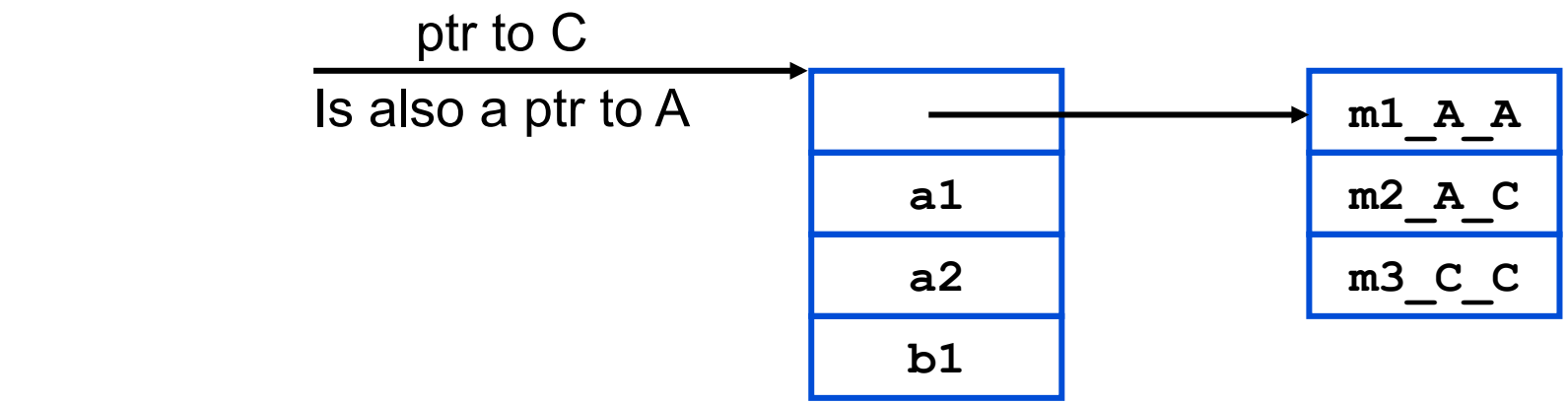
Static vs. Dynamic

- which method to invoke on overloaded polymorphic types?



Dynamic dispatch

- implementation: dispatch tables

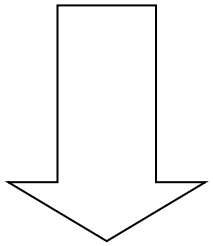



```
class C *c = ...;
class A *a = c;

a->m2(3);
```

```
*(a->dispatch_table[1])(a, 3);
```

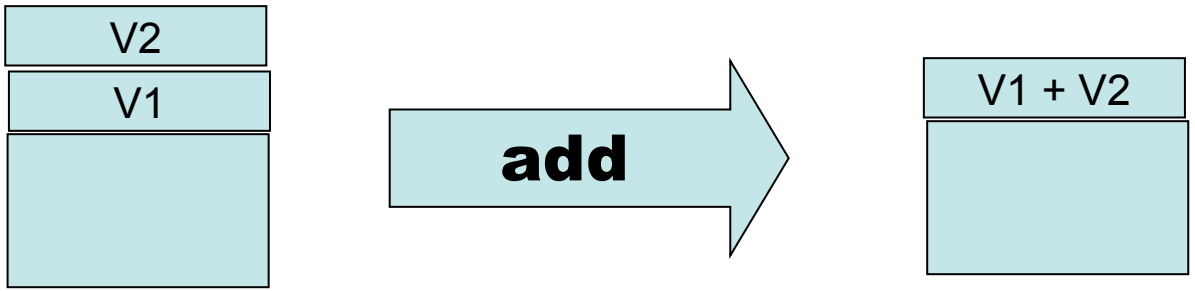
This implicitly uses some form of pointer subtyping

