MoonBit 月兔编程

现代编程思想

案例：栈式虚拟机

Hongbo Zhang

编译与解释

- 编译
  - 源程序 x 编译器 -> 目标程序
  - 目标程序 x 输入数据 -> 输出数据
- 解释
  - 源程序 x 输入数据 x 解释器 -> 输出数据
  - CPU可以被视为广义上的解释器
- 拓展阅读：二村映射/部分计算
  - 部分计算：程序优化，根据已知信息，运算进行特化
  - 已知源程序与解释器，进行部分运算，获得目标程序
    - 目标程序 x 输入数据 -> 输出数据

虚拟机

- 一处编写，处处运行
  - 定义一个不基于任何平台的指令集
  - 在不同平台上实现解释器
- 两种常见的虚拟机
  - 堆栈虚拟机：运算数存储在栈上，数据遵循先进后出原则
  - 寄存器虚拟机：运算数存储在寄存器中

MoonBit 月兔编程

寄存器虚拟机

- 例：Lua VM (*The Implementation of Lua 5.0*)
  - 取最大值

```
MOVE   2 0 0 ; R(2) = R(0)
LT     0 0 1 ; R(0) < R(1)?
JMP    1     ; JUMP -> 5 (4 + 1)
MOVE   2 1 0 ; R(2) = R(1)
RETURN 2 2 0 ; return R(2)
RETURN 0 1 0 ; return
```

MoonBit 月兔编程

堆栈虚拟机

- 例：WebAssembly Virtual Machine
  - 取最大值 `fn max(a : Int, b : Int) -> Int`

```
1. local.get $a local.set $m                    ;; let mut m = a
2. local.get $a local.get $b i32.lt_s           ;; if a < b {
3. if local.get $b local.set $m end             ;; m = b }
4. local.get $m                                 ;; m
```

MoonBit 月兔编程

WebAssembly

- WebAssembly是什么?
  - 一个虚拟指令集
  - 可以在浏览器以及其他运行时（Wasmtime WAMR WasmEdge等）中运行
  - MoonBit的第一个后端
- WebAssembly指令集的子集为例

MoonBit 月兔编程

简单指令集

- 数据
  - 只考虑32位有符号整数
  - 非零代表`true`，零代表`false`
- 指令
  - 数据操作：`const` `add` `minus` `equal` `modulo`
  - 数据存储：`local.get` `local.set`
  - 控制流：`if/else` `call`

类型定义

- 数据

```
enum Value { I32(Int) } // 只考虑32位有符号整数
```

- 指令

```
enum Instruction {
  Const(Value)                                  // 常数
  Add; Sub; Modulo; Equal                       // 加、减、取模、相等
  Call(String)                                  // 函数调用
  Local_Get(String); Local_Set(String) // 取值、设值
  If(Int, List[Instruction], List[Instruction]) // 条件判断
}
typealias @list.T as List
```

类型定义

- 函数

```
1. struct Function {
2.   name : String
3.   // 只考虑一种数据类型，故省略每个数据的类型，只保留名称和数量
4.   params : List[String]; result : Int; locals : List[String]
5.   instructions : List[Instruction]
6. }
```

- 程序

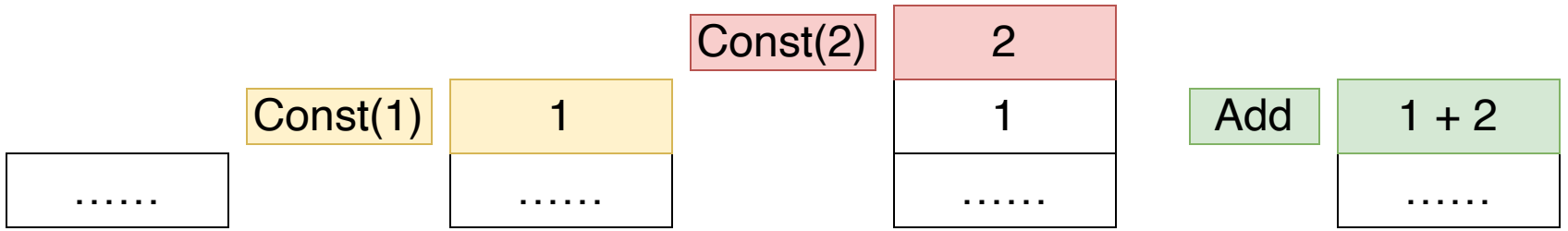
```
1. struct Program {
2.   functions : List[Function]
3.   start : Option[String]
4. }
```

