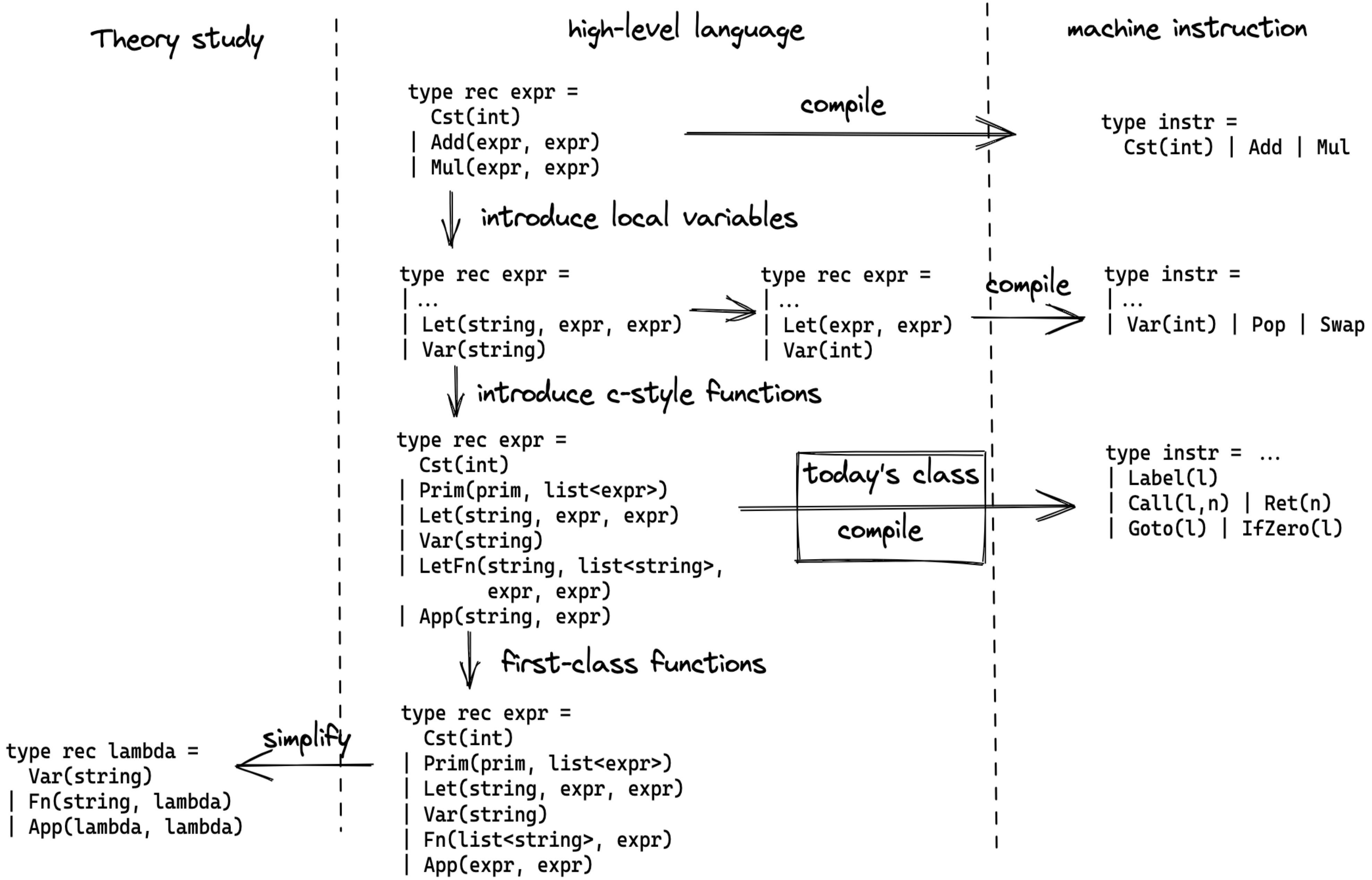
栈式虚拟机和函数(Part2)