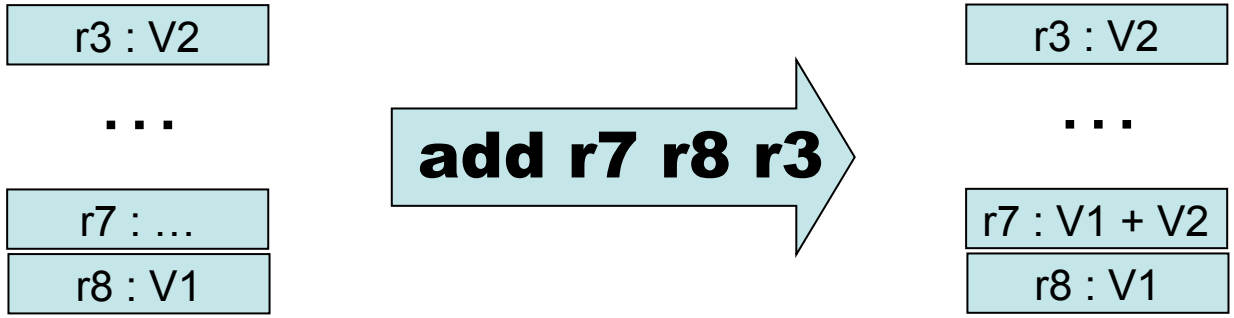
```
void m2(int i) {
        a2 = c1 + i;
}
```



```
void m2_A_C(class_A *this_A, int i) {
   class_C *this = convert_ptrA_to_ptrC(this_A);

   this->a2 = this->c1 + i;
}
```

Topic 3 : stack vs regsisters





Stack-oriented:
(+) argument locations is implicit, so instructions are smaller.
(-) Execution is slower

Register-oriented:
(+) Execution faster
(-) argument location is explicit, so instructions are larger

Topic 4: Simple optimisations. (a) Inline expansion

```
fun f(x) = x + 1
fun g(x) = x - 1
…
…
fun h(x) = f(x) + g(x)
```

inline f and g

```
fun f(x) = x + 1
fun g(x) = x - 1
…
…
fun h(x) = (x+1) + (x-1)
```

(+) Avoid building activation records at runtime

(+) May allow further optimisations

(-) May lead to "code bloat" (apply only to functions with "small" bodies?)

Question: if we inline all occurrences of a function, can we delete its definition from the code?
What if it is needed at link time?

Be careful with variable scope

Inline g in h

```
let val x = 1
    fun g(y) = x + y
    fun h(x) = g(x) + 1
in
   h(17)
end
```

NO

```
let val x = 1
    fun g(y) = x + y
    fun h(x) = x + y + 1
in
   h(17)
end
```

YES

```
let val x = 1
    fun g(y) = x + y
    fun h(z) = x + z + 1
in
   h(17)
end
```

(b) Constant propagation, constant folding

```
let x = 2
let y = x - 1
let z = y * 17
```

```
let x = 2
let y = 2 - 1
let z = y * 17
```

```
let x = 2
let y = 1
let z = y * 17
```

```
let x = 2
let y = 1
let z = 1 * 17
```

```
let x = 2
let y = 1
let z = 17
```

Propagate constants and evaluate simple expressions at compile-time

Note : opportunities are often exposed by inline expansion!

David Gries :
"Never put off till run-time what you can do at compile-time."

But be careful

How about this?

Replace

x * 0

with

0

OOPS, not if x has type float!

NAN*0 = NAN,

(c) peephole optimisation

Peephole Optimization

W. M. McKeeman
Stanford University, Stanford, California

Communications of the ACM, July 1965

Example 1. Source code:

$$X := Y;$$
$$Z := X + Z$$

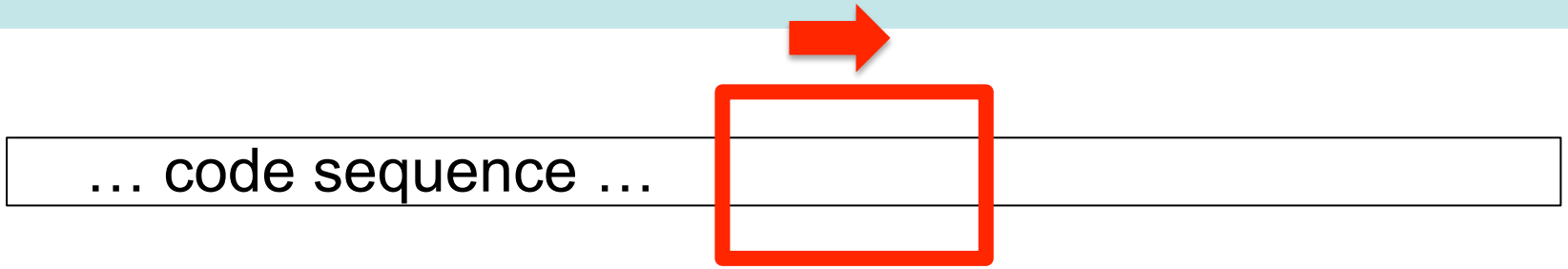
Compiled code:

```
LDA Y   load the accumulator from Y
STA X   store the accumulator in X
LDA X   load the accumulator from X
ADD Z   add the contents of Z
STA Z   store the accumulator in Z
```

Eliminate!

