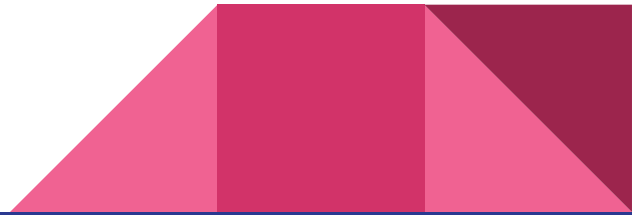
JIT 编译技术

Heyang Zhou

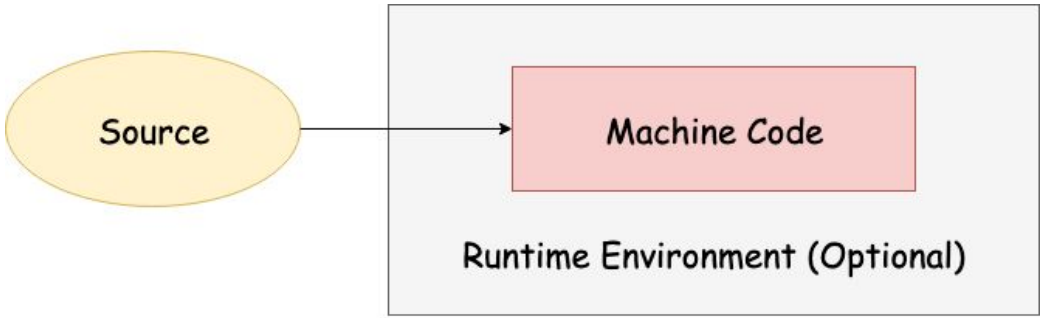
May 25, 2019

JIT 简介

- 运行时编译
- 现代编程语言实现**动态性**与**安全性**，并**减少性能损失**的重要技术

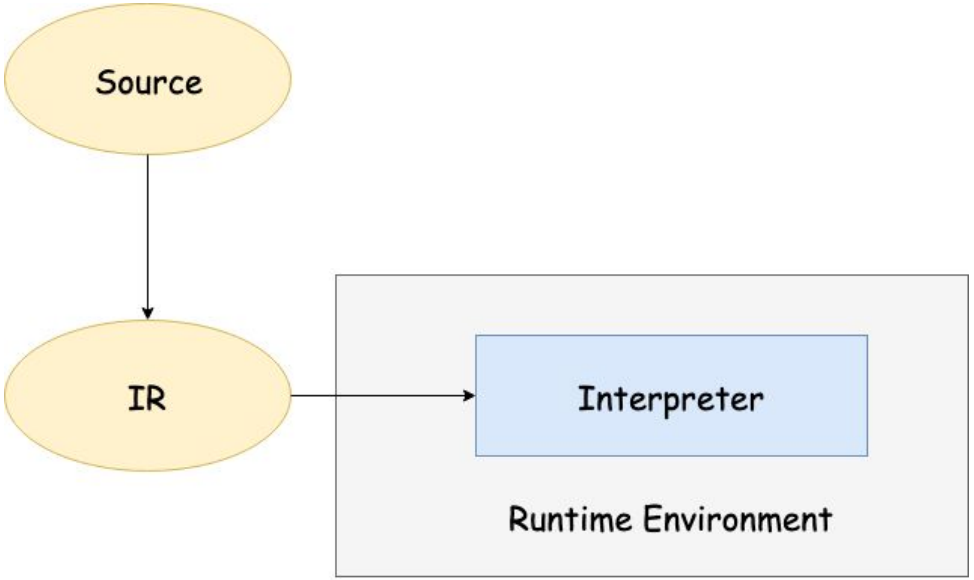


JIT 简介



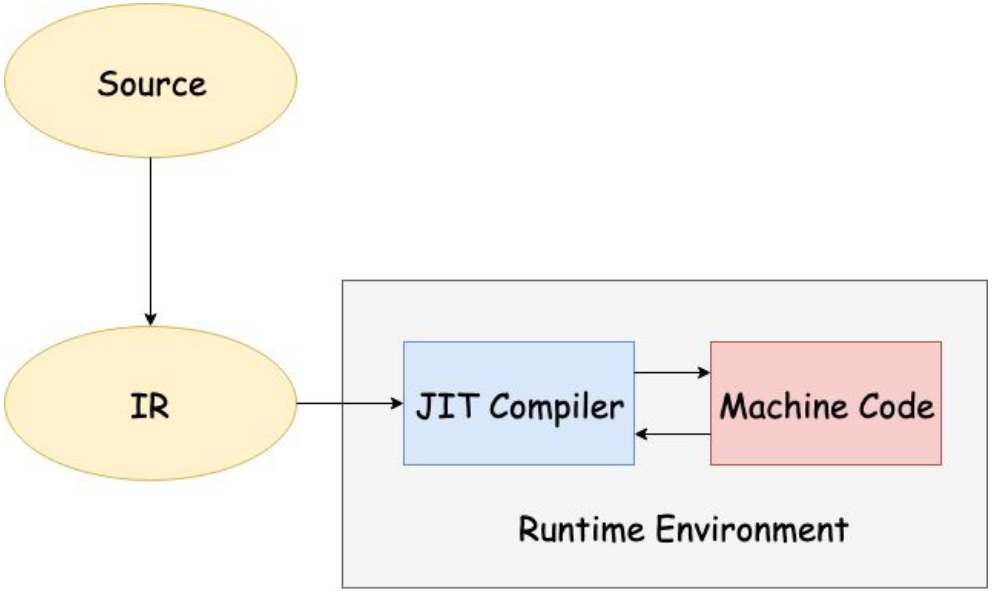
Native languages

JIT 简介



VM-based languages - Interpreter