基础软件理论与实践公开课

ZhangYu

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 =
| …
| Let(string, expr, expr)
| Var(string)
type rec expr =
| …
| Let(expr, expr)
| Var(int)
compile
type instr =
| …
| Var(int) | Pop | Swap
introduce c-style functions
type rec expr =
Cst(int)
| Prim(prim, list<expr>)
| Let(string, expr, expr)
| Var(string)
| LetFn(string, list<string>, expr, expr)
| App(string, expr)
today's class
compile
type instr = …
| Label(l)
| Call(l,n) | Ret(n)
| Goto(l) | IfZero(l)
first-class functions
type rec expr =
Cst(int)
| Prim(prim, list<expr>)
| Let(string, expr, expr)
| Var(string)
| Fn(list<string>, expr)
| App(expr, expr)
simplify
type rec lambda =
Var(string)
| Fn(string, lambda)
| App(lambda, lambda)

Question

how do we compile the language with function call and if-then-else?

Compile functions

- Tiny language 3

```
type rec expr =
  | Cst (int)
  | Add (expr, expr)
  | Mul (expr, expr)
  | Var (string)
  | Let (string, expr, expr)
  | Fn (list<string>, expr)
  | App (expr, list<expr>)
```

- Leave first-class functions to later classes
- Funarg problem: the difficulty of using stack-based memory allocation to implement first-class functions

Simplify the language

- Tiny language 4 (essentially equivalent to tiny c)
  - No free variables: *lifetime* of variables are aligned with function invocation
  - No indirect calls: resolve the function label statically

```
type prim = Add | Mul | ...
type rec expr =
    | Cst (int)
    | Var (string)
    | Let (string, expr, expr)
    | Letfn (string, list<string>, expr, expr)
    | App (string, list<expr>)
    | Prim (prim, list<expr>)
    | If (expr, expr, expr)
```

Example: Fibonacci function

Concrete syntax:

```
letfn fib(n) =
  if n <= 1 then { 1 }
  else { fib(n-1) + fib(n-2) }
in fib(5)
```

Abstract syntax:

```
Letfn(
  fib, [n],
  If(Prim(<=, [Var(n), Cst(1)]),
     Cst(1),
     Prim(+, [App(fib, ...), App(fib, ...)]))
  App(fib, [Cst(5)])
)
```

Overview

1. Preprocess: flatten the code by lifting the functions
2. Compilation: compile the functions: how to deal with arguments?
   - caller: push arguments to the stack
   - callee: find the arguments on the stack
3. Postprocess: add an entrance and an exit

```
let a = 2 in
letfn cube(x) =
  letfn square(x) = x * x in
  square(x) * x in
cube(a)
```

```
letfn cube(x) = square(x) * x in
letfn square(x) = x * x in
letfn main() = let a = 2 in cube(a) in
  main()
```

Overview: whole program

- After preprocess: the program becomes a list of functions $[main, f_1, \cdots, f_n]$
- For the whole program:

$$\begin{aligned}\textbf{Prog}\,[\![prog]\!] &= \textbf{Prog}\,[\![main, f_1, \cdots, f_n]\!] \\ &= \mathsf{Call}(\mathrm{main}, 0)\ ;\ \mathsf{Exit}\ ;\ \textbf{Fun}\,[\![main]\!]\ ;\ \textbf{Fun}\,[\![f_1]\!]\ ;\ \cdots\ ;\ \textbf{Fun}\,[\![f_n]\!]\end{aligned}$$

- Declare type synonyms

```
type fun = (string, list<string>, expr)
type prog = list<fun>
```

Overview: functions

- For functions:

$$\mathbf{Fun}\,[\![(\mathrm{f}, [p_1, \cdots, p_n], e)]\!] = \mathrm{Label}(f)\ ;\ \mathbf{Expr}[\![e]\!]_{p_n, \cdots, p_1}\ ;\ \mathrm{Ret}(n)$$

