基础软件中心

栈式虚拟机和函数(Part1)

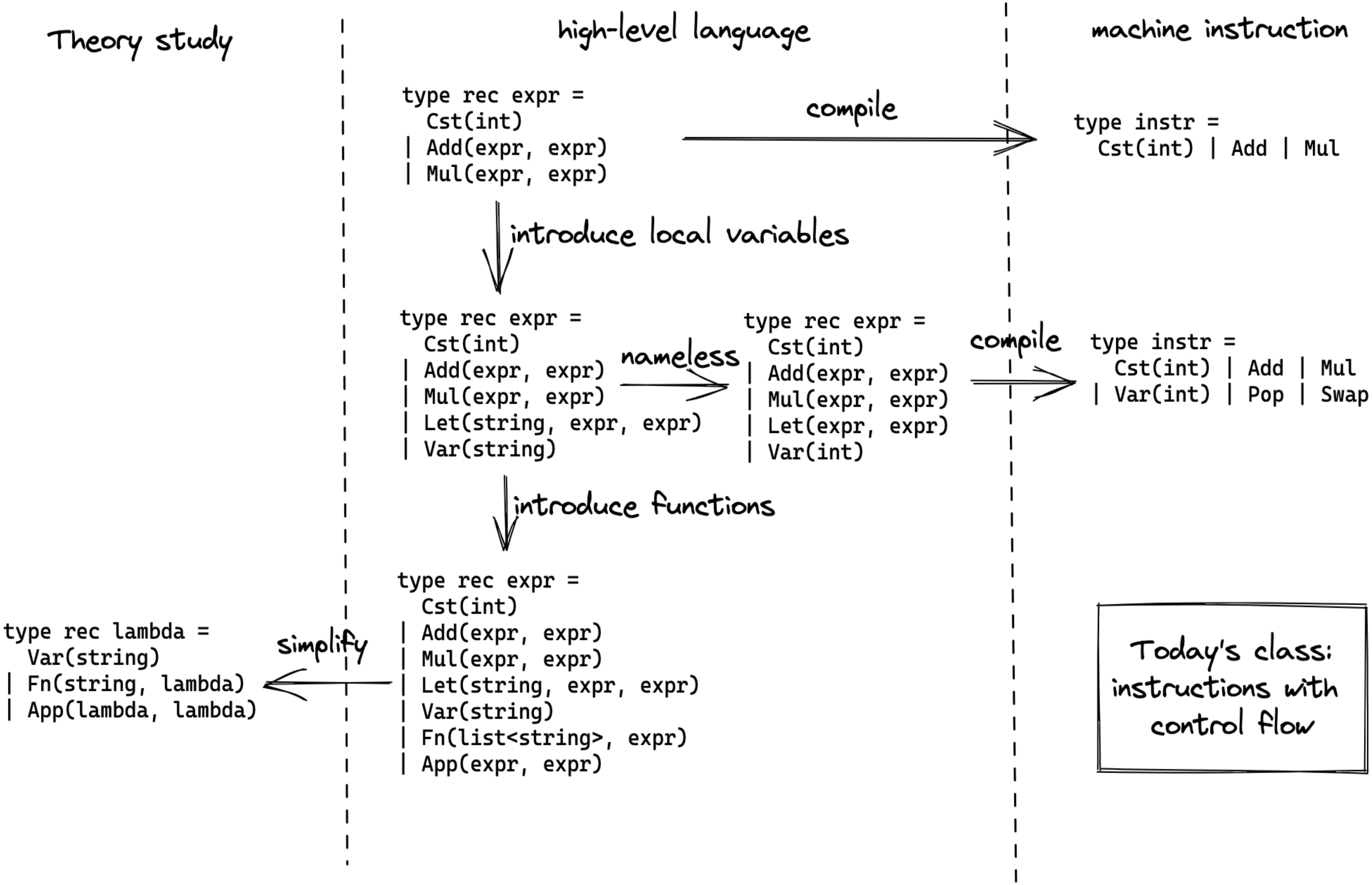
基础软件理论与实践公开课

张宏波

Review

What we have learned so far?

idea
基础软件中心


Theory study
high-level language
machine instruction
type rec expr =
Cst(int)
| Add(expr, expr)
| Mul(expr, expr)
compile
type instr =
Cst(int) | Add | Mul
introduce local variables
type rec expr =
Cst(int)
| Add(expr, expr)
| Mul(expr, expr)
| Let(string, expr, expr)
| Var(string)
nameless
type rec expr =
Cst(int)
| Add(expr, expr)
| Mul(expr, expr)
| Let(expr, expr)
| Var(int)
compile
type instr =
Cst(int) | Add | Mul
| Var(int) | Pop | Swap
introduce functions
type rec lambda =
Var(string)
| Fn(string, lambda)
| App(lambda, lambda)
simplify
type rec expr =
Cst(int)
| Add(expr, expr)
| Mul(expr, expr)
| Let(string, expr, expr)
| Var(string)
| Fn(list<string>, expr)
| App(expr, expr)
Today's class:
instructions with
control flow

