Dirty details behind **”high level”** programming languages

Aurélien Naldi

IJM – 2020/02/13

- High level language ??
  - Hides <u>bare-metal details</u>
- Advantages
  - Development speed
  - Improved correctness and safety
- Hidden costs
  - Extra memory / computing

<u>Coarse overview, with shortcuts and weird jargon</u>

interruptions welcome!

Surprising differences in speed

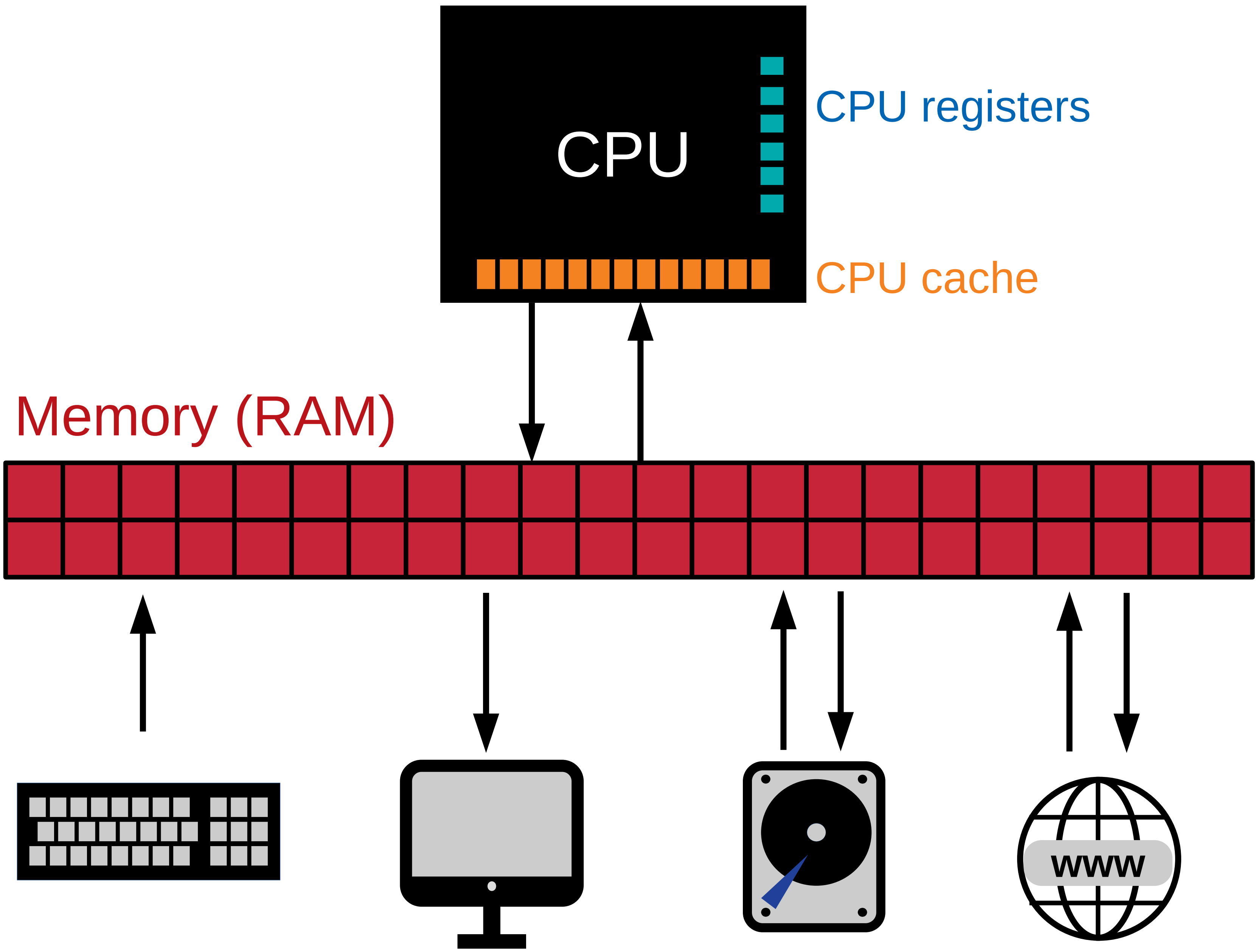
C	NumPy loop	NumPy

`c = int[n]`
`for (int i=0 ; i<n ; i++) {`
`    c[i] = a[i] + b[i]`
`}`
	`c =  a.copy()`
	`for i in range(n):`
	`    c[i] += b[i]`
		`c = a + b`

32 ms	2450 ms	27 ms

Time for n = 10 000 000

Simplified computer architecture



Timescales in computing operations

Operation	Time	Human scale
CPU Cycle	0.5 ns	1 s
CPU cache	5 ns	10 s
Memory	100 ns	4 min
SSD storage	25 – 150 μs	1 – 4 days
Hard drive	1 – 10 ms	1 – 10 months
Internet	50 – 200 ms	4 – 15 years