Results for syntax-directed code generation.

peephole optimisation

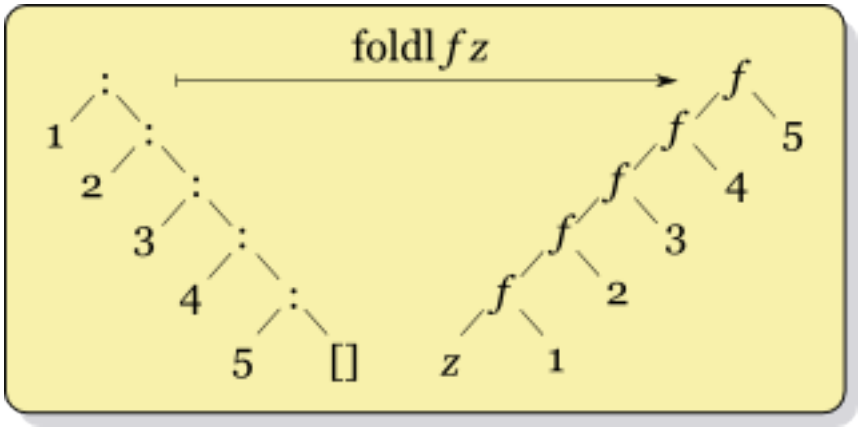


Sweep a window over the code sequence looking for instances of simple code patterns that can be rewritten to better code … (might be combined with constant folding, etc, and employ multiple passes)

Examples
-- eliminate useless combinations (push 0; pop)
-- introduce machine-specific instructions
-- improve control flow (rewrite “GOTO L1 … L1: GOTO L2” to “GOTO L2 … L1 : GOTO L2”)

(d) Eliminate Tail recursion

A recursive function exhibits tail recursion if on all recursive branches the last thing it does is call itself.




```
fun foldl f e []       = e
  | foldl f e (x::xr) = foldl f (f(x, e)) xr
```

We should be able to compile this to a LOOP in order to avoid constructing many activation records at runtime.
Exercise : How?

Topic 5 : Boxed and unboxed objects

```
map : ('a -> 'b) -> 'a list -> 'b list

fun map f [] = []
  | map f (a::rest) = (f a) :: (map f rest)
```

The code generated for `map` must work for any times `'a` and `'b`.

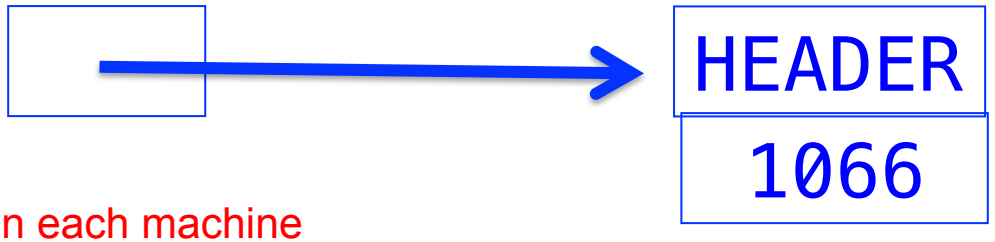
So it seems that all values of any type must be represented by objects of the same size.

Boxing and Unboxing

An unboxed integer : 1066

On the heap

A boxed integer :



Many ML compilers use a single bit in each machine word to distinguish boxed from unboxed values. This is why mosml has 31 (or 63) bit integers.

It is better to work with unboxed values than with boxed values.

Compilers for ML-like languages must expend a good deal of effort trying to find good optimizations for boxed/unboxed choices.