Overview: expressions

For expression:

$$
\begin{aligned}
\mathbf{Expr}[\![\mathsf{Cst}(i)]\!]_s &= \mathrm{Cst}(i) \\
\mathbf{Expr}[\![\mathsf{Var}(x)]\!]_s &= \mathrm{Var}(\mathsf{get\_index}(\mathsf{x}, \mathsf{s})) \\
\mathbf{Expr}[\![\mathsf{Let}(x, e_1, e_2)]\!]_s &= \mathbf{Expr}[\![e_1]\!]_s \ ; \mathbf{Expr}[\![e_2]\!]_{x::s} \ ; \mathrm{Swap} \ ; \mathrm{Pop} \\
\mathbf{Expr}[\![\mathsf{Prim}(p, [e_1, \cdots, e_n])]\!]_s &= \cdots \\
\mathbf{Expr}[\![\mathsf{App}(f, [a_1, \cdots, a_n])]\!]_s &= \mathbf{Exprs}[\![a_1, \cdots, a_n]\!]_s \ ; \mathrm{Call}(f, n) \\
\mathbf{Expr}[\![\mathsf{If}(cond, e_1, e_2)]\!]_s &= \cdots
\end{aligned}
$$

and expressions

$$
\mathbf{Exprs}[\![a_1, \cdots, a_n]\!]_s = \mathbf{Expr}[\![a_1]\!]_s \ ; \mathbf{Expr}[\![a_2]\!]_{*::s} \ ; \cdots
$$

Note: $*$ is a placeholder for the temperary variable in the compile env

Overview: expressions

$$
\begin{aligned}
\mathbf{Expr}[\![\mathsf{Cst}(i)]\!]_s &= \mathrm{Cst}(i) \\
\mathbf{Expr}[\![\mathsf{Var}(x)]\!]_s &= \mathrm{Var}(\mathsf{get\_index}(x, s)) \\
\mathbf{Expr}[\![\mathsf{Let}(x, e_1, e_2)]\!]_s &= \mathbf{Expr}[\![e_1]\!]_s \;;\; \mathbf{Expr}[\![e_2]\!]_{x::s} \;;\; \mathrm{Swap} \;;\; \mathrm{Pop} \\
\mathbf{Expr}[\![\mathsf{Prim}(p, [e_1, \cdots, e_n])]\!]_s &= \cdots \\
\mathbf{Expr}[\![\mathsf{App}(f, [a_1, \cdots, a_n])]\!]_s &= \mathbf{Exprs}[\![a_1, \cdots, a_n]\!]_s \;;\; \mathrm{Call}(f, n) \\
\mathbf{Expr}[\![\mathsf{If}(cond, e_1, e_2)]\!]_s &= \cdots
\end{aligned}
$$

what happens to `Letfn`?

```
type rec expr =
  | Cst (int)
  | Var (string)
  | Let (string, expr, expr)
  | Letfn (string, list<string>,
           expr, expr)
  | App (string, list<expr>)
  | Prim (prim, list<expr>)
  | If (expr, expr, expr)
```

```
module Flat = {
  type rec expr =
    | Cst (int)
    | Var (string)
    | Let (string, expr, expr)
    | App (string, list<expr>)
    | Prim (prim, list<expr>)
    | If (expr, expr, expr)
}
```

Implementation overview

```
type fun  = (string, list<string>, Flat.expr)
type prog = list<fun>
// preprocessing function to collect function definitions
let preprocess = (expr: expr) : list<fun> => { ... }
// compile the whole program
let compile = (prog: prog) : list<instr> { ... }
// compile functions
let compile_fun = ((name, args, body): fun) : list<instr> => { ... }
// compile expression under a compile-time environment
let compile_expr = (env:env, expr: Flat.expr) : list<instr> => { ... }
let compile_exprs = (env: env, exprs: list<Flat.expr>) : list<instr> => { ... }
```

$\mathbf{Prog}[\![-]\!]$ corresponds to `compile: prog => list<instr>`

$\mathbf{Fun}[\![-]\!]$ corresponds to `compile_fun: fun => list<instr>`

$\mathbf{Expr}[\![-]\!]_s$ corresponds to `compile_expr: (env, Flat.expr) => list<instr>`