Source: www.prowesscorp.com/computer-latency-at-a-human-scale

Capacity of memory layers

Type	Size	Speed
Registers	1ko	
CPU cache L1	128ko	700 Go/s
CPU cache L2	1Mo	200Go/s
CPU cache L3	6Mo	100Go/s
CPU cache L4	128Mo	40Go/s
RAM	32Go	10Go/s
Hard drive	2To	< 2Go/s

https://en.wikipedia.org/wiki/Memory_hierarchy

Some CPU instructions speeds

Bitwise: OR, AND, SHIFT, … (1 cycle)

Integer: ADD (1), MUL (3-5), DIV (10-50)

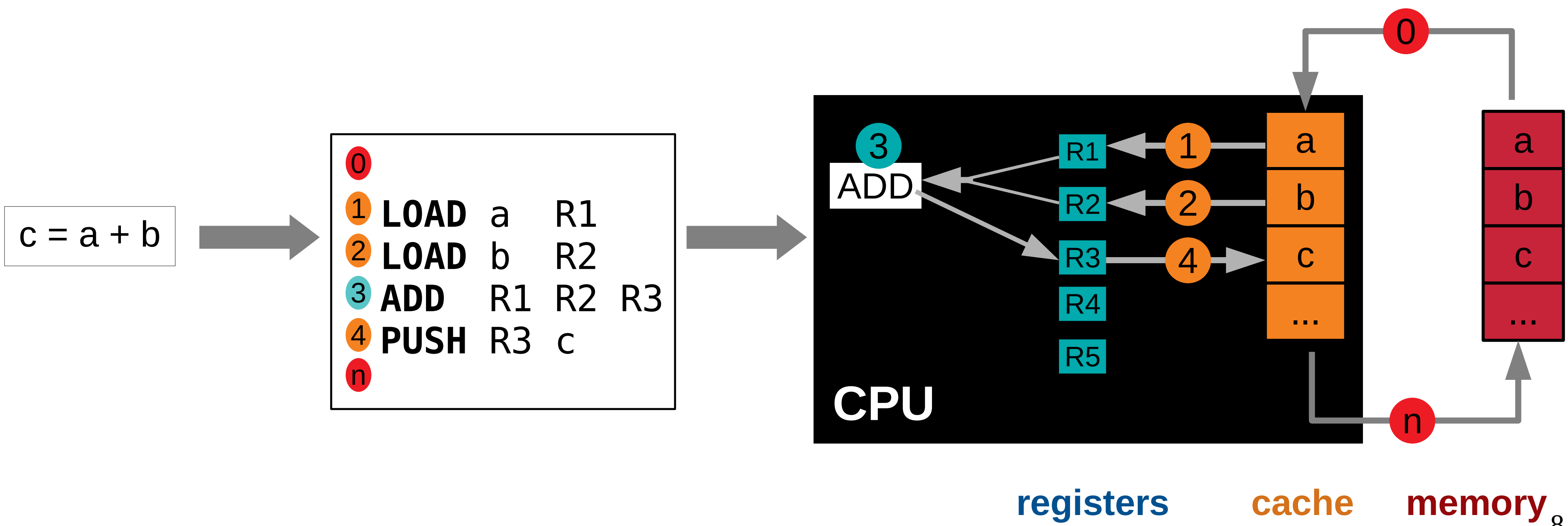
Float: FADD (1-3), FMUL (2-5), FDIV (35-40)