MoonBit 月兔编程

简单计算

- 例：`1 + 2`

```
1. List::[ Const(1), Const(2), Add ]
```

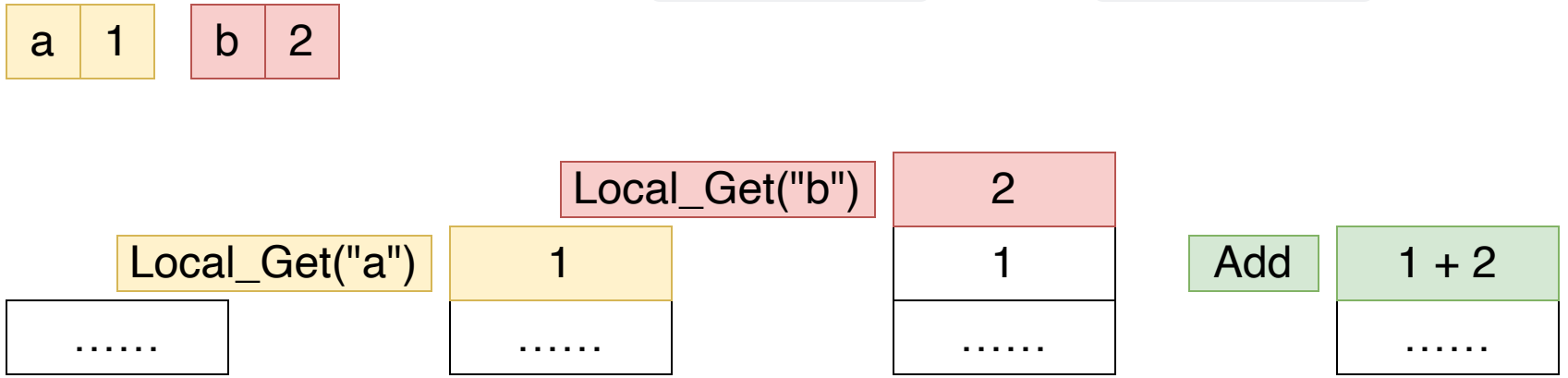


本地变量使用

- 例：`fn add(a : Int, b : Int) { a + b }`

```
1. List::[ Local_Get("a"), Local_Get("b"), Add ]
```