$\mathbf{Exprs}[\![-]\!]_s$ to `compile_exprs: (env, list<Flat.expr>) => list<instr>`

1. Flatten the code

```
// Auxiliary functions
let rec collect_funs = (expr: expr): list<fun> => { ... }
let rec remove_funs = (expr: expr): Flat.expr => { ... }
// Preprocessing
let preprocess = (expr: expr): list<fun> => {
  let main = ("main", list{}, remove_funs(expr)) // what if we have cmd line args?
  let rest = collect_funs(expr)
  list{ main, ...rest }
}
```

For example,

```
[(main, [], let a = 2 in cube(a)),
 (square, [x], x * x),
 (cube, [x], square(x) * x)]
```

2. Callee: Find arguments

- According to the *convention*, the arguments are pushed to the stack by the caller
- Recall the other kinds of variables we have met so far: local and temp

For example, suppose we have a function after preprocessing

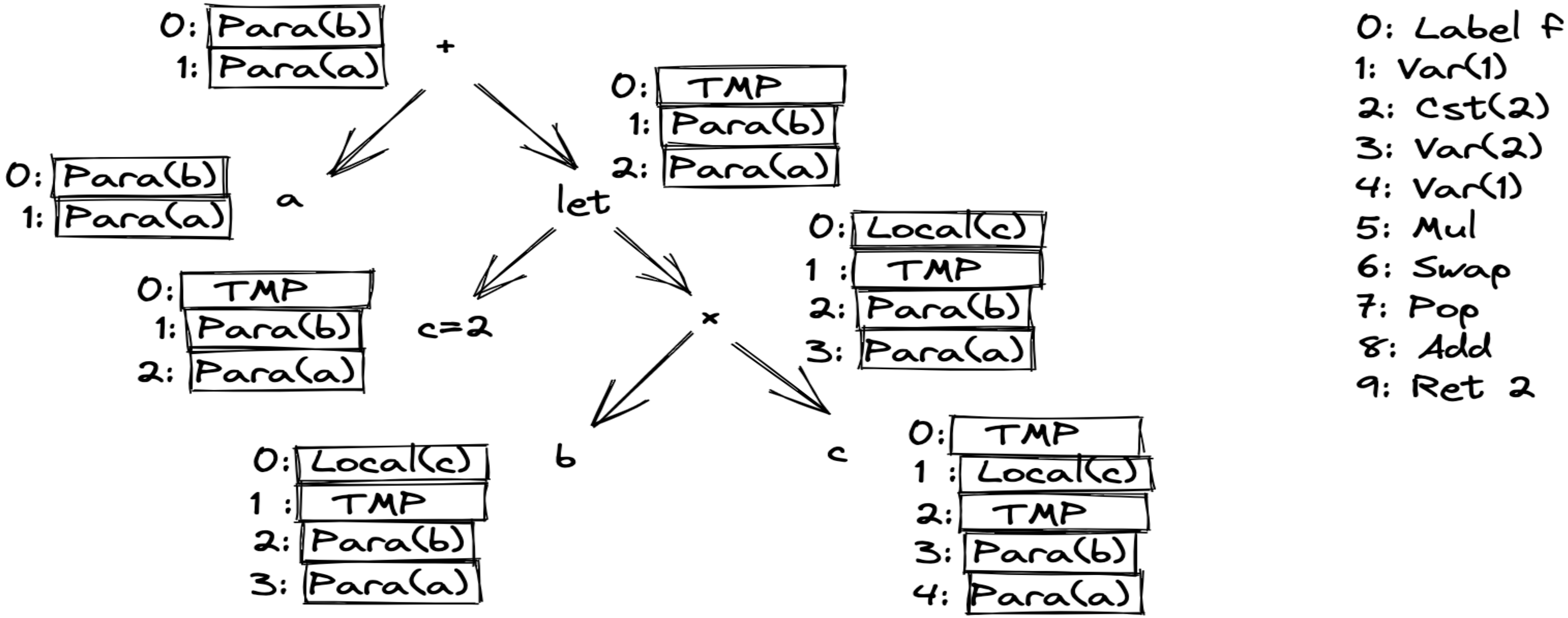
```
(f, [a, b], a + let c = 2 in b * c)
```