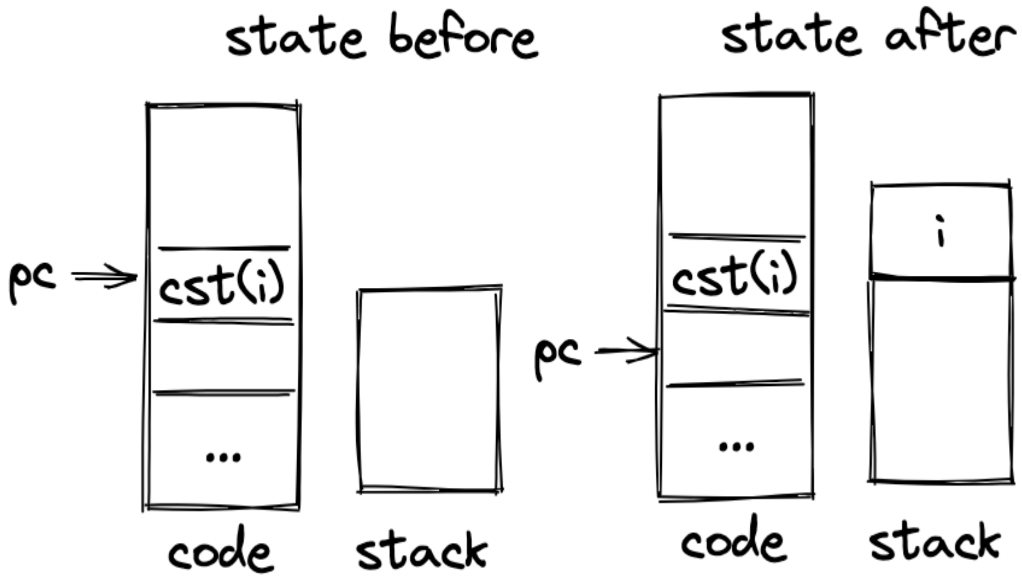
Next step

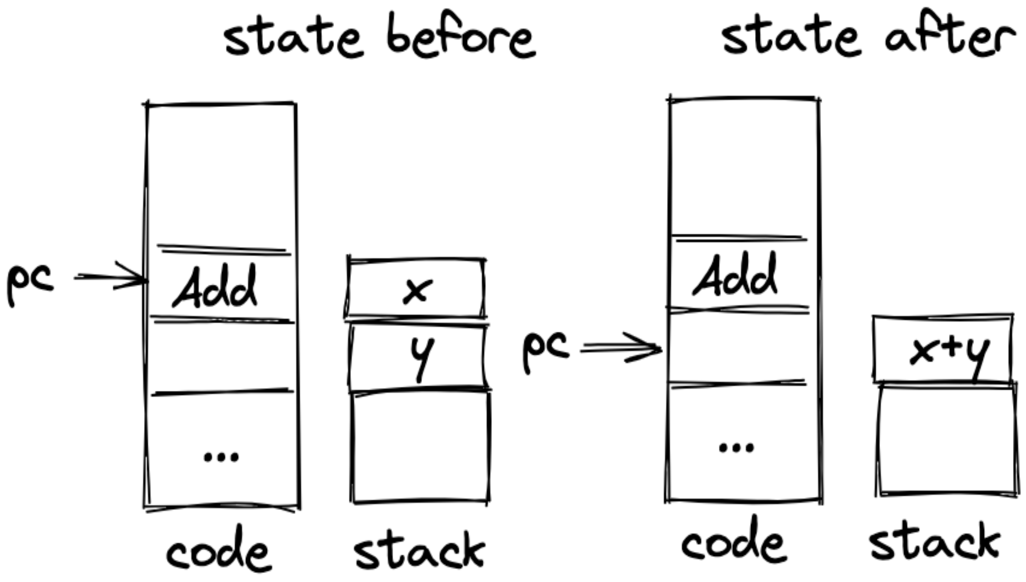
Compile functions and conditional expressions to stack machine instructions

- Today's class: introduces new instructions to support function call and branch
- Next week's class: compile a simplified "tiny" language (support c-like functions and conditional expression) to the instruction
- Future: compile first-class functions

Stack machine (Review)