  - 函数参数及本地变量可通过`Local_Get`获取、`Local_Set`修改

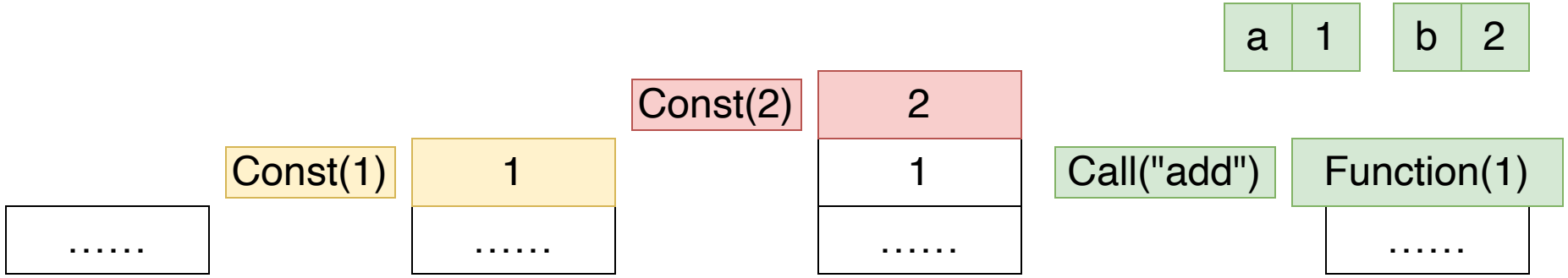


函数计算

- 例：`add(1, 2)`

```
1. Lists::[ Const(1), Const(2), Call("add") ]
```

  - 在栈上存储函数的返回值数量

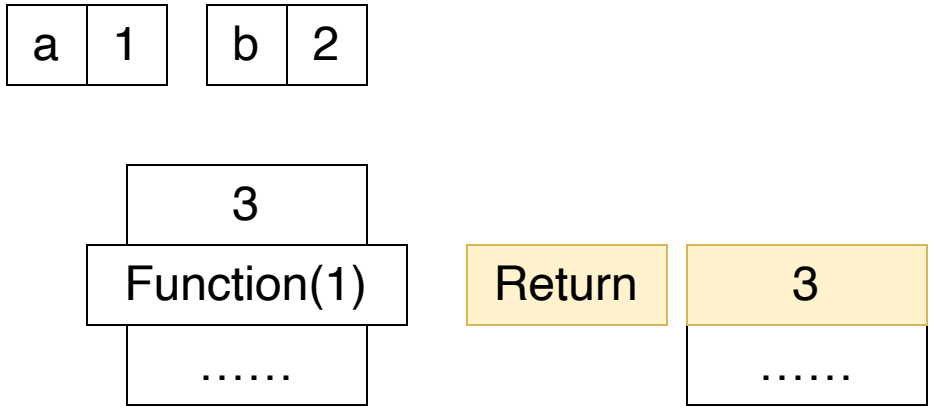


函数计算

- 例：`add(1, 2)`

```
1. @list.from_array([ Const(1), Const(2), Call("add") ])
```

  - 在栈上存储函数的返回值数量



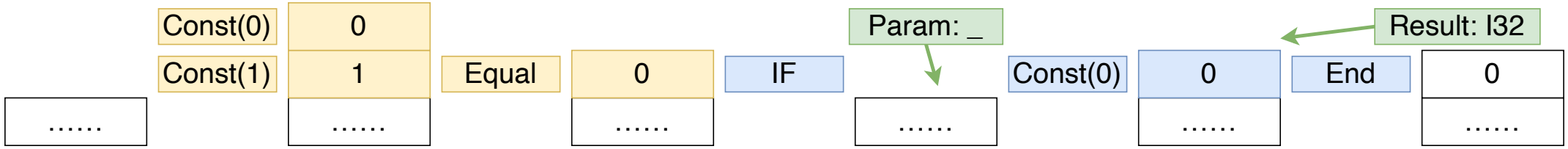
条件跳转

MoonBit 月兔编程

- 例：`if 1 == 0 { 1 } else { 0 }`

```
1. @list.from_array([
2.   Const(1), Const(0), Equal,
3.   If(1, List::[Const(1)], List::[Const(0)])
4. ])
```

  - 当执行`If(n, then_block, else_block)`时栈顶为非0整数（`true`）
    - 执行then_block
    - 否则，执行else_block
  - `n`表示代码块返回的参数数量



例：加法

```
let program : Program = Program::{

  start: Some("test_add"), // 程序入口

  functions: @list.of([
    Function::{
      name: "add", // 加法函数
      params: @list.of(["a", "b"]), result: 1, locals: @list.of([]),
      instructions: @list.of([Local_Get("a"), Local_Get("b"), Add]),
    },

    Function::{
      name: "test_add", // 加法运算并输出
      params: @list.of([]), result: 0, locals: @list.of([]), // 入口函数无输入无输出
      // “print_int"为特殊函数
      instructions: @list.of([Const(I32(0)), Const(I32(1)), Call("add"), Call("print_int")]),
    },
  ]),
}
```

程序整体架构

- 输出程序

```
;; 多个函数
;; WASM本身只定义运算；交互需依赖外部函数
(func $print_int (import "spectest" "print_int") (param i32))

(func $add (export "add") ;; 导出函数使运行时可以直接使用
  (param $a i32) (param $b i32) (result i32 ) ;; (a : Int, b : Int) -> Int
  local.get $a local.get $b i32.add ;;
)

(func $test_add (export "test_add") (result ) ;; 入口函数无输入无输出
  i32.const 0 i32.const 1 call $add call $print_int
)

(start $test_add)
```

