

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

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
```

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
```

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

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

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

- 指令

```
1. enum Instruction {  
2.     Const(Int) // 常数  
3.     Add; Sub; Modulo; Equal // 加、减、取模、相等  
4.     Call(String) // 函数调用  
5.     Local_Get(String); Local_Set(String) // 取值、设值  
6.     If(Int, List[Instruction], List[Instruction]) // 条件判断  
7. }
```


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. }
```

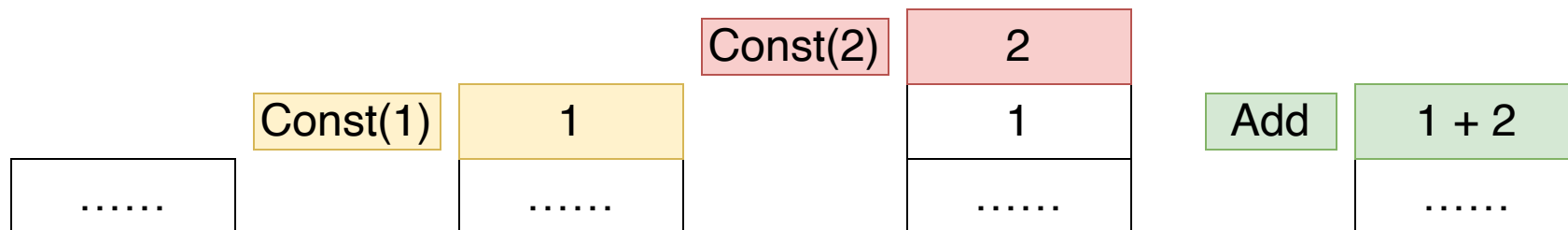
- 程序

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

简单计算

- 例: `1 + 2`

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



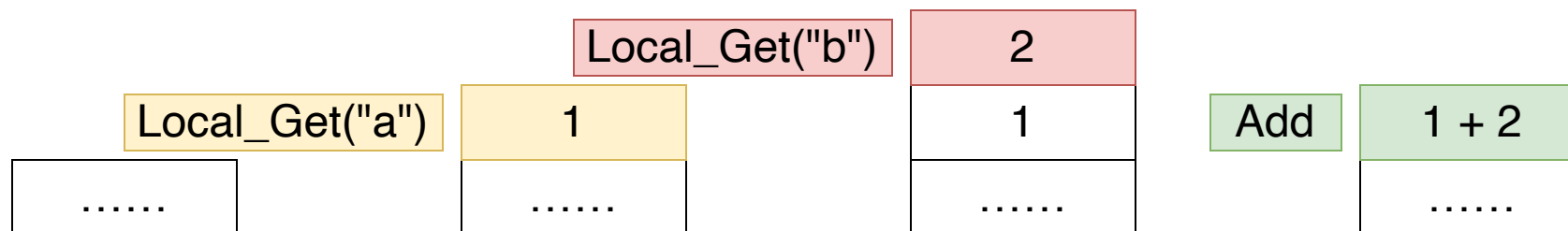
本地变量使用

• 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

a	1	b	2
---	---	---	---

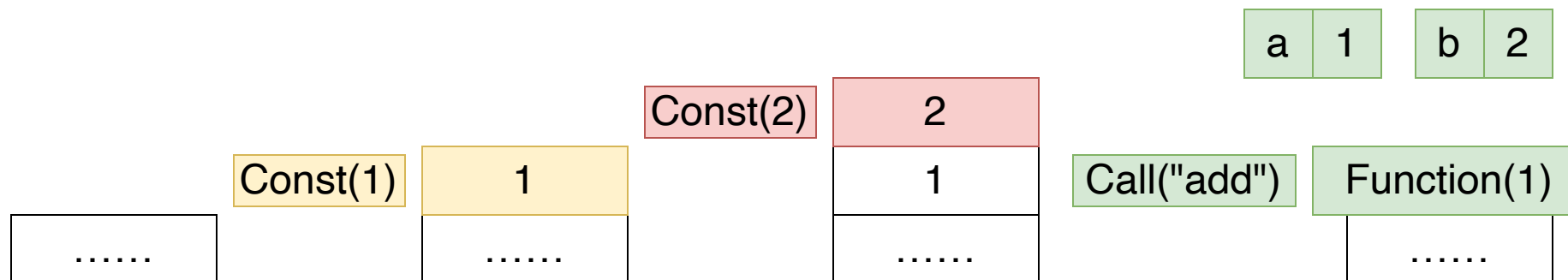


函数计算

- 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

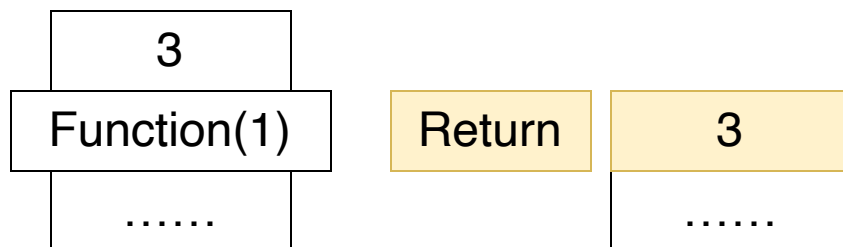
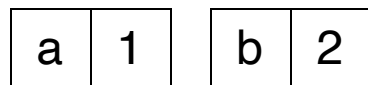


函数计算

- 例: `add(1, 2)`

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

- 在栈上存储函数的返回值数量

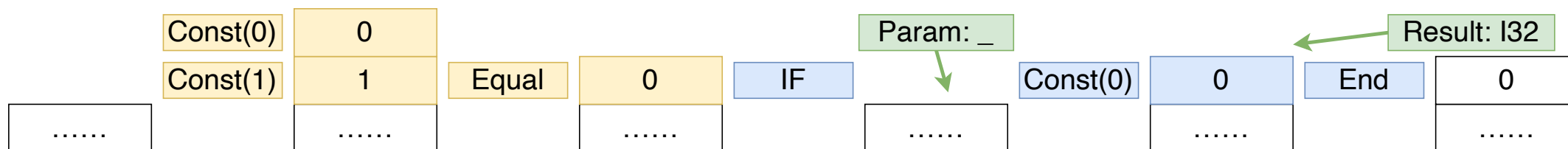


条件跳转

- 例: `if 1 == 0 { 1 } else { 0 }`