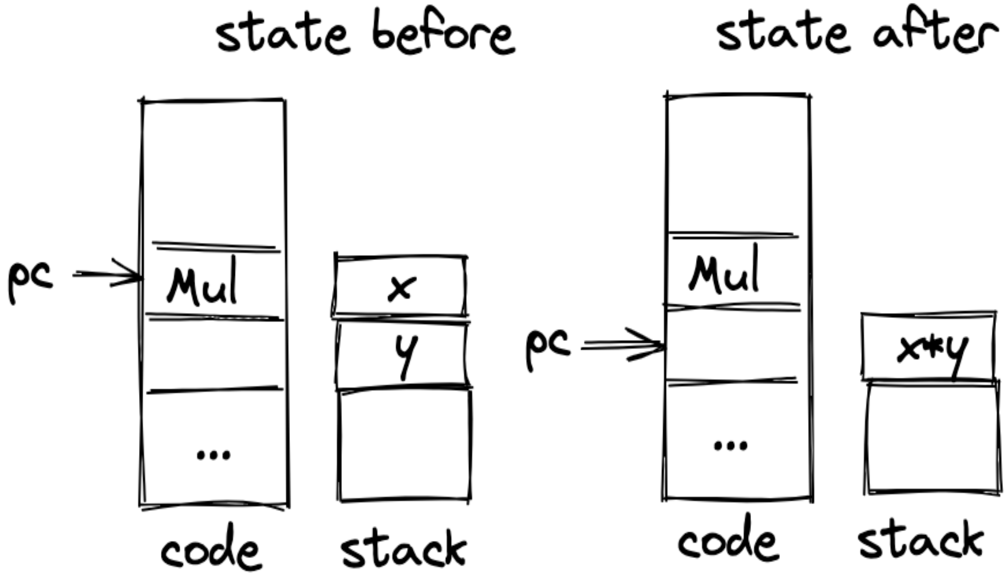
```
type instr = Cst(int) | Add | Mul | Var(int) | Pop | Swap
```

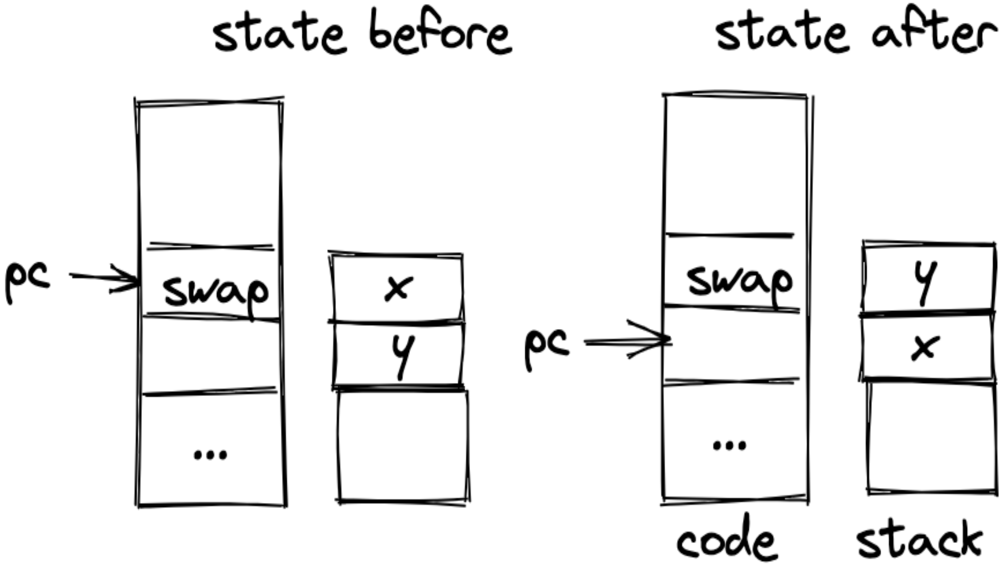




Stack machine (Review)

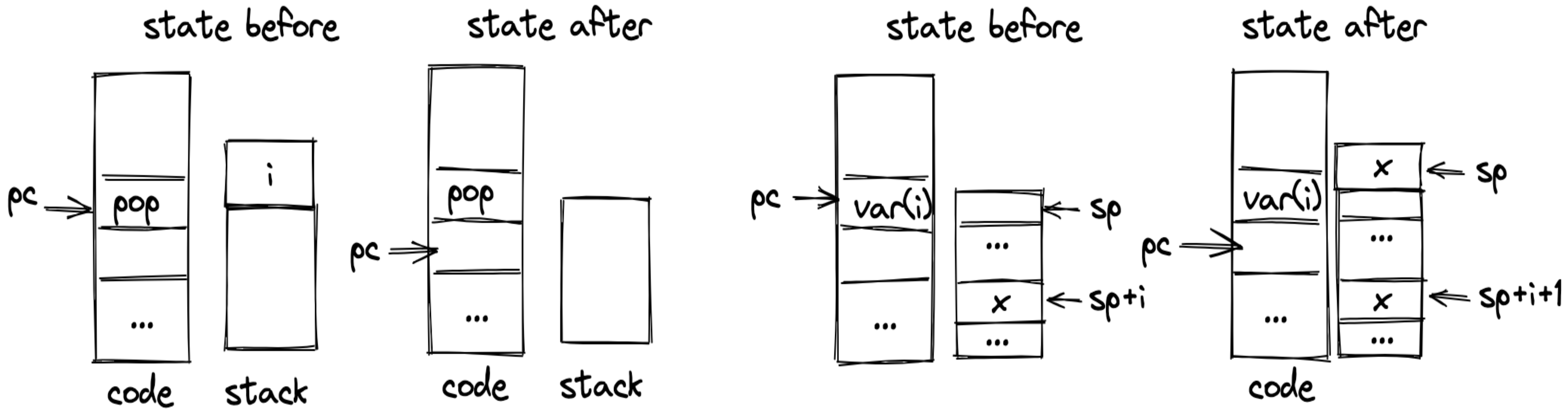
```
type instr = Cst(int) | Add | Mul | Var(int) | Pop | Swap
```





Stack machine (Review)

```
type instr = Cst(int) | Add | Mul | Var(int) | Pop | Swap
```



Question

What do we need if we want to support functions and if-then-else?