在线尝试

- https://webassembly.github.io/wabt/demo/wat2wasm/

```
1. const wasmInstance = new WebAssembly.Instance(wasmModule, {"spectest":{"print_int": console.log}});
```

MoonBit 月兔编程

构造编译器

指令	WebAssembly指令
`Const(0)`	`i32.const 0`
`Add`	`i32.add`
`Local_Get("a")`	`local.get $a`
`Local_Set("a")`	`local.set $a`
`Call("add")`	`call $add`
`If(1, List::[Const(0)], List::[Const(1)])`	`if (result i32) i32.const 0 else i32.const 1 end`

编译程序

- 利用内建 `StringBuilder` 数据结构，比字符串拼接更高效

```
1. fn Instruction::to_wasm(self : Instruction, buffer : StringBuilder) -> Unit
2. fn Function::to_wasm(self : Function, buffer : StringBuilder) -> Unit
3. fn Program::to_wasm(self : Program, buffer : StringBuilder) -> Unit
```

编译指令

- 利用内建 `StringBuilder` 数据结构，比拼接字符串更高效

```
1. fn Instruction::to_wasm(self : Instruction, buffer : StringBuilder) -> Unit {
2.   match self {
3.     Add => buffer.write_string("i32.add ")
4.     Local_Get(val) => buffer.write_string("local.get $\{val} ")
5.     _ => buffer.write_string("...")
6.   }
7. }
```

MoonBit 月兔编程

Wasm的二进制格式

文本格式	二进制格式
`i32.const`	0x41
`i32.add`	0x6A
`local.get`	0x20
`local.set`	0x21
`call $add`	0x10
`if else end`	0x04 (vec[instructions]) 0x05 (vec[instructions]) 0x0B

MoonBit 月兔编程

多层编译

- 字符串 -> 单词流 -> (抽象语法树) -> WASM IR -> 运行

```
enum Expression {
  Number(Int)
  Plus(Expression, Expression)
  Minus(Expression, Expression)
  Multiply(Expression, Expression)
  Divide(Expression, Expression)
}
```

MoonBit 月兔编程

多层编译

- 字符串 -> 单词流 -> (抽象语法树) -> WASM IR -> 编译/运行

```
fn compile_expression(expression : Expression) -> List[Instruction] {
  match expression {
    Number(i) => @list.of([Const(I32(i))])
    Plus(a, b) => compile_expression(a) + compile_expression(b) + @list.of([Add])
    Minus(a, b) => compile_expression(a) + compile_expression(b) + @list.of([Sub])
    _ => @list.of([])
  }
}
```

MoonBit 月兔编程

构建解释器

- 一种可能的解释器
- 操作数栈
  - 参与运算的数值
  - 函数执行前的本地变量
- 指令队列
  - 当前执行的指令
  - 分为普通指令和控制指令（如函数结束时的返回）

解释器结构

MoonBit 月兔编程

```
enum StackValue {
  Val(Value) // 普通数值
  Func(@immut/hashmap.T[String, Value]) // 函数堆栈，存放先前的本地变量
}

enum AdministrativeInstruction {
  Plain(Instruction) // 普通指令
  EndOfFrame(Int) // 函数结束指令
}

struct State {
  program : Program
  stack : List[StackValue]
  locals : @immut/hashmap.T[String, Value]
  instructions : List[AdministrativeInstruction]
}
```