Context switch: 1000s of cycles

→ Reset cache and registers

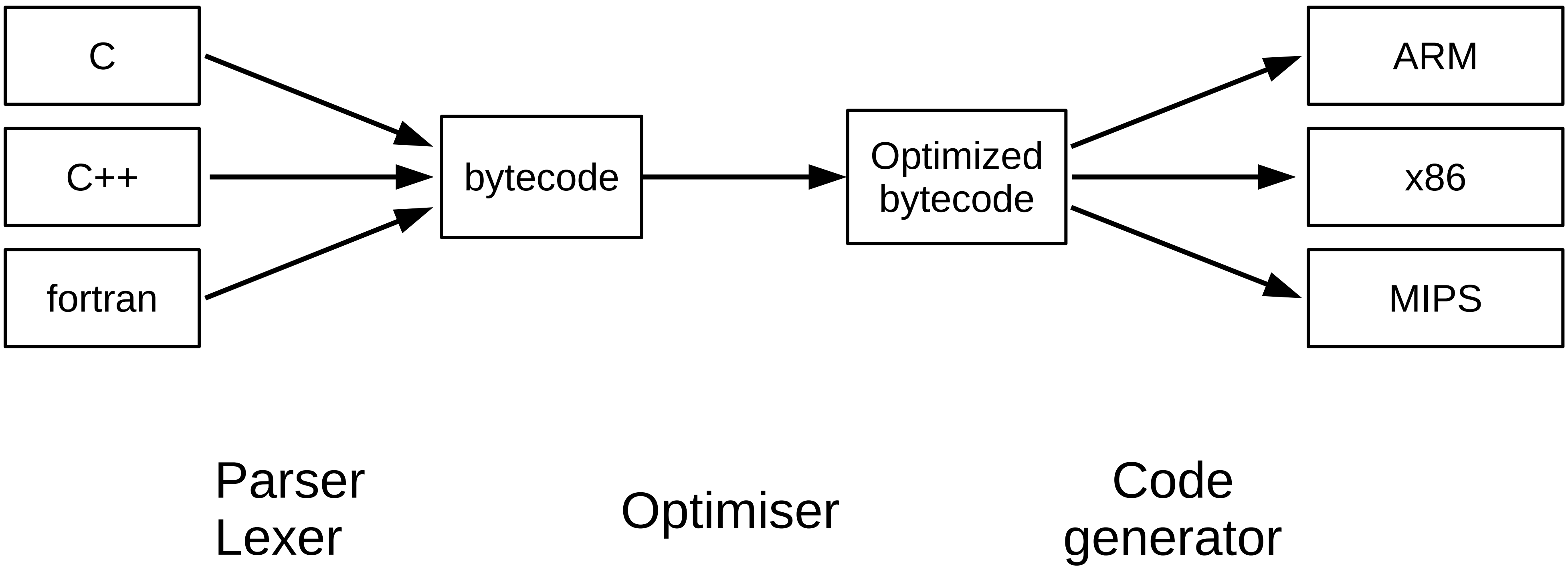
ASM and CPU instructions

- Manipulate register values
  - Basic and extended instructions
- Load and push values in memory
  - CPU cache and page fault



Compilation

Turning an “abstract” programming language into “concrete” processor instructions



Interpreted language

Interprets bytecode: “Virtual machine”

- Compile during execution (Python, R)
  - Caching bytecode (.pyc)
- Compile in advance (Java)