Question

What do we need if we want to support functions?

- Function call and return
  - Remember the PC before the function call and jump back when it returns
  - Pass arguments and return values

What do we need if we want to support functions and if-then-else?

- If-then-else
  - jump to a code location based on the value of an operand

New instructions (Label)

- Pseudo-instruction (more details will be explained later)
- None of the previous instructions manipulates PC to jump around
- Labels are locations in the code that can be jump targets
- Programs ususually start executing from a specific code label (e.g. "main")

```
type label = string
type instr = ... | Label(label)
```

New instructions (Call and Ret)

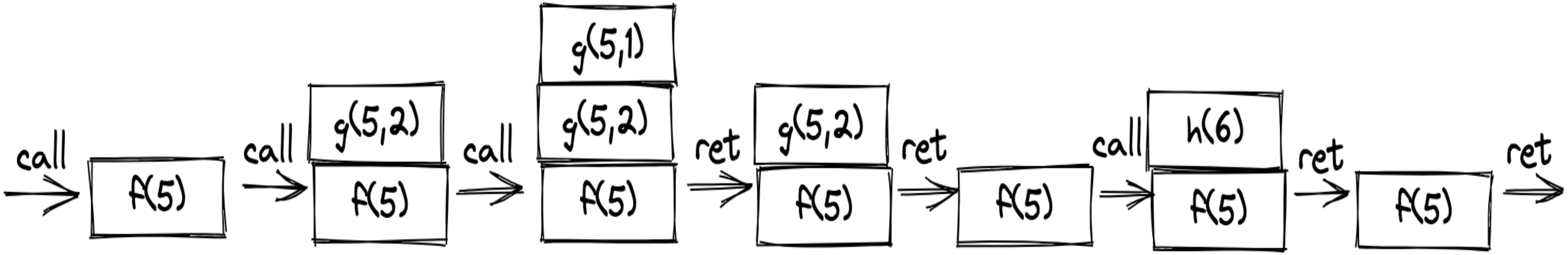
```
type instr = ... | Call(label, int) | Ret(int)
```

- The `int` part corresponds to the arity of the function, which is part of the *metadata*
- The arity information is used to maintain the stack balance property

Stack frame

For example,

```
let f(x) = g(x, 2) + h(x + 1) in
let rec g(x, n) = if n > 1 then g(x, n-1) else 0 in
let h(x) = x * 2 in
f(5)
```

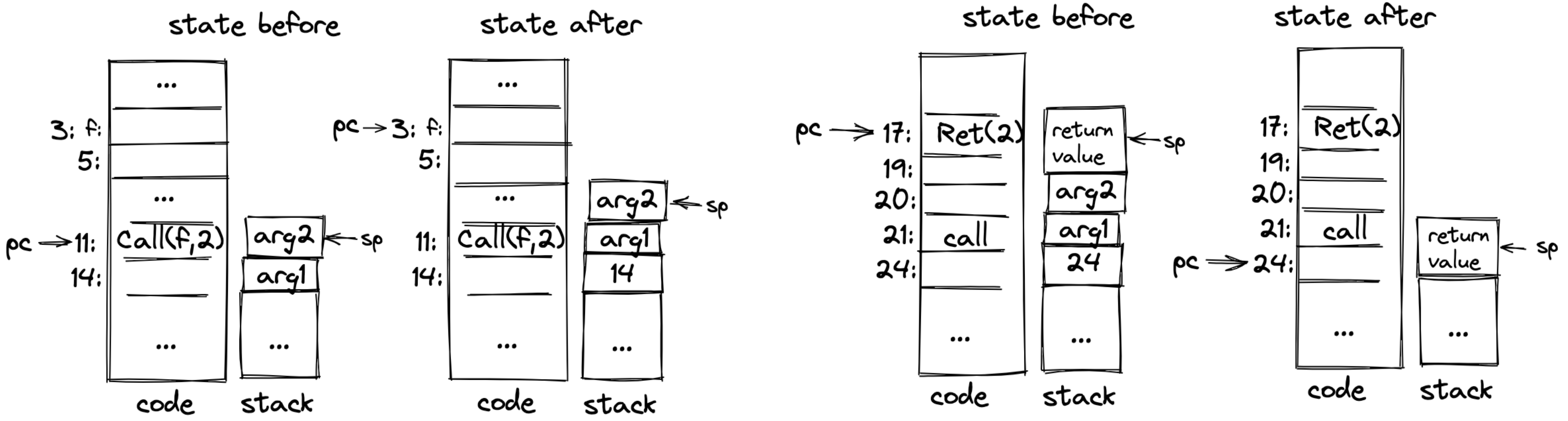


- Stack frames keep track of the program counter(PC), arguments, etc

New instructions (Call and Ret)