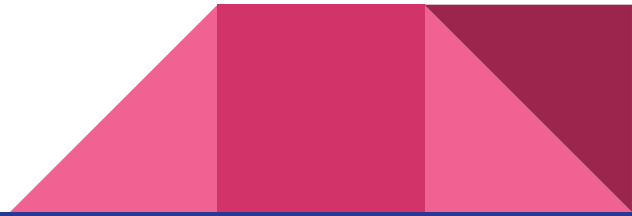
JIT 简介



VM-based languages - JIT

代码生成基础

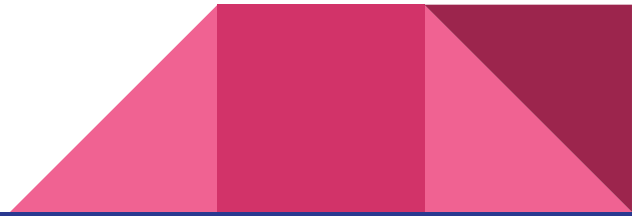
- JIT 的核心功能：将 IR（中间表示）转换为目标架构的机器码
- 虚拟指令集 -> 目标架构指令集



代码生成基础

x86-64

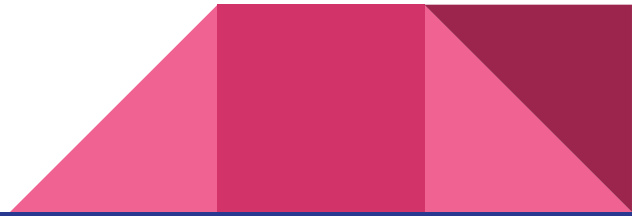
- 64 位
- 16 个通用寄存器 + 16 个 XMM (SSE) 寄存器
- System V ABI
- 向下增长的机器栈
- 分页虚拟内存