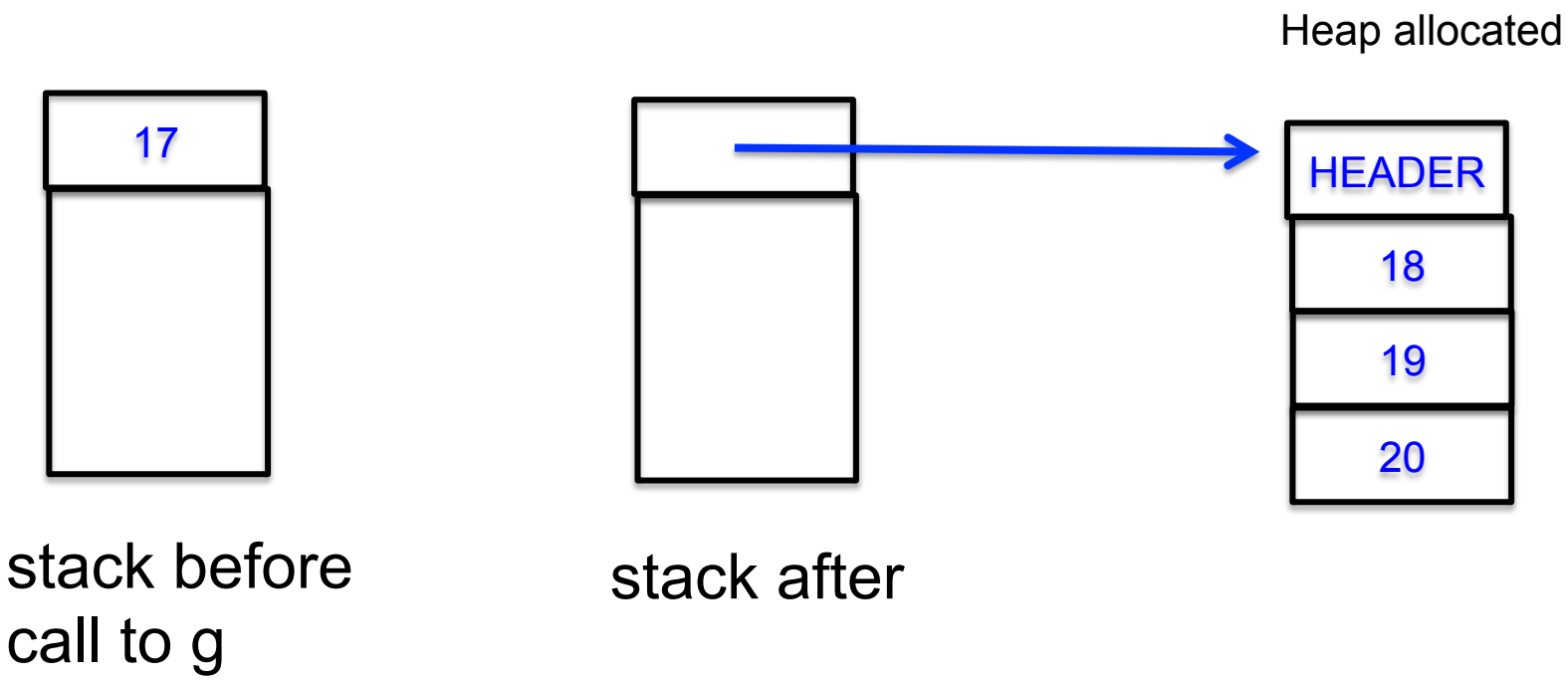
See Appel.

Similar terminology is used in Java for putting a value in a container class (boxing) and taking it out (unboxing)

For example, put an int into the Integer container class.

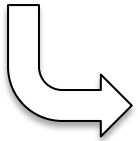
Tuples (in ML-like, L3-like languages)

```
g: int -> int * int * int

fun g x = (x+1, x+2, x+3)

   . . . (g 17) . . .
```



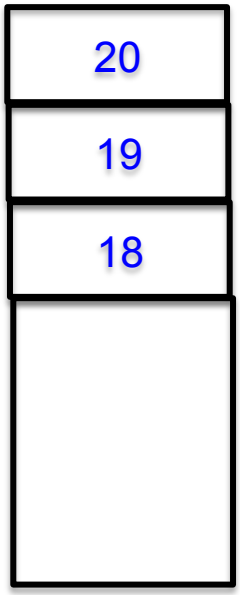
On a stack-oriented machine

```
fun g x = (x+1, x+1, x+3)
```

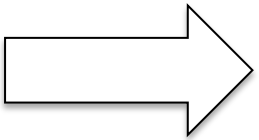


Some IR

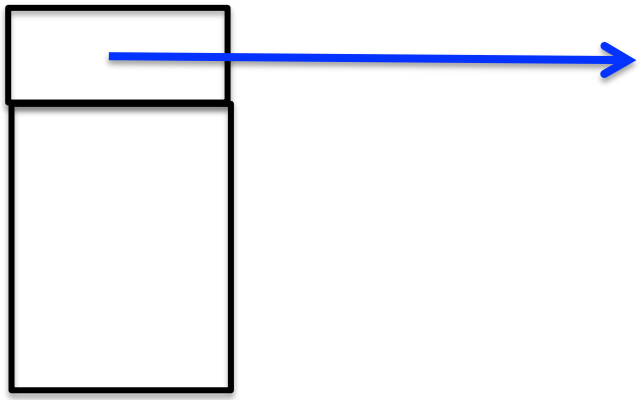
```
fun g x =
    let val y1 = x+1
        val y2 = x+2
        val y3 = x+3
    in return (ALLOCATE_TUPLE 3) end
```