- The stack frames form a structure of stack so they can be merged with the stack

```
type instr = ... | Call(label, int) | Ret(int)
```



New instructions (Goto and IfZero)

- Conditional/unconditional branch

```
type instr = ... | Goto(label) | IfZero(label)
```

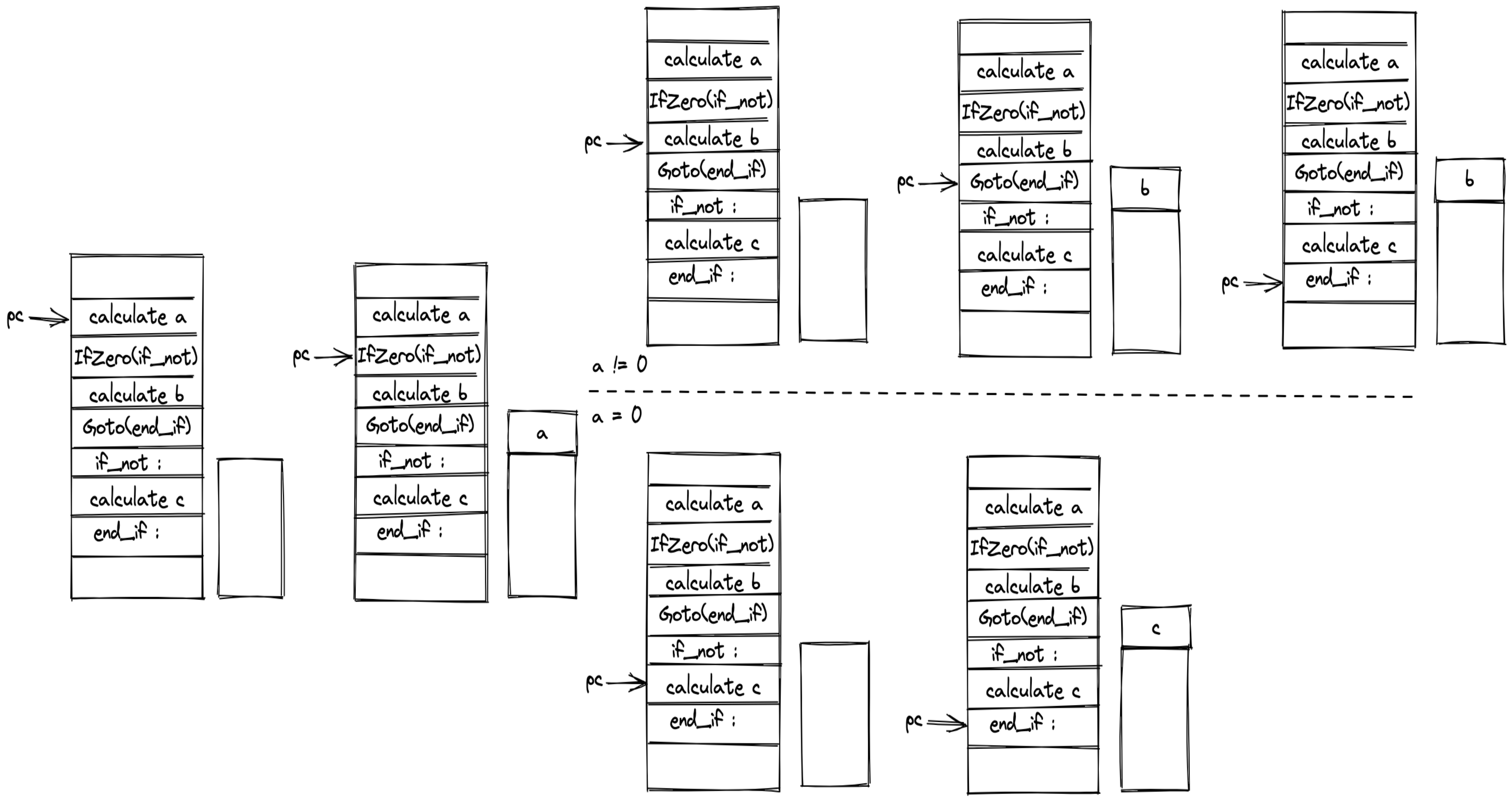
- For example, `[[ if a then b else c]]` is compiled to

```
[[a]]
IfZero(if_not) // if a = 0 then jump to if_not else fall through
[[b]]          // code for ifso
Goto(end_if)   // do not fall through
Label(if_not)
[[c]]          // code for ifnot
Label(end_if)
```

- `[[a]]` represents the generated instructions for calculating the expression `a`

If-Then-Else

基础软件中心



Instructions

```
type instr =
  | Cst(int) | Add | Mul | Var(int) | Swap | Pop | Label(string)
  | Call(string, int) | Ret(int) | Goto(label) | IfZero(label) // control flow instructions
  | Exit
```