Static vs Dynamic typing

C / Java

```
int a = 2;
a = 3;
a = 2.5;
```

Python

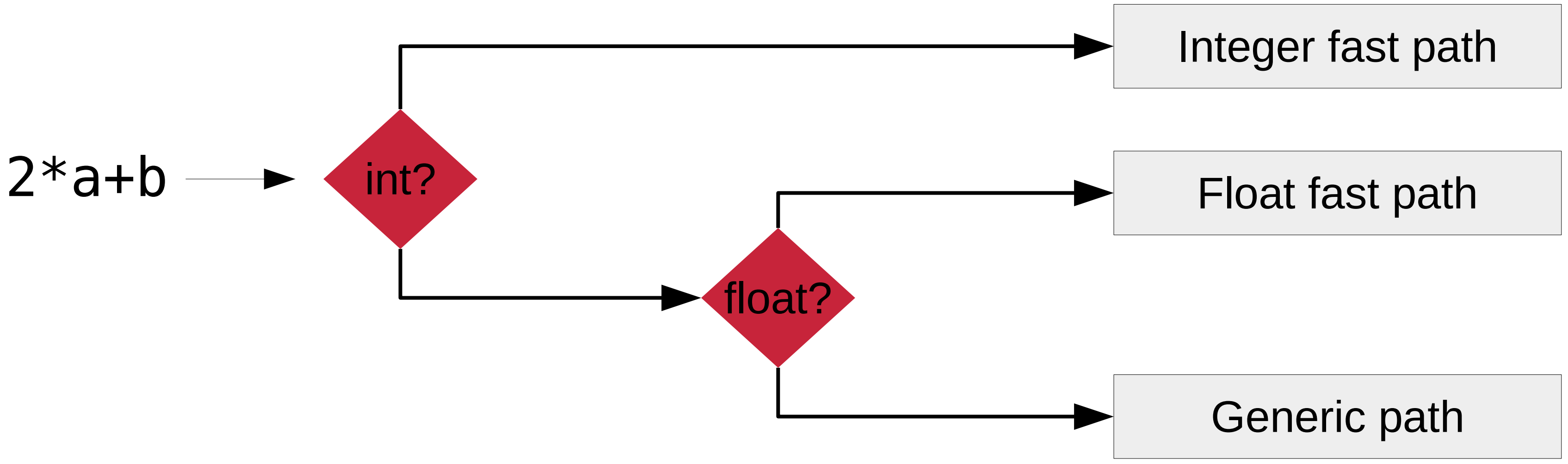
```
a = 2
a = 3.5
a = "whatever"
```

Dynamic: variables have no fixed type

→ Execution depends on the current type

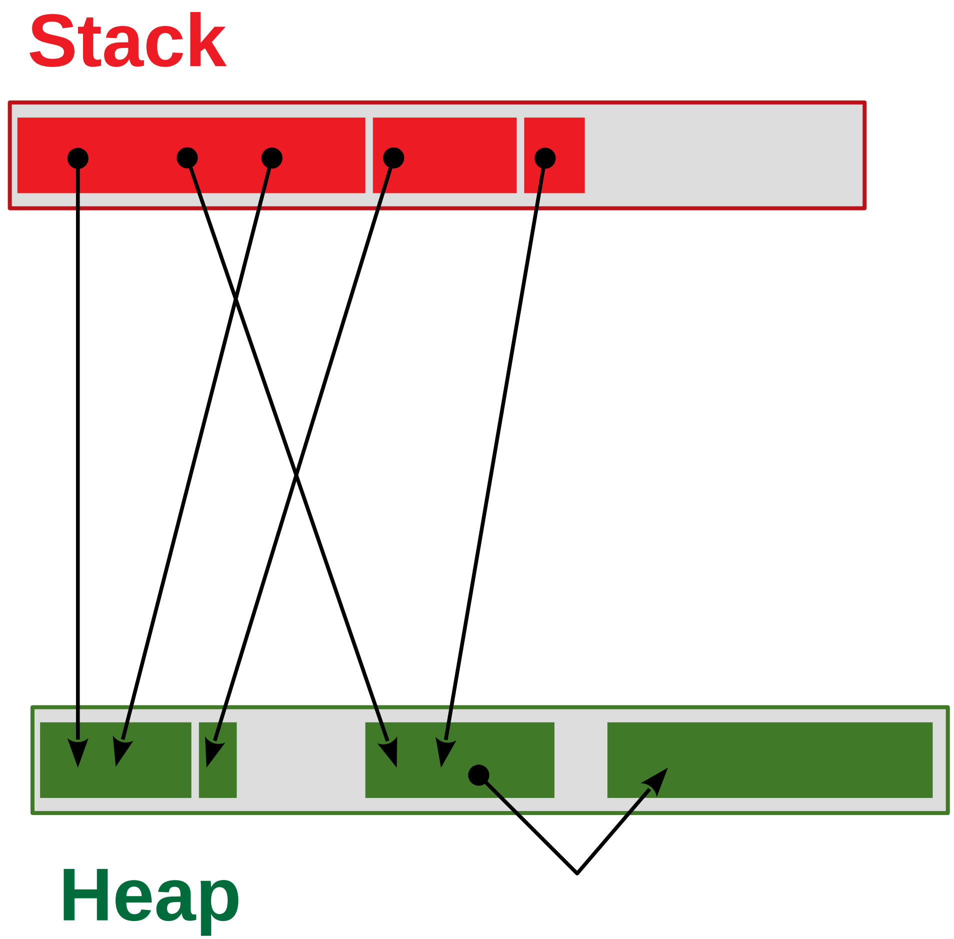
Adapting to dynamic types

- Abstract bytecode
- Runtime types → fast paths



Cost of type checking
→ Apply to "large" blocks of code

Memory management



- Follows function calls
- Sized at compile time

- Sized at runtime
- Explicit allocation/free
  - Manual or managed
  - Fragmentation
- Pointers in the stack (or heap)

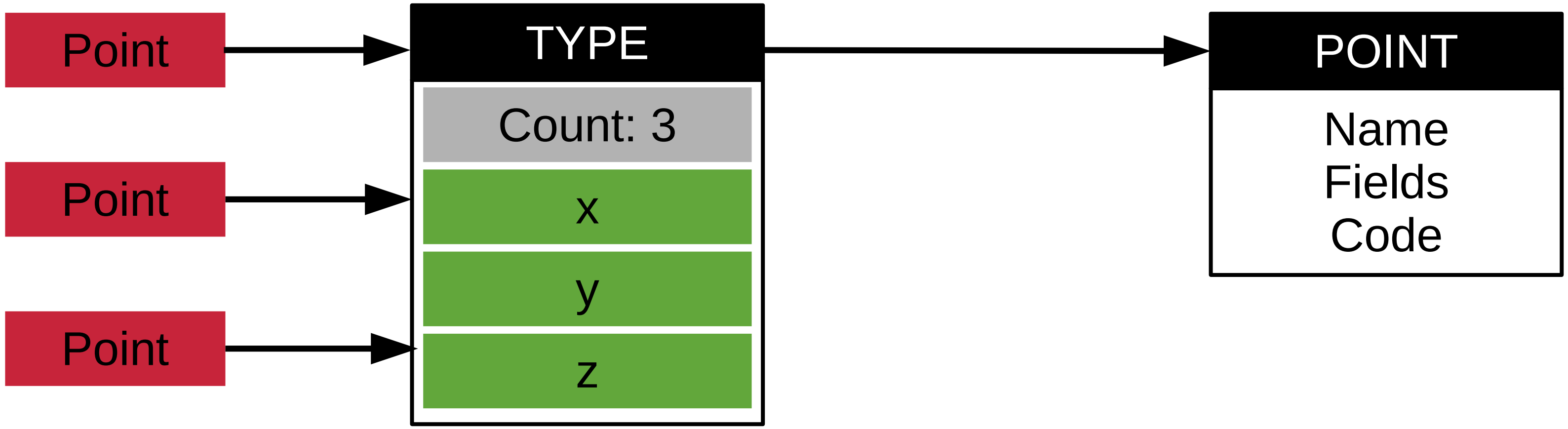
Problems with pointers

- Missing free
  - Memory leaks
- Multiple copies of a pointer
  - Dangling pointer (early free)
  - Race conditions, memory corruption
- Common misuses
  - NULL or invalid pointers
  - Buffer overflows