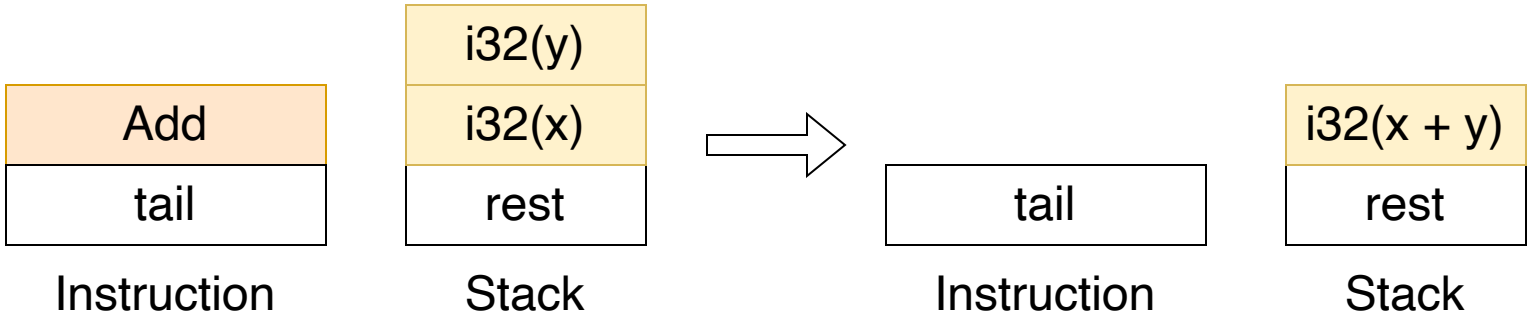
- 状态迭代

```
fn evaluate(state : State, stdout : Buffer) -> Option[State]
```

解释器构造

- 读取当前指令与栈顶数据

```
1. fn evaluate(state : State, stdout : StringBuilder) -> Option[State] {
2.   match (state.instructions, state.stack) {
3.     (More(Plain(Add), tail=tl), More(Val(I32(b)), tail=More(Val(I32(a)), tail=rest))) =>
4.       Some(
5.         State::{ ..state, instructions: tl, stack: @list.construct(Val(I32(a + b)), rest) },
6.       )
7.     _ => None
8.   }
9. }
```