代码生成基础

WebAssembly

- 32 位
- 栈机
- 栈深度静态决定
- 本地变量、全局变量
- 线性内存

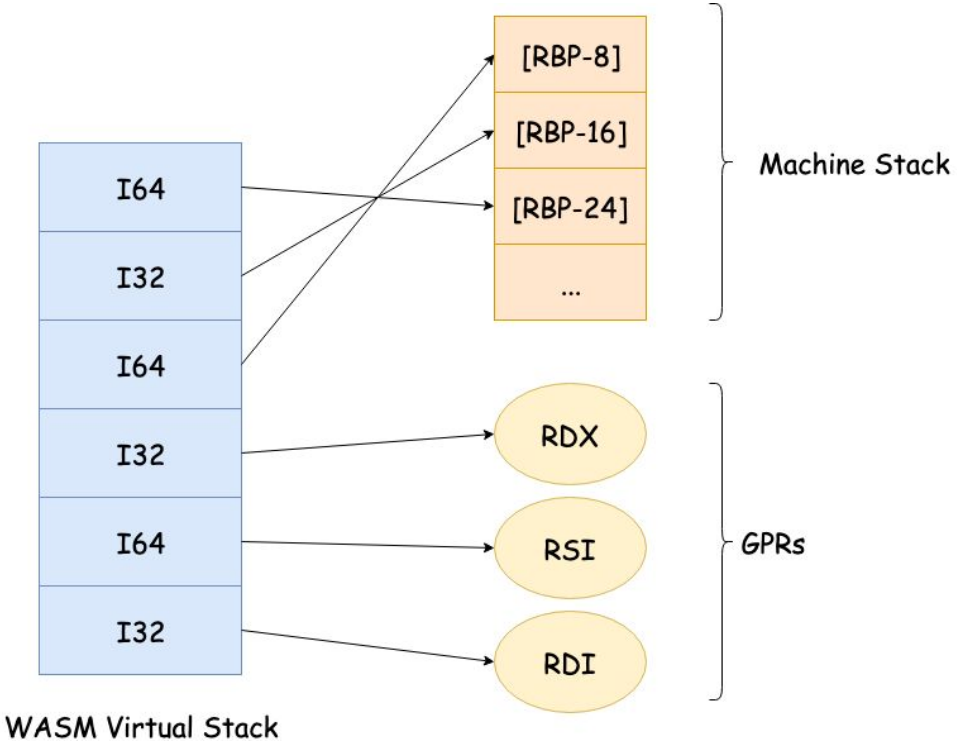


代码生成基础

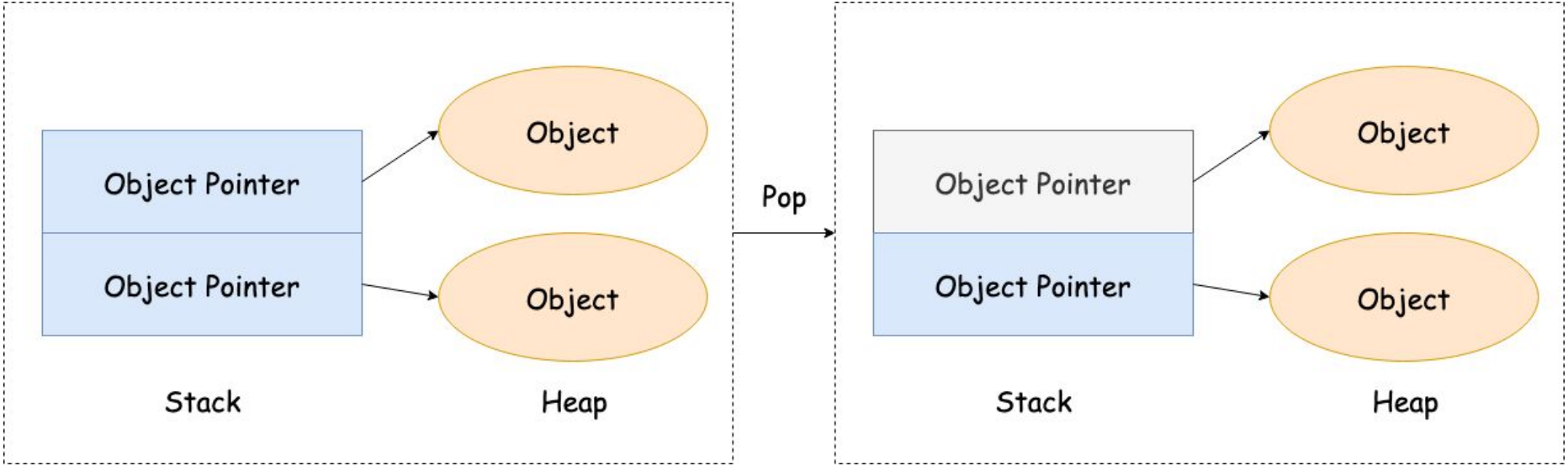
- 怎样将 WebAssembly 虚拟指令集的语义映射到 x86-64 ？

https://github.com/wasmerio/wasmer/blob/master/lib/singlepass-backend