- `Exit` terminates the execution and return the value off the top of the stack

Pseudo-instruction

- Labels are *pseudo-instructions* that are translated away by *assembler*
- The assembly language instructions is a layer of *abstraction* of top of machine code
  - for example, `Cst(1); Var(2); Add`
- The machine can only understand the binary code, which is *language independent*
  - for example, `1000011011...`
- The assembler translates assembly to binary code

Encoding specification

Instr	Opcode	Oprand1	Oprand2	Size
Cst(i)	0	i	–	2
Add	1	–	–	1
Mul	2	–	–	1
Var(i)	3	i	–	2
Pop	4	–	–	1
Swap	5	–	–	1
Call(l, n)	6	get_addr(l)	n	3
Ret(n)	7	n	–	2
IfZero(l)	8	get_addr(l)	–	2
Goto(l)	9	get_addr(l)	–	2
Exit	10	–	–	1

Implementing assembler

基础软件中心

- Translate labels in `Goto(label)`, `IfZero(label)`, `Call(label, n)` to addresses

```
// auxiliary function
let size_of_instr = (instr: instr): int => { ... }
```

```
let encode = (instrs: array<instr>): array<int> => {
  let int_code: array<int> = Int32Array.make(...)
  let position = ref(0)   // index to the int_code
  let label_map: HashMap.t<string, int> = HashMap.make(...)
  for cur in 0 to length(instrs) - 1 { // construct the label_map
    ... // record the PC per each label
  }
  for cur in 0 to length(instrs) - 1 { // translate to int_code
    ...
  }
  int_code
}
```

Implementing assembler

- construct the `label_map`

```
let encode = (instrs: array<instr>): array<int> => {
  let int_code: array<int> = Int32Array.make(...)
  let position = ref(0)   // index to the int_code
  let label_map: HashMap.t<string, int> = HashMap.make(...)
  for cur in 0 to length(instrs) - 1 { // construct the label_map
    switch instrs[cur] {
    | Label(l) => HashMap.set(label_map, l, position.contents)
    | instr => position := position.contents + size_of_instr(instr)
    }
  }
  position := 0
  for cur in 0 to length(instrs) - 1 { // generate int_code
    ...
  }
  int_code
}
```

Implementing assembler

idea
基础软件中心

```
let encode = (instrs: array<instr>): array<int> => {
  ... // construct the label_map
  position := 0
  for cur in 0 to length(instrs) - 1 { // generate int_code
    switch instrs[cur] {
    | ...
    | Call(l, n) => {
        let label_addr = HashMap.get(label_map, l)
        int_code[position.contents] = 6 // opcode of Call is 6
        int_code[position.contents+1] = label_addr
        int_code[position.contents+2] = n
      }
      position := position.contents + 3
    }
  }
  int_code
}
```

- Homework: complete the assembler following the encoding spec

Implementation

基础软件中心

- Runtime state of the stack machine

```
type operand = int
type vm = {
  code: array<int>, // immutable
  stack: array<operand>, // runtime stack
  mutable pc: int,  // pc register
  mutable sp: int, // sp register
}
// stack operators
let push = (vm: vm, x: operand) => { ... }
let pop = (vm: vm) : operand => { ... }
// initial state
let initVm = code => {
  code, stack: init_stack, pc: get_init_pc(code), sp: 0,
}
```