```
1. List::[  
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

```
1. let program = Program::{
2.
3.   start: Some("test_add"), // 程序入口
4.
5.   functions: List::[
6.     Function::{
7.       name: "add", // 加法函数
8.       params: List::["a", "b"], result: 1, locals: List::[],
9.       instructions: List::[Local_Get("a"), Local_Get("b"), Add],
10.    },
11.
12.    Function::{
13.      name: "test_add", // 加法运算并输出
14.      params: List::[], result: 0, locals: List::[], // 入口函数无输入无输出
15.      // "print_int"为特殊函数
16.      instructions: List::[Const(0), Const(1), Call("add"), Call("print_int")],
17.    },
18.  ],
19. }
```

程序整体架构

- 输出程序

```
1. ;; 多个函数
2. ;; WASM本身只定义运算；交互需依赖外部函数
3. (func $print_int (import "spectest" "print_int") (param i32))
4.
5. (func $add (export "add") ;; 导出函数使运行时可以直接使用
6.   (param $a i32) (param $b i32) (result i32) ;; (a : Int, b : Int) -> Int
7.   local.get $a local.get $b i32.add ;;
8. )
9.
10. (func $test_add (export "test_add") (result ) ;; 入口函数无输入无输出
11.   i32.const 0 i32.const 1 call $add call $print_int
12. )
13.
14. (start $test_add)
```


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

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

wat2wasm demo

WebAssembly has a [text format](#) and a [binary format](#). This demo converts from the text format to the binary format.

Enter WebAssembly text in the textarea on the left. The right side will either show an error, or will show a log with a descriptive message.