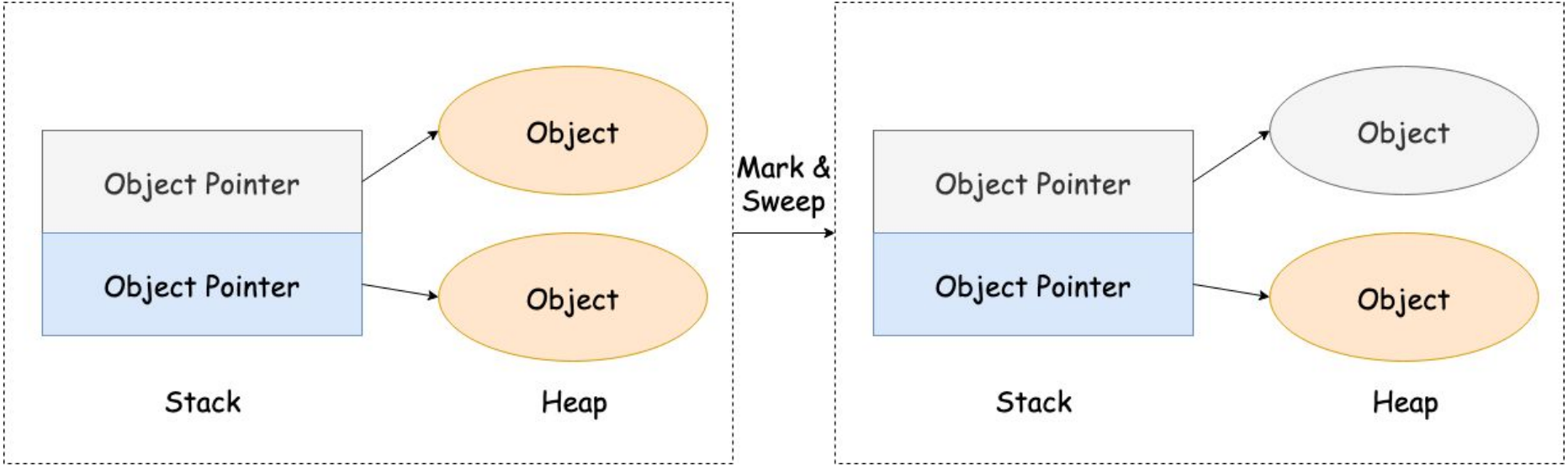
Stackmap：状态映射



垃圾收集(GC)

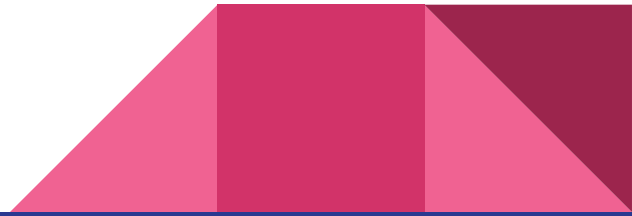

Object Pointer
Object Pointer
Object
Object
Stack
Heap
Pop
Object Pointer
Object Pointer
Object
Object
Stack
Heap

垃圾收集(GC)