We need to resolve the variable references:

```
(f, [a, b], Var(?) + let c = 2 in Var(?) * Var(?))
```

2. Callee: Find arguments

```
(f, [a, b], a + let c = 2 in b * c)
```



- Observe that there's no difference between `Para` and `Local`

2. Callee: Label and Return

```
type var = Local(string) | Temp // Params and locals are treated uniformly
let compile_fun = ((name, args, body): fun): list<instr> => {
  let n = List.length(args)
  let env = List.rev(List.map((a) => Local(a), args))
  list{
    Label(name),
    ...compile_expr(env, body),
    Ret(n),
  }
}
```

For example,

```
[
  [Label(main); compile_expr([], let a = 2 in cube(a)); Ret(0)];
  [Label(square); compile_expr([x], x * x); Ret(1)];
  [Label(cube); compile_expr([x], square(x) * x); Ret(1)]
]
```

idea 基础软件中心

2. Caller: Push arguments and make the call

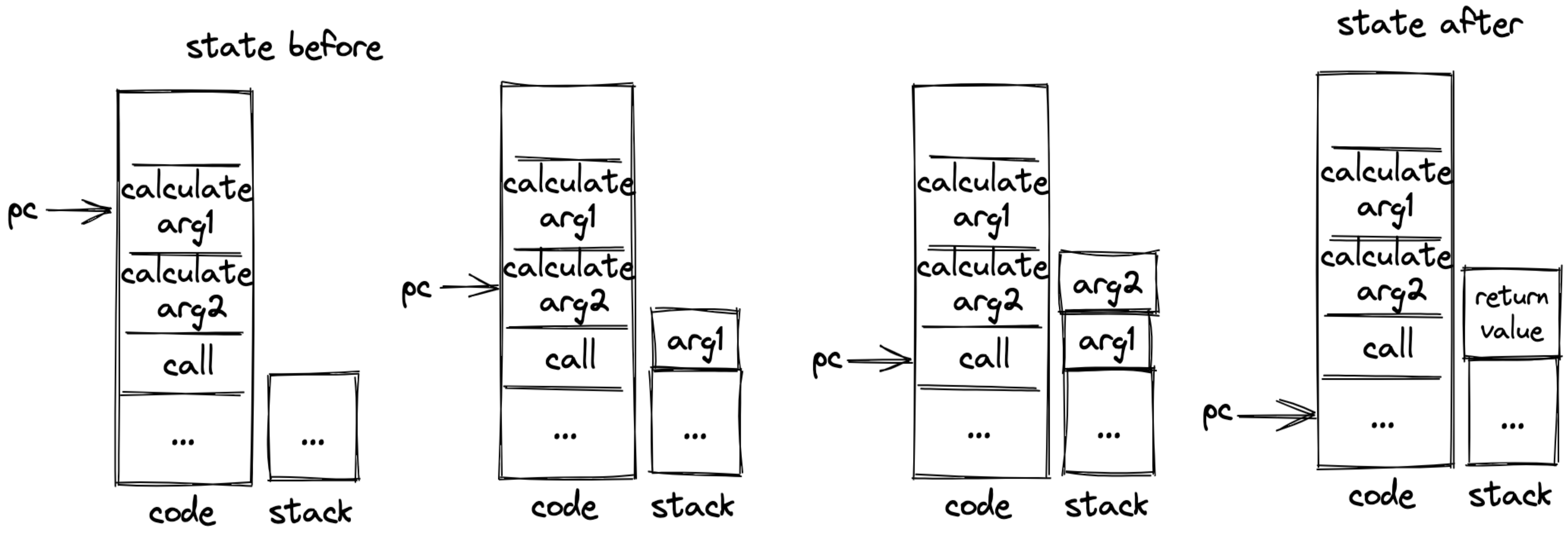
```
let rec compile_expr = (env: env, expr: expr): list<instr> => {
  switch expr {
  | ...
  | App(fn, args) => {
    let n = List.length(args)
    let args_code = compile_exprs(env, args)
    list{...args_code, Call(fn, n)}
  }
}
and compile_exprs = (env: env, exprs: list<expr>): list<instr> => { ... }
```