Enabled features:

exceptions mutable globals saturating float to int sign extension simd threads function references
 bulk memory reference types annotations code metadata gc memory64 multi memory extended co

```
WAT example: simple Download
1 ;; 多个函数
2 ;; WASM本身只定义运算: 交互需依赖外部函数
3 (func $print_int (import "spectest" "print_int") (param i32))
4
5 (func $add (export "add") ;; 导出函数使运行时可以直接使用
6   (param $a i32) (param $b i32) (result i32) ;; (a : Int, b : Int) -> Int
7   local.get $a local.get $b i32.add ;;
8 )
9
10 (func $test_add (export "test_add") (result ) ;; 入口函数无输入无输出
11   i32.const 0 i32.const 1 call $add call $print_int
12 )
13
14 (start $test_add)

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

构造编译器

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

编译程序

- 利用内建 Buffer 数据结构，比字符串拼接更高效

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

编译指令

- 利用内建 Buffer 数据结构，比拼接字符串更高效

```
1. fn Instruction::to_wasm(self : Instruction, buffer : Buffer) -> 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. }
```

Wasm的二进制格式

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

多层编译

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

```
1. enum Expression {  
2.     Number(Int)  
3.     Plus(Expression, Expression)  
4.     Minus(Expression, Expression)  
5.     Multiply(Expression, Expression)  
6.     Divide(Expression, Expression)  
7. }
```

多层编译

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

```
1. fn compile_expression(expression : Expression) -> List[Instruction] {
2.   match expression {
3.     Number(i) => List::[Const(I32(i))]
4.     Plus(a, b) => compile_expression(a) + compile_expression(b) + List::[Add]
5.     Minus(a, b) => compile_expression(a) + compile_expression(b) + List::[Sub]
6.     _ => List::[]
7.   }
8. }
```

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）


```
1. enum StackValue {
2.     Val(Value) // 普通数值
3.     Func(@map.Map[String, Value]) // 函数堆栈, 存放先前的本地变量
4. }
5.
6. enum AdministrativeInstruction {
7.     Plain(Instruction) // 普通指令
8.     EndOfFrame(Int) // 函数结束指令
9. }
10.
11. struct State {
12.     program : Program
13.     stack : List[StackValue]
14.     locals : @map.Map[String, Value]
15.     instructions : List[AdministrativeInstruction]
16. }
```

- 状态迭代

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

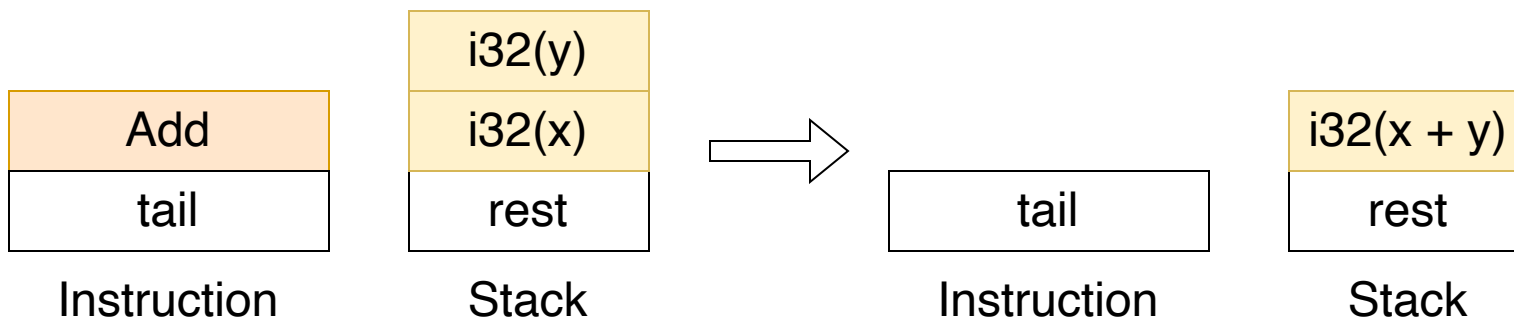
解释器构造

- 读取当前指令与栈顶数据