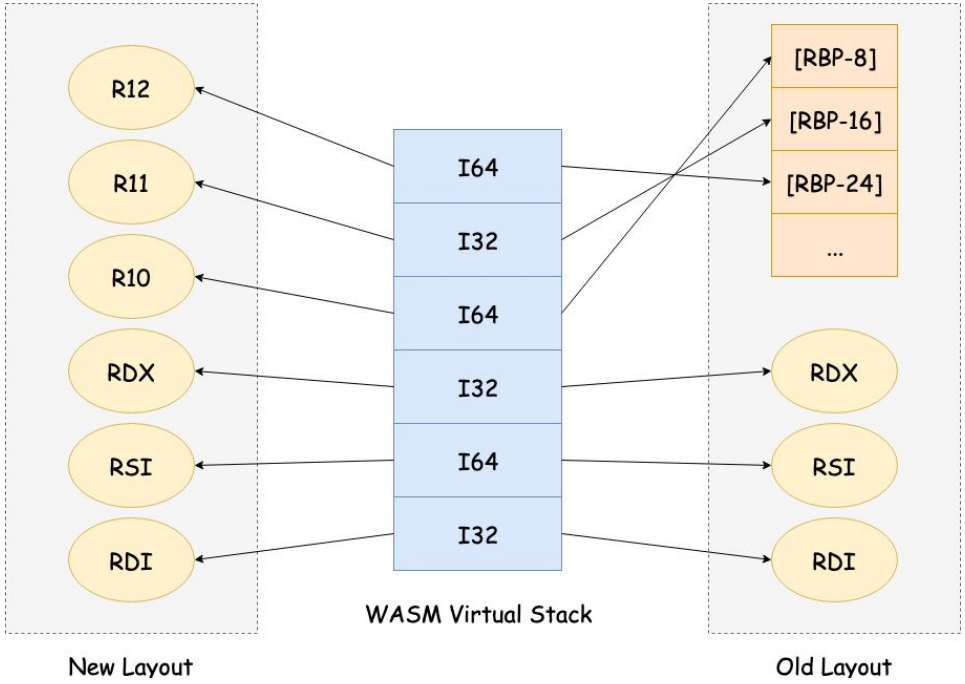

Object Pointer
Object Pointer
Object
Object
Stack
Heap
Mark &
Sweep
Object Pointer
Object Pointer
Object
Object
Stack
Heap

栈上替换(OSR)

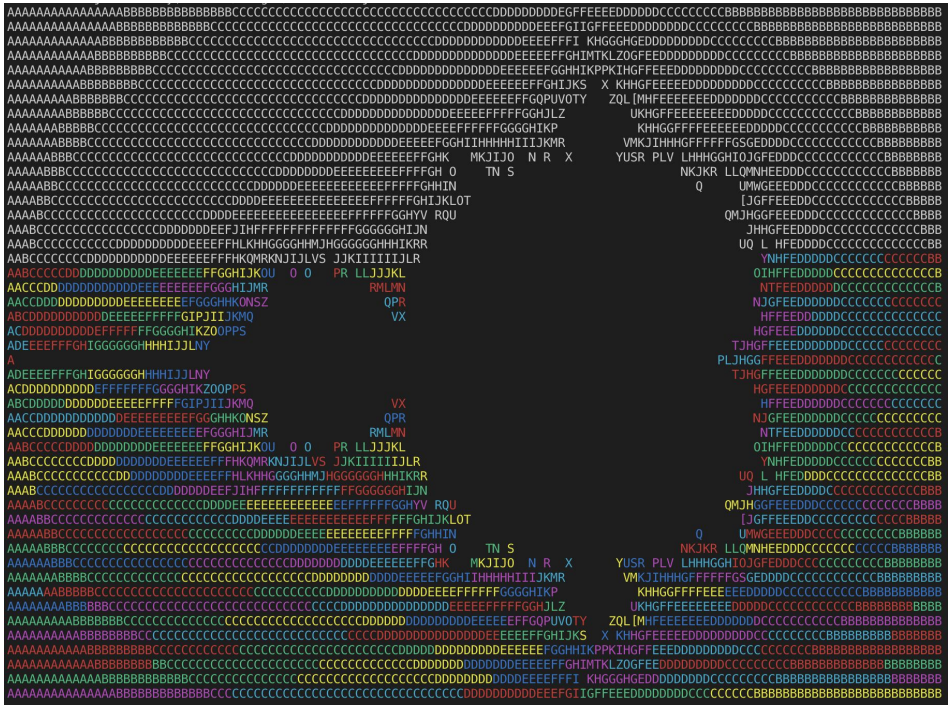
- JIT 可以支持不同的优化等级
- 优化等级越高，需要的编译时间越长
- 栈上替换:用低优化等级的代码快速启动程序。当满足条件时，开始更高优化等级的编译。编译完成后，暂停原代码的执行，构造新的寄存器和栈状态，切换到新代码。