Automatic memory management

Reference counting

- Increase for each pointer
- Decrease when pointer removed from Stack

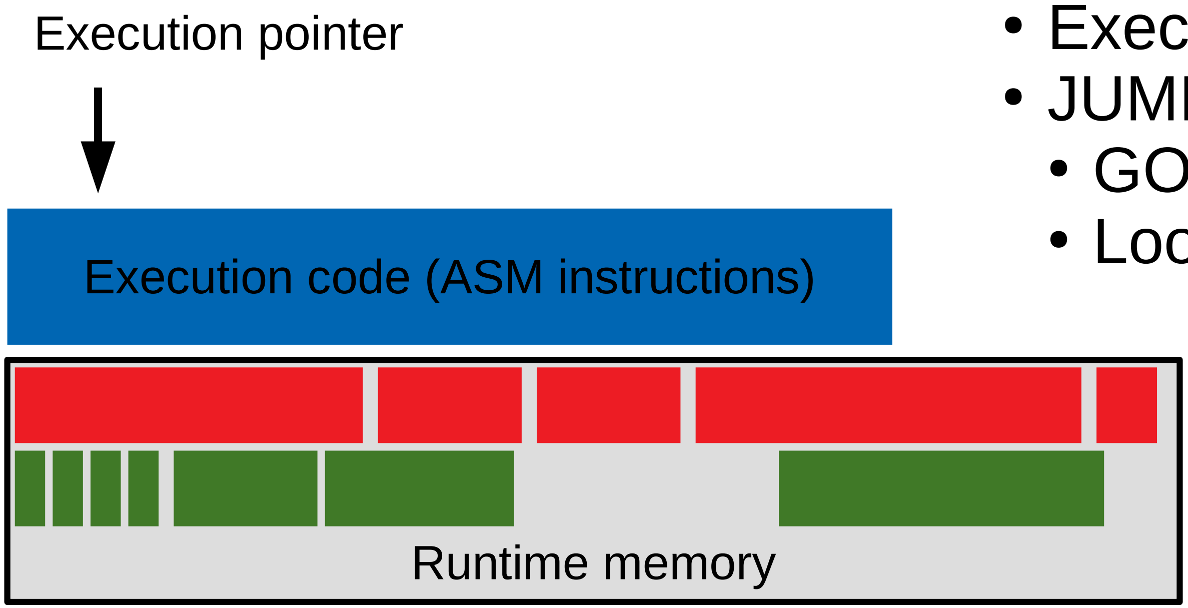


Cyclic references → memory leak

➔ Garbage collection

- Detect isolated parts of a pointer graph
- Delayed free, extra cost

Memory layout of a program

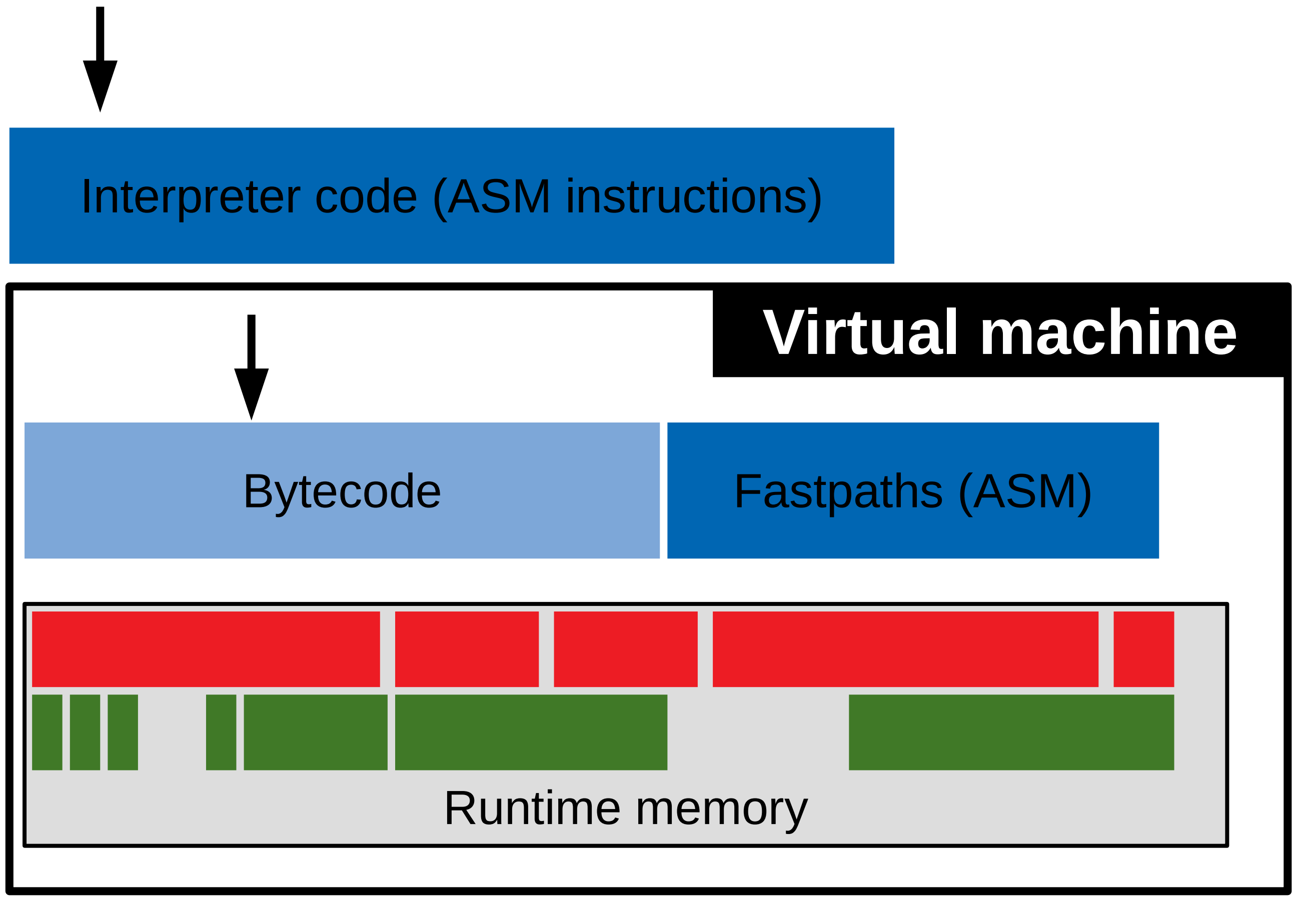


- Execute $\rightarrow$ advance the pointer
- JUMP Instruction
  - GOTO in early languages
  - Loops and function calls

Memory layout is critical for performance

Variables in the same “page” (often 4kb)
are transferred to CPU cache in a single step

Architecture of an interpreter



Bytecode/ASM ratio