ALLOCATE_TUPLE 3



Heap allocated

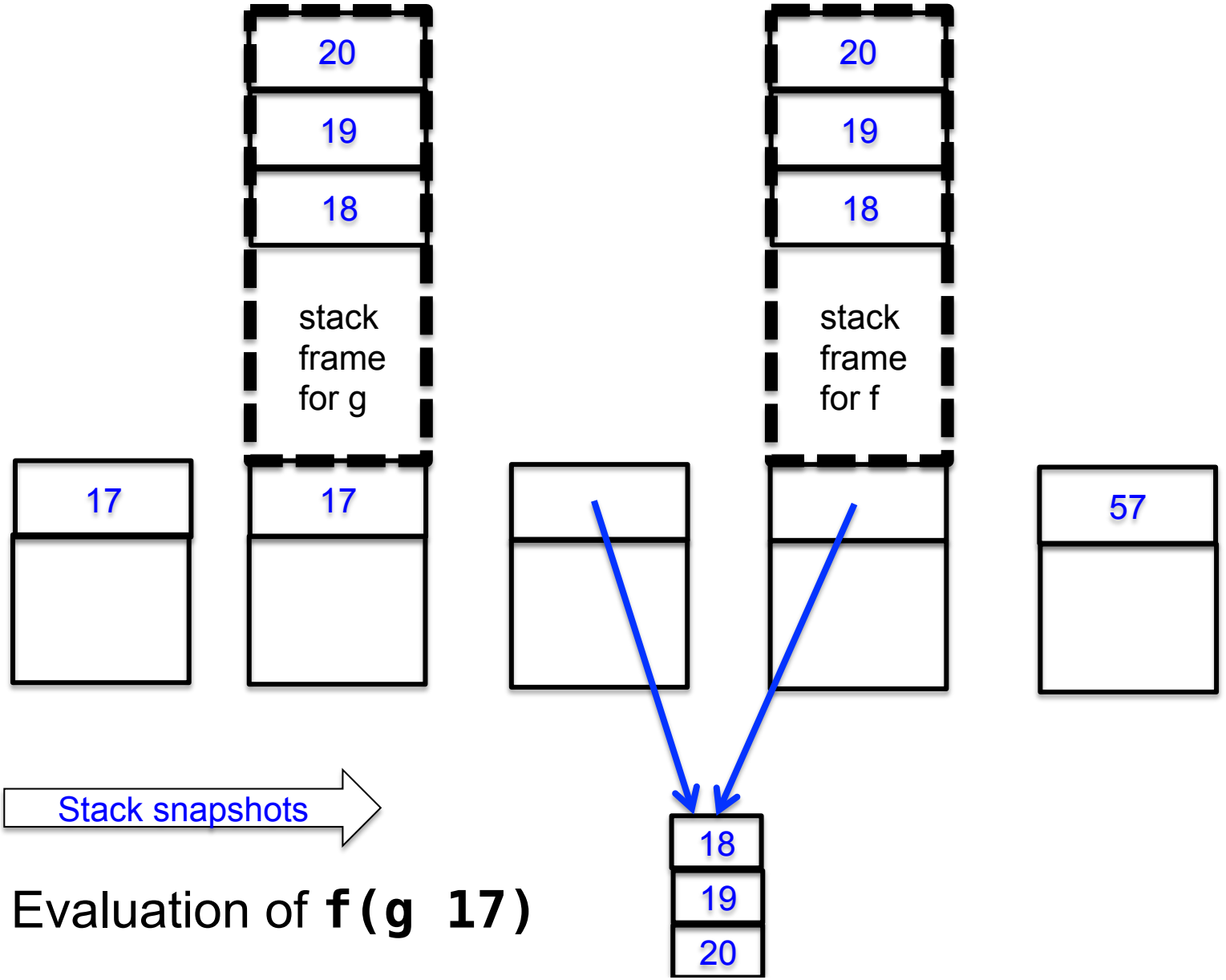


Tuples (in ML-like, L3-like languages)

```
fun g x = (x+1, x+1, x+3)

fun f (u, v, w) = u + v + w

   . . . f (g 17) . . .
```

- Does function f take 3 arguments or 1?
- How would you inline f?

How might we avoid this?



Evaluation of **`f(g 17)`**