Recall the formalization:

$$\mathbf{Expr}[\![\mathrm{App}(f, [a_1, \cdots, a_n])]\!]_s = \mathbf{Exprs}[\![a_1, \cdots, a_n]\!]_s \ ; \mathrm{Call}(f, n)$$

$$\mathbf{Exprs}[\![a_1, \cdots, a_n]\!]_s = \mathbf{Expr}[\![a_1]\!]_s \ ; \mathbf{Expr}[\![a_2]\!]_{*::s} \ ; \cdots$$

2. Caller: Push arguments and make the call



3. Entrance and exit

```
let compile = (expr: expr) => {
  let funs = preprocess(expr)
  let funs_code = List.concat(List.map(compile_fun, funs))

  list{
    Call("main", 0),
    Exit,
    ...funs_code
  }
}
```

3. Entrance and exit

idea
基础软件中心

```
let a = 2 in
letfn cube(x) =
  letfn square(x) = x * x in
  square(x) * x in
cube(a)
```

```
0: Call main 0
1: Exit

2: Label main
3: Cst(2)
4: Var(0)
5: Call cube 1
6: Swap
7: Pop
8: Ret 0

9: Label cube
10: Var(0)
11: Call square 1
12: Var(1)
13: Mul
14: Ret 1

15: Label square
16: Var(0)
17: Var(1)
18: Mul
19: Ret 1
```

Summary

- For the whole program:

$$\begin{aligned}\mathbf{Prog}\,[\![prog]\!] &= \mathbf{Prog}\,[\![main, f_1, \cdots, f_n]\!] \\ &= \mathsf{Call}(\mathrm{main}, 0)\ ;\ \mathsf{Exit}\ ;\ \mathbf{Fun}\,[\![main]\!]\ ;\ \mathbf{Fun}\,[\![f_1]\!]\ ;\ \cdots\ ;\ \mathbf{Fun}\,[\![f_n]\!]\end{aligned}$$

- For functions:

$$\mathbf{Fun}\,[\![(\mathrm{f}, [p_1, \cdots, p_n], e)]\!] = \mathrm{Label}(f)\ ;\ \mathbf{Expr}[\![e]\!]_{p_n, \cdots, p_1}\ ;\ \mathrm{Ret}(n)$$

- For expression:

$$\mathbf{Expr}[\![\mathsf{App}(f, [a_1, \cdots, a_n])]\!]_s = \mathbf{Exprs}[\![a_1, \cdots, a_n]\!]_s\ ;\ \mathrm{Call}(f, n)$$

$$\cdots$$

- and expressions

$$\mathbf{Exprs}[\![a_1, \cdots, a_n]\!]_s = \mathbf{Expr}[\![a_1]\!]_s\ ;\ \mathbf{Expr}[\![a_2]\!]_{*::s}\ ;\ \cdots$$

idea
基础软件中心

Summary

- Calling convention: interface between caller/callee
- Stack balance property
- Stack frames to store *metadata*, parameters, local, and temp
- Type enforced invariants

From tiny language 4

- Pascal-like functions: allow free variables but closures may not escape

For example[1],

```
function () {
    var a = 42;
    var f = function () { return a + 1; }
    foo(f); // `foo` is a function declared somewhere else.
}
```

- full-blown first-class function

```
function () {
    var a = 42;
    var f = function () { return a + 1; }
    return f;
}
```

Homework

- Complete the compiler
- Implement an interpreter that supports recursive functions

```
letfn fib(n) =
  if n <= 1 then { 1 }
  else { fib(n-1) + fib(n-2) }
in fib(5)
```