- Depends on the interpreter
- Changes during runtime (JIT)

Binary representation of integers

(signed) sum of powers of two

`00011011`

`4+8+32+64`

`→ 108`

8 "bits" → 2^8 = 256 possible values

- `[0 → 255] or [-127 → 127]`

`→ overflow (127+1 = -127)`

Variants:

- Little/big endian: side of the strong bit
- Sign encoding:
  - Absolute value + sign?
  - One's complement: swap all bits

Other binary encodings

Floats: value * multiplier

0001101000111001

±2^factor*value

Factor has it's own sign

Precision scales with value
→ Floats are not exact

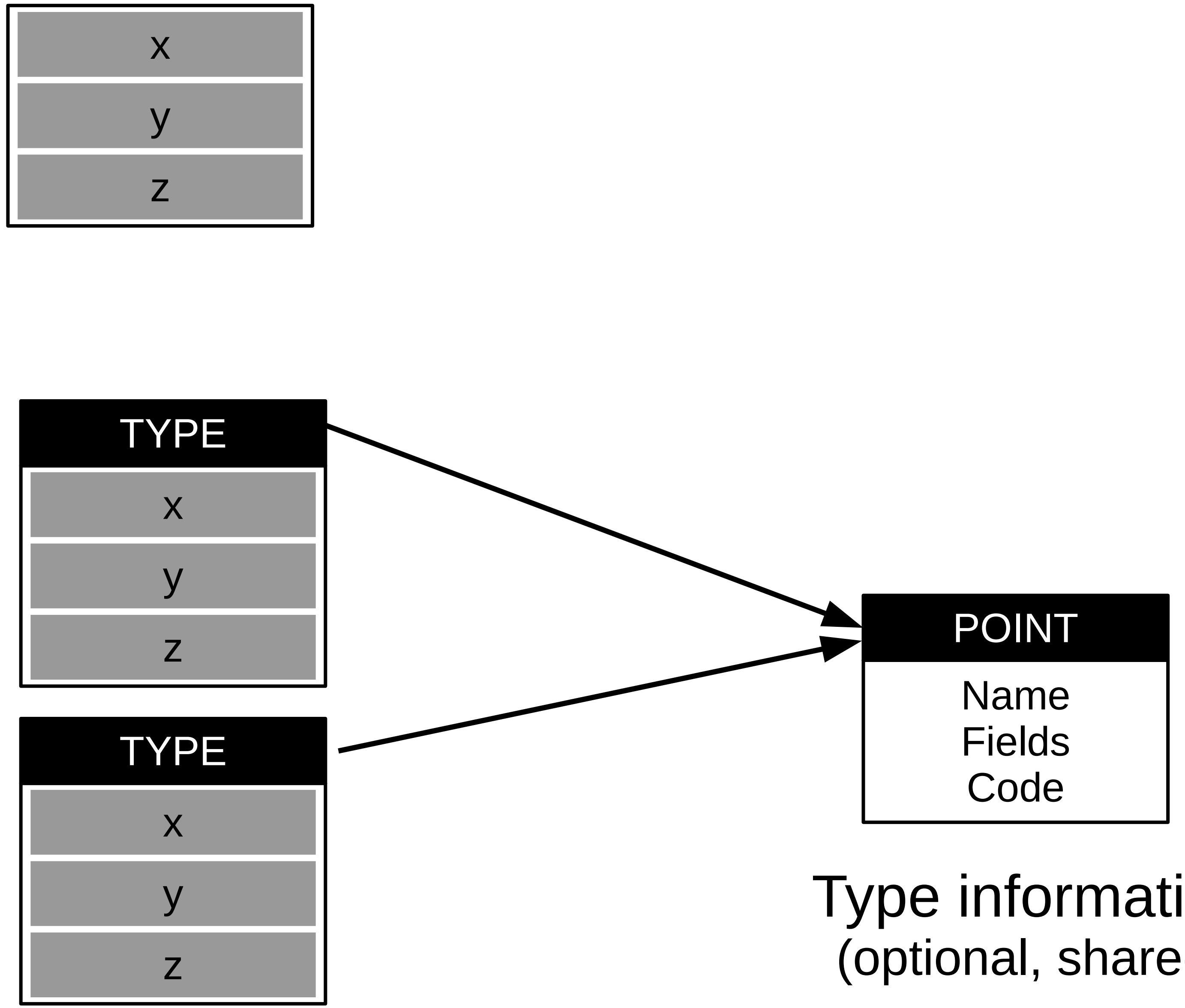
Single (32 bits) or double (64bits) precision