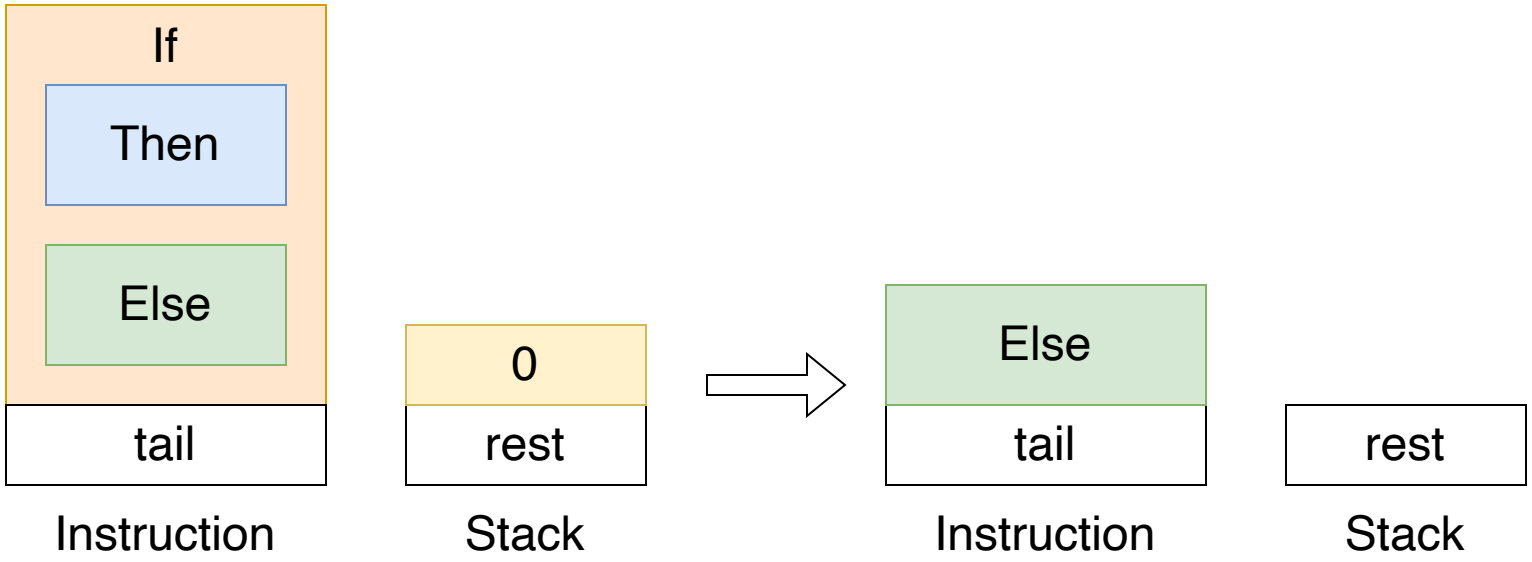


解释器构造

MoonBit 月兔编程

- 条件判断时，根据分支取出对应代码

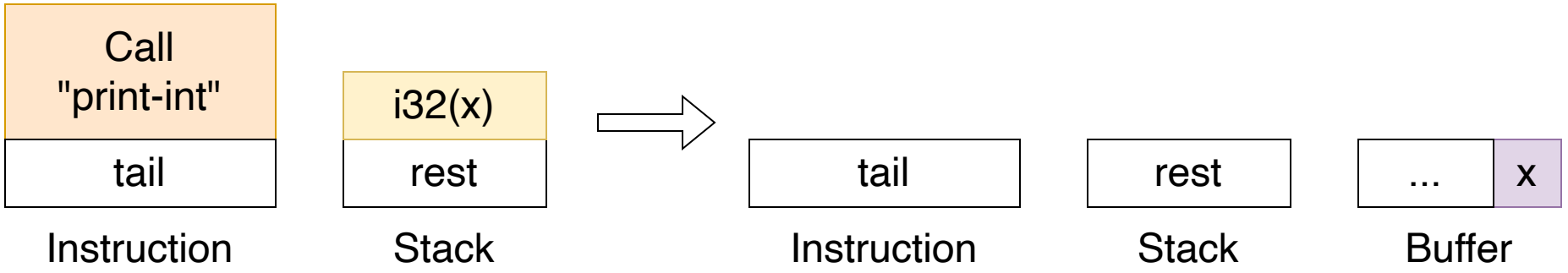
```
1. (More(Plain(If(_, then, else_)), tail=tl), More(Val(I32(i)), tail=rest)) =>
2.   Some(State::{..state,
3.        stack: rest,
4.        instructions: (if i != 0 { then } else { else_ }).map(
5.          AdministrativeInstruction::Plain,
6.        ).concat(tl)})
```



解释器构造

- 对输出函数进行特判

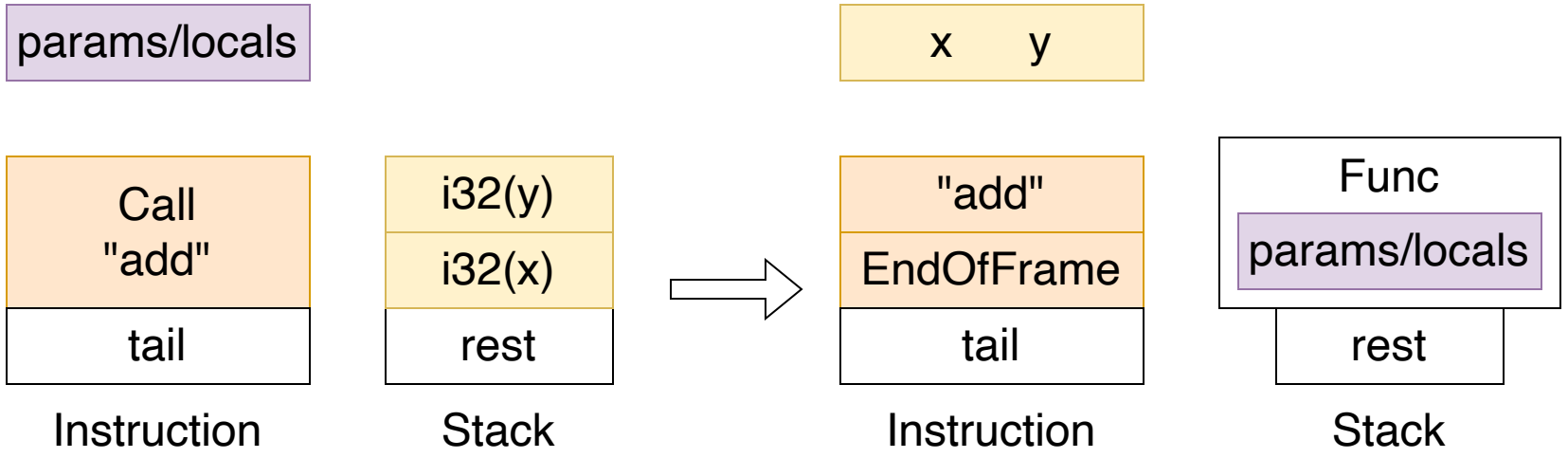
```
1. (More(Plain(Call("print_int")), tail=tl), More(Val(I32(i)), tail=rest)) => {
2.   stdout.write_string(i.to_string())
3.   Some(State::{ ..state, stack: rest, instructions: tl })
4. }
```