```

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

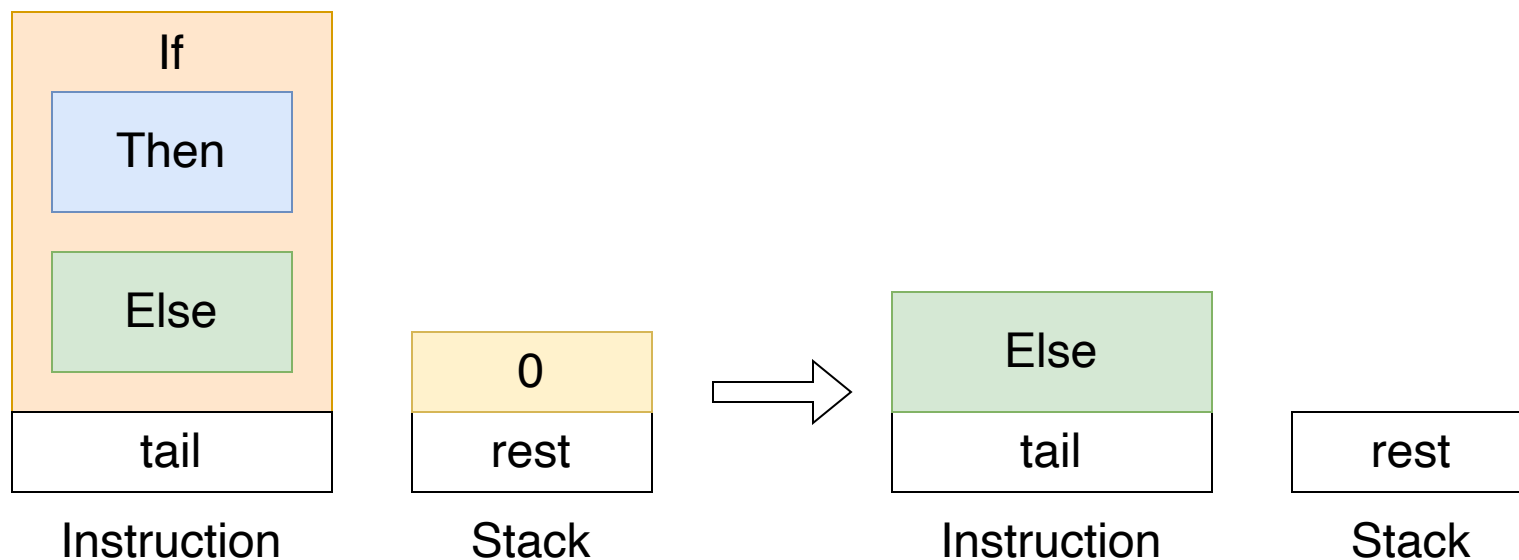
```



解释器构造

- 条件判断时，根据分支取出对应代码

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



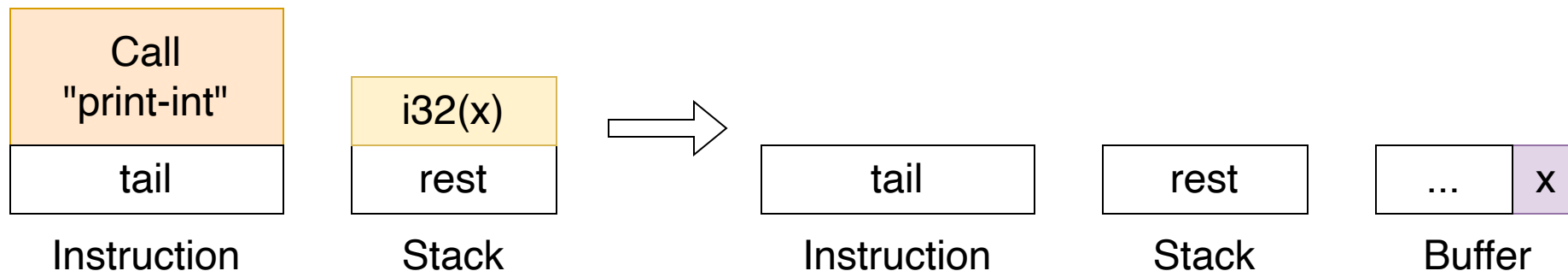
解释器构造

- 对输出函数进行特判