Text: the encoding can of worms

- ASCII: historical 7-bits encoding (128 characters, no accents)
- Many 8-bits extensions (ISO-8859-1 in western europe)
- Unicode: mapping characters worldwide
  - ➔ Encodings: UTF-8, UTF-16, UTF-32

Grouping related values with objects

Point in space: (x, y, z)



Type information
(optional, shared)

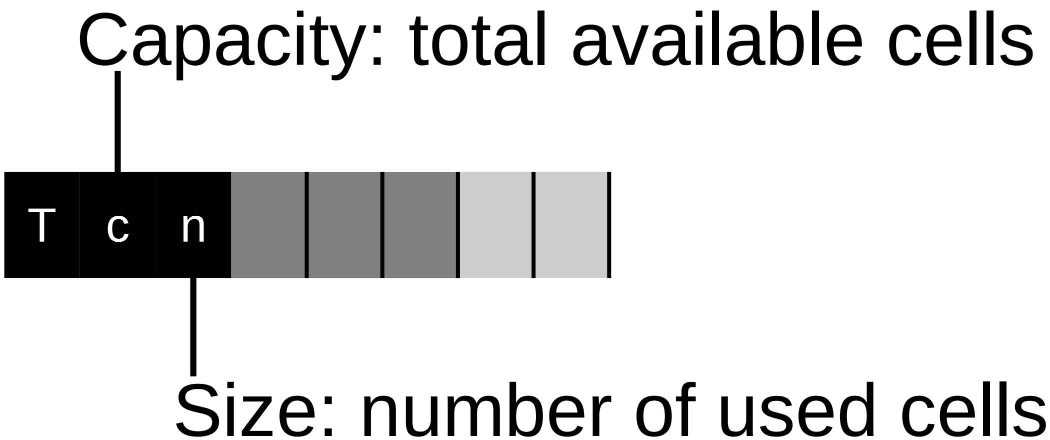
Arrays: lists of values

Raw array: just the values

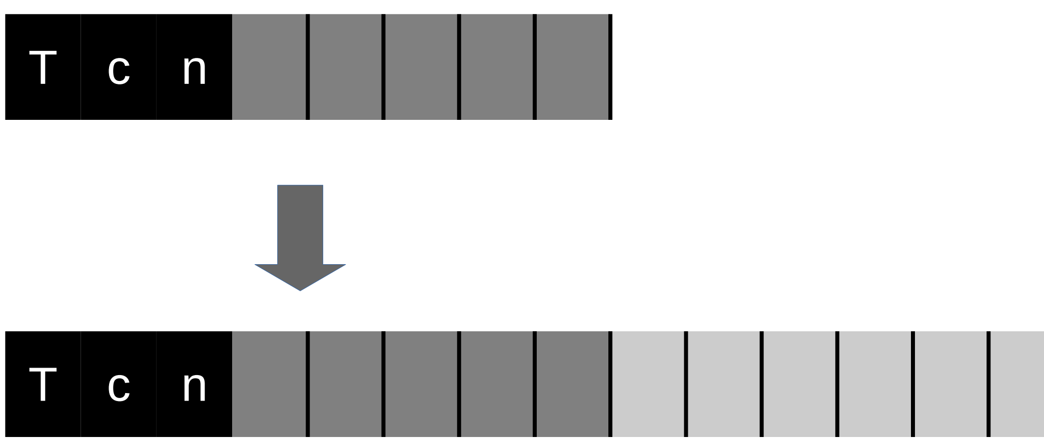
Add size (avoid overflows)

Add type information

Dynamic (growing) arrays



Increase size until the capacity is reached, then allocate a new array, copy values, and free the old one



Array and pointer arithmetics

S = SIZE(T)

Computing the address of an array element

```
v = A[k]
```

```
if k > *(A+1): ERROR
a = A + 2 + k*S
v = *a
```