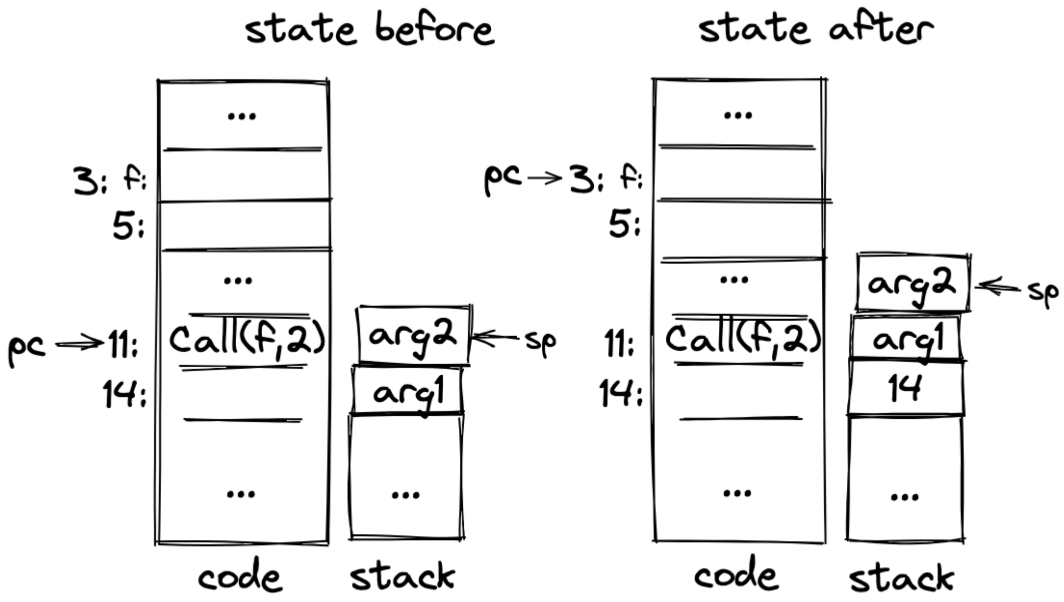
Implementation

- Overall structure of stack machine's execution

```
let run = (vm: vm): operand => {
  let break = ref(false)
  while !break.contents {
    let opcode = vm.code[vm.pc]
    switch opcode {
    | ... => { ... }
    | 10 => break := true //Exit
    | _ => assert false
    }
  }
  pop(vm) // return value
}
```

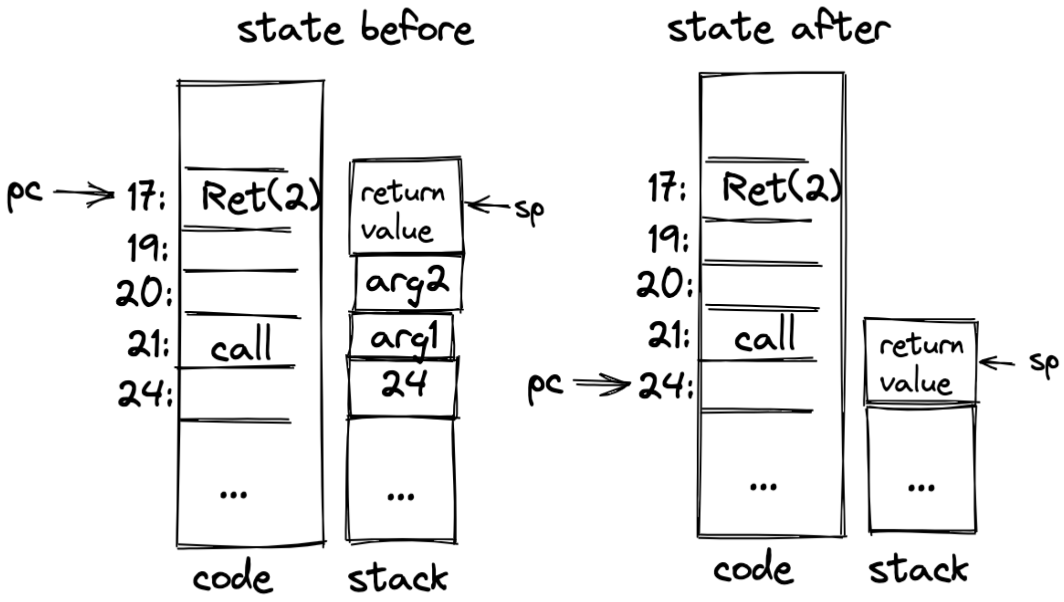
Implementation

```
| 6 => { // Call(addr, arity)
  let target_pc = vm.code[vm.pc+1]
  let arity   = vm.code[vm.pc+2]
  let next_pc = target_pc
  spliceInPlace(
    ~pos=vm.sp - arity,
    ~remove=0,
    ~add=[vm.pc+3], // return address
    vm.stack)
  vm.sp = vm.sp + 1
  vm.pc = next_pc
}
```