栈上替换(OSR)

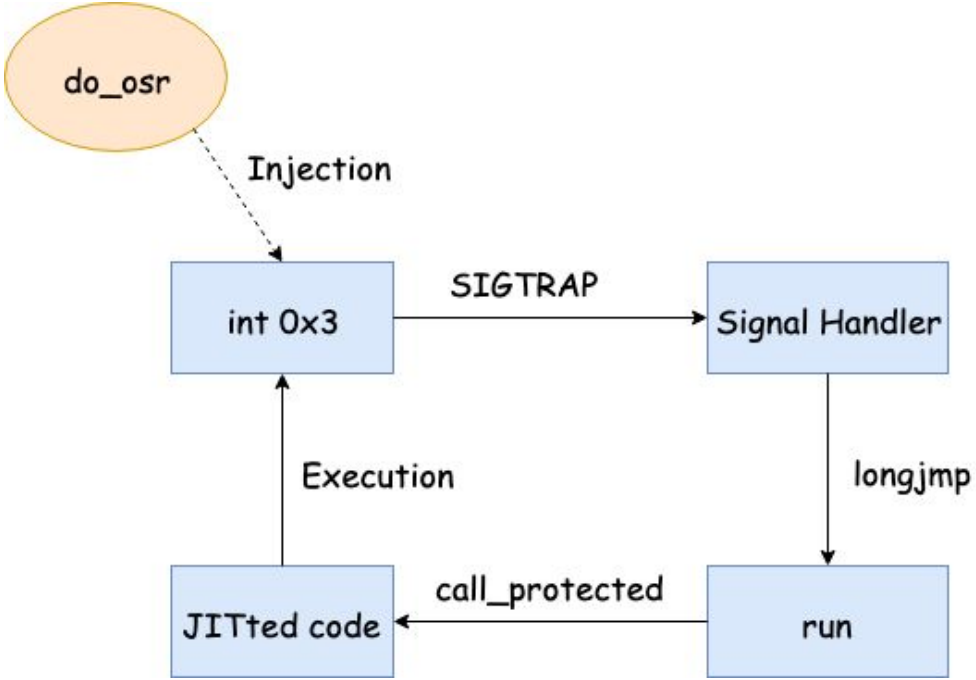

R12
R11
R10
RDX
RSI
RDI
I64
I32
I64
I32
I64
I32
[RBP-8]
[RBP-16]
[RBP-24]
...
RDX
RSI
RDI
WASM Virtual Stack
New Layout
Old Layout

栈上替换(OSR)



Example：bfjit-osr

栈上替换(OSR)

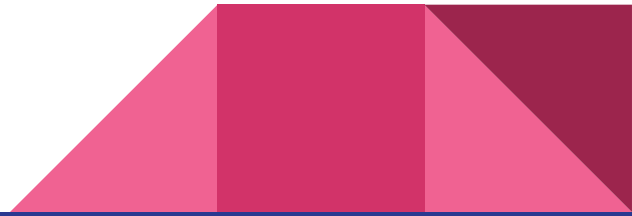

do_osr
Injection
int 0x3
SIGTRAP
Signal Handler
Execution
longjmp
JITted code
call_protected
run