Fast path to access the next element (no multiplication)

```
a = A + 2
end = A + 2 + n*S
```

```
while a < end:
    a += S
    v = *a
```

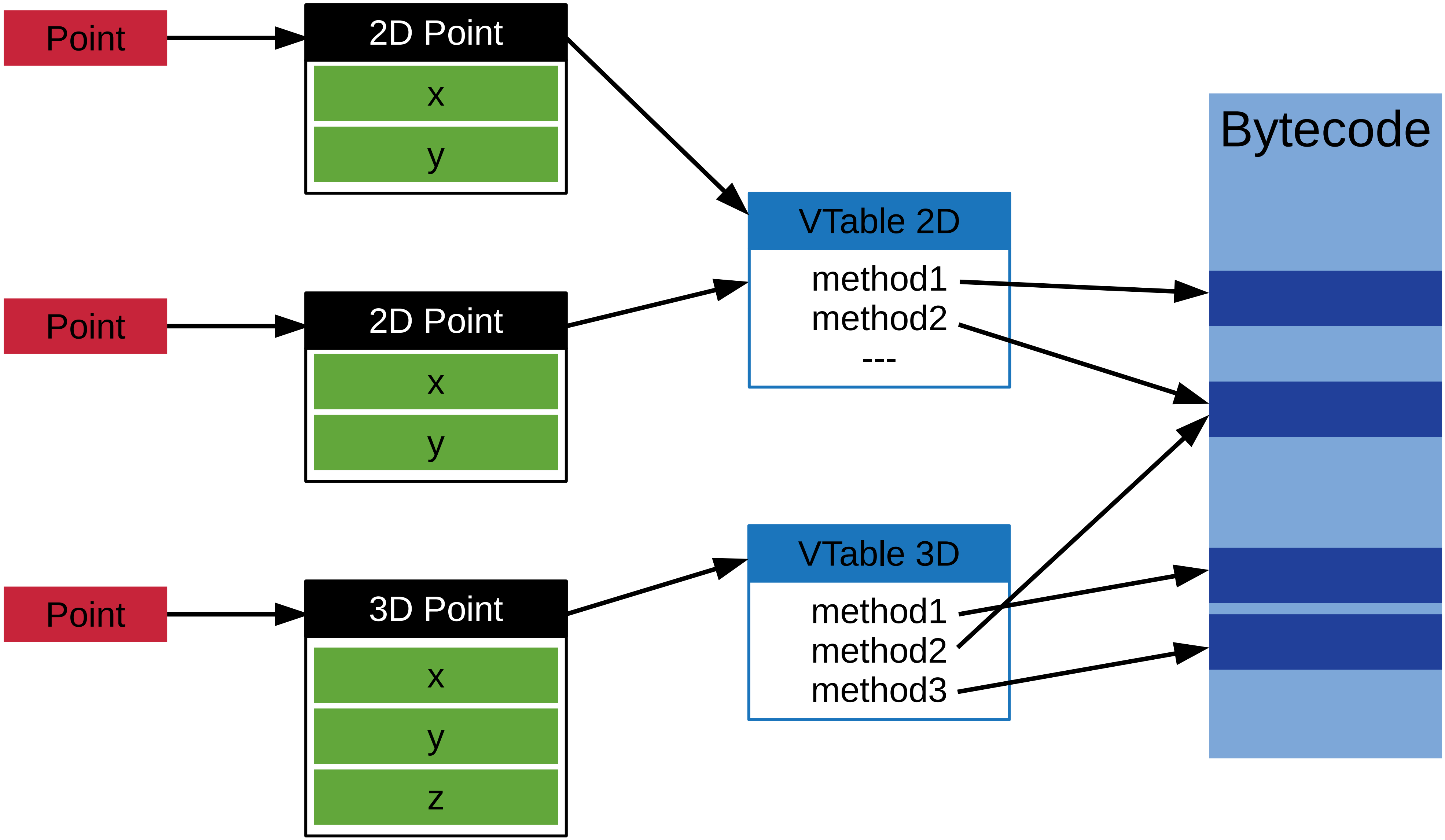
Calling a function

- Save the execution pointer to stack
- Add parameters of the function call
- JUMP to the function code

… Execute the function …

- Remove function from the stack
- Add the return value
- JUMP to the stored execution pointer

Inheritance and dynamic dispatch



2+ extra pointers to find the JUMP address

Summary: memory cost

Shared

- Runtime
- Type annotations, function tables

Per object

- Pointer + Typeref + Refcount
  - Triple memory for small objects (or worse)

  ➔Use collections of native objects (numpy)

Summary: CPU cost

- At startup: compile again and again
- Bytecode interpretation
- Type checking
- Array bound checking
- Dynamic dispatch
- Garbage collector

➡ Not so bad for data analysis (I/O bound)
And long-running tasks with JIT

When/how should we go low level?

- Identify bottlenecks: memory or CPU?
  - ➔ <u>Some</u> native module(s)

- Pick a language: ASM C/C++ Rust
  - Beware of pointers, right compromise?

- Data structures are key
  - Fast for most common operation
  - Caching is tricky but can help