MoonBit 月兔编程

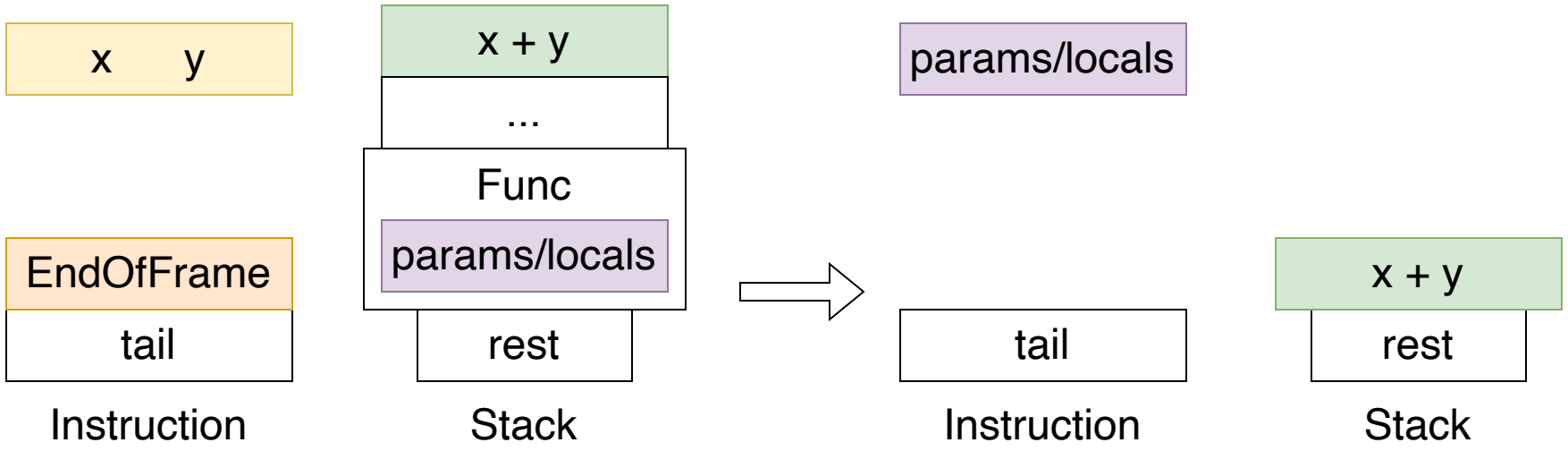
解释器构造

- 对于函数调用，将当前的运行状态（变量）存入堆栈



解释器构造

- 在函数执行完成后
  - 从栈顶根据返回值数量取出元素
  - 将调用栈的信息展开



总结

- 本节课展示了一个堆栈虚拟机
  - 介绍了WebAssebmly指令集的一小部份
  - 实现了一个编译器
  - 实现了一个解释器
- 挑战
  - 在语法解析器中拓展函数定义
  - 在指令集中添加提前返回指令（`return`）