栈上替换(OSR)

```
80 3f 00                cmpb    $0x0, (%rdi)

0f 85 1f ff ff ff       jne     0x1001d486e [backedge]
```

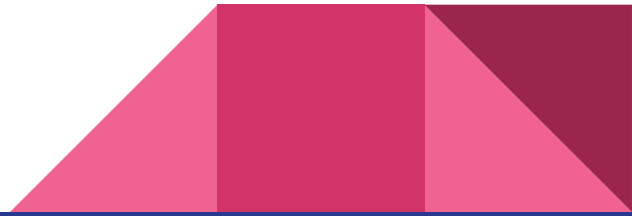
栈上替换(OSR)

```
cc 3f 00                int 0x3 [+invalid]

0f 85 1f ff ff ff       jne     0x1001d486e [backedge]
```



运行时特化

- **类型**特化
- **值**特化



运行时特化

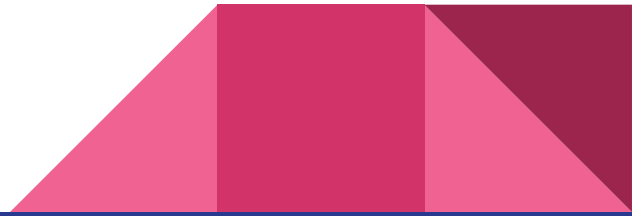
```
function add(a, b) {
    return a + b;
}
add(1, 2);
```

运行时特化

```
/* GUARD: type(a) == i32, type(b) == i32 */

function add(a: i32, b: i32) -> i32 {

     return a + b;

}
```

运行时特化

```
function foo(a) {
      if(satisfies_condition_1(a)) do_1();
      else if(satisfies_condition_2(a)) do_2();
      …
      else if(satisfies_condition_42(a)) do_42();
      else do_0();
}
foo(30); // in case satisfies_condition_30(a) == true
```



运行时特化

```
/* GUARD: satisfies_condition_30(a) */

function foo(a) {

      do_30();

}
```

Tracing / Meta-tracing

User program	Trace when x is set to 6	Optimised trace
`f x < 0:`	`guard_type(x, int)`	`guard_type(x, int)`
`  x = x + 1`	`guard_not_less_than(x, 0)`	`guard_not_less_than(x, 0)`
`else:`	`guard_type(x, int)`	`x = int_add(x, 5)`
`  x = x + 2`	`x = int_add(x, 2)`	
`x = x + 3`	`guard_type(x, int)`	
	`x = int_add(x, 3)`	

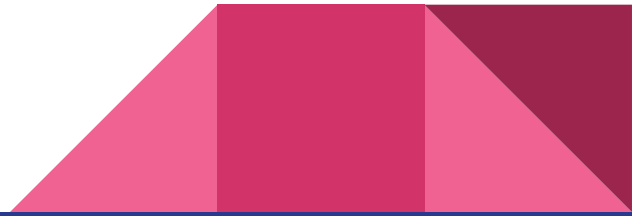
Tracing / Meta-tracing

User program	Trace when x is set to 6	Optimised trace
`f x < 0:`	`guard_type(x, int)`	`guard_type(x, int)`
`  x = x + 1`	`guard_not_less_than(x, 0)`	`guard_not_less_than(x, 0)`
`else:`	`guard_type(x, int)`	`x = int_add(x, 5)`
`  x = x + 2`	`x = int_add(x, 2)`	
`x = x + 3`	`guard_type(x, int)`	
	`x = int_add(x, 3)`	

Meta-tracing: 被 Trace 的代码本身是目标语言的解释器

Partial Evaluation

- 当只有 `Bytecode` 已知时，对 `Interpret(Bytecode, Input)` 部分求值
- `Functional -> SSA`



Thank you!