```

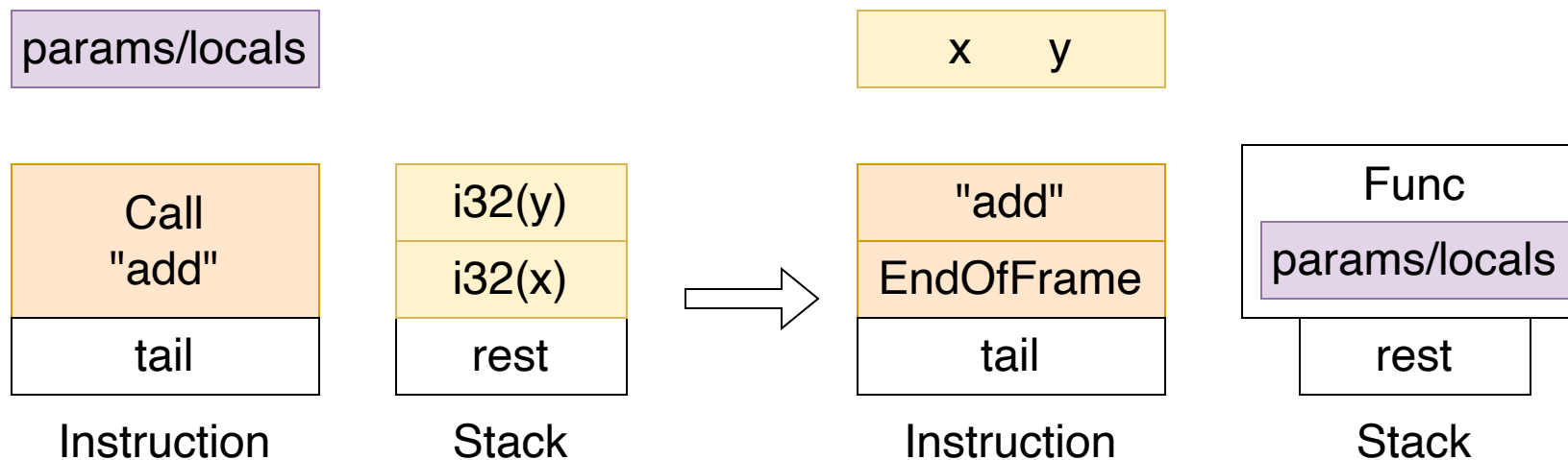
1. (Cons(Plain(Call("print_int")), tl), Cons(Val(I32(i)), rest)) => {
2.   stdout.write_string(i.to_string())
3.   Some(State::{ ..state, stack: rest, instructions: tl })
4. }

```



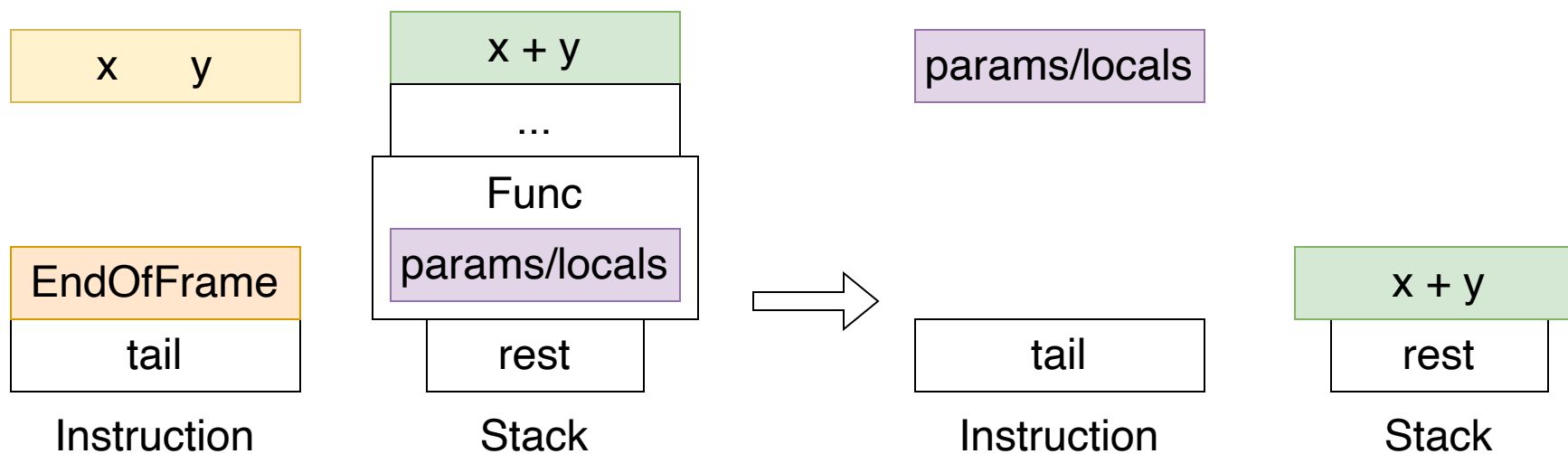
解释器构造

- 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WebAssembly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`)