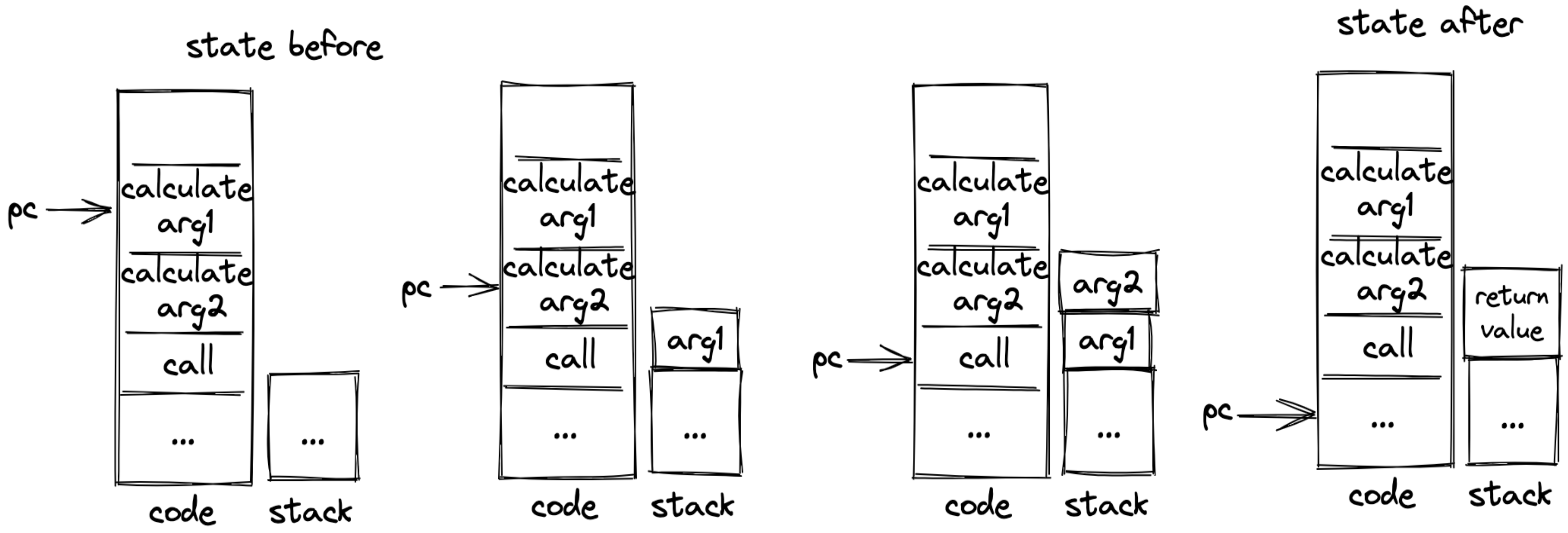


Implementation

```
| 7 => { // Ret(arity)
  let arity = vm.code[vm.pc+1]
  let res = pop(vm)
  vm.sp = vm.sp - arity
  let next_pc = pop(vm)
  let _ = push(vm, res)
  vm.pc = next_pc
}
```



Stack balance property



- Only one extra value on the top of the stack: `f(arg1,arg2, ...argN)`

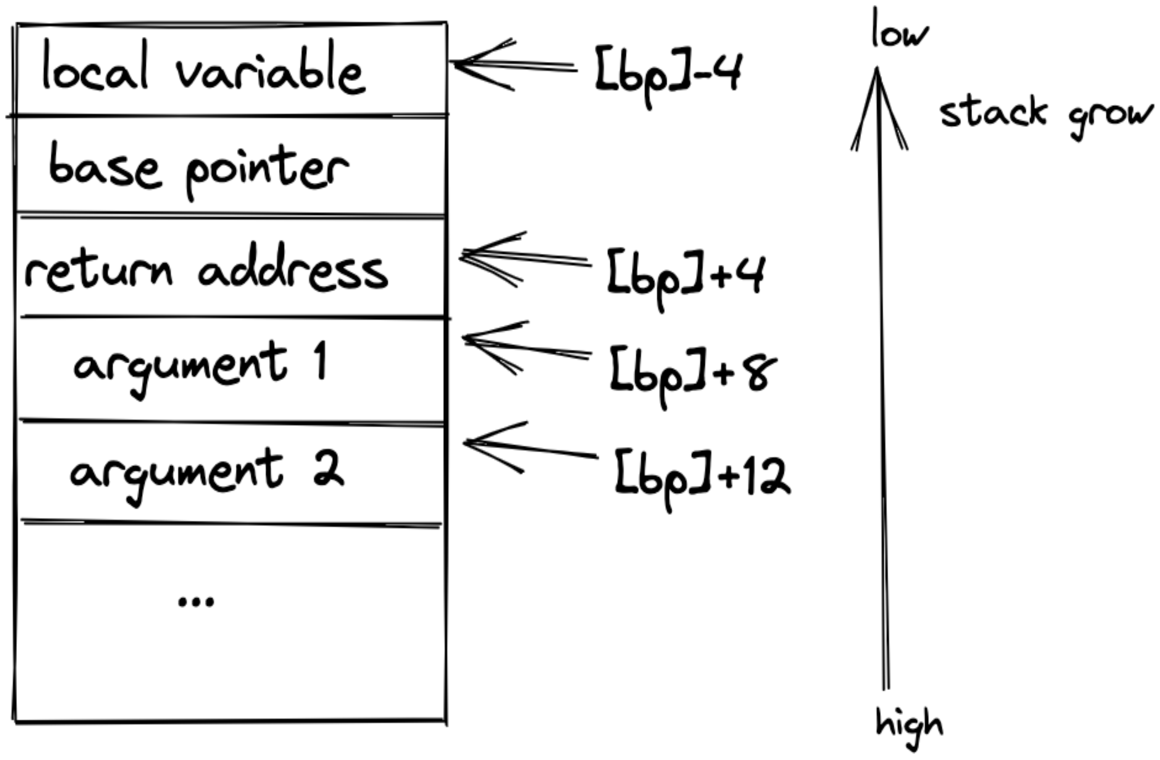
Calling convention

The *interface* between caller and callee: how the stack/registers are organized when function call happens

- The order of pushing arguments and PC to the stack
- `Call(f, n)` and `Ret(n)` carry the arity of the function explicitly, which is not necessary under some conventions
- caller/callee saved registers
- return value passed by register or stack

Alternative conventions

- stable addressing mode: using base pointer



- x86 convention

Homework

Implement the virtual machine in C/C++/Rust