闭包

我们要做什么

直到 KNF 中间代码为止，我们都可以声明嵌套函数，嵌套的函数能捕获外面的变量：

```
let rec adder(a: Int) -> (Int) -> Int = {
  let rec add(b: Int) -> Int = {
    a + b   // 我能访问外层函数定义的变量 a
  } in
  add
} in

let add1 = adder(1) // 这个时候外层函数的环境已经不存在了
add1(2)             // 但是我们依然能算出来结果是 3
```

但是大多数低级语言（如汇编）是不支持嵌套函数的。

我们需要让函数脱离了原先的环境，依然能够访问之前环境中定义的变量。

所以……

我们需要让函数脱离了原先的环境，依然能够访问之前环境中定义的变量。

那直接把环境一起存起来不就好了！

> **闭包**（Closure），又称词法闭包（Lexical Closure）或函数闭包（function closures），是在支持头等函数的编程语言中实现词法绑定的一种技术。闭包在实现上是一个结构体，它存储了一个函数（通常是其入口地址）和一个关联的环境（相当于一个符号查找表）。

Flat Closure vs. Link Closure

Link Closure 将每一层函数调用的局部变量都存起来，同时存储指向上一层的指针。符合解释器的结构特点（如 Lua 解释器在用），但是性能差。

```
struct StackFrame { parent: StackFrame?, locals: ... }
struct Closure { ptr: fn(), env: StackFrame }
```

Flat Closure 将所有被捕获的变量都存到自己的结构体中。编译型语言一般用它。

```
struct Closure { ptr: fn(), captures: ... }
```

所有代码都是伪代码，下同

构建闭包

我们将以以下代码为例

```
let rec adder(a: Int) -> (Int) -> Int = {
  let rec add(b: Int) -> Int = {
    a + b
  } in
  add
} in

let add1 = adder(1)
add1(2)
```

构建闭包

我们可以找到一个函数用到（“捕获”）了哪些变量，a.k.a. 函数定义的自由变量（free variable）是哪些。

```
fn fv(e: Knf) -> Set {
  match e {
    Add(a, b) => Set::of([a, b])      // 简单表达式用到的变量就是它们的参数
    Sub(a, b) => Set::of([a, b])

    // Let 定义了这个变量，所以它不是自由变量
    Let(x, t, e, cont) => union(fv(e), fv(cont).remove(x))
    // LetRec 定义了函数名和函数的参数，所以它们也都不是自由变量
    LetRec(f, args, cont) => union(fv(f).remove(args), fv(cont)).remove(f)
  }
}
```

比如对于上文的 `adder`，`fv(add) == {a}`，代表它捕获了 `a` 变量。

构建闭包

如果能把闭包捕获的变量存起来，调用的时候就依然能获取到他们的值。

```
struct SimpleAddClosure {
  captured_a: Int              // a 是被捕获的变量
}

fn make_add_closure(a: Int) -> SimpleAddClosure* {
  allocate(SimpleAddClosure::{
    captured_a: a,             // 在构建闭包的时候捕获了 a
  })
}

fn real_add(cls: AddClosure*, b: Int) -> Int {
  let a = cls.captured_a       // 在被调用时加载被捕获的 a
  a + b
}
```

调用闭包

在调用闭包的时候，把捕获的变量传进去就好了。

```
struct SimpleAddClosure {
  captured_a: Int              // a 是被捕获的变量
}

fn adder(a: Int) -> SimpleAddClosure* {
  make_add_closure(a)          // 捕获变量
}

fn main {
  let add1 = adder(1)          // 拿到闭包
  real_add(add1, 2)            // 调用闭包
}
```

但是老师，我想要函数指针

用到闭包的时候，你有可能不只需要函数捕获的环境，还有函数实现本身……

```
// 我是一个高阶函数，我需要函数作为参数
fn map(f: (Int) -> Int, arr: Array[Int]) -> Array[Int] {...}

let add1 = adder(1)
let mul255 = multiplier(255)
let arr = map(add1, ...)    // 把不同的闭包传进去
let arr2 = map(mul255, arr)
```

一般来说你会想传一个函数指针进去，但是现在你还要同时传函数捕获的环境……

加一个函数指针

我们可以规定，闭包的第一个元素一定是指向具体实现的函数指针……

```
struct Closure {
  func: fn(Int) -> Int
}

struct AdderClosure extends Closure {
  // func: fn(Int) -> Int 是第一个字段
  captured_a: Int
}

fn make_add_closure(a: Int) -> Closure* {
  allocate(AdderClosure::{
    func: real_add,
    captured_a: a,
  }) as Closure*
}
```

加一个函数指针

……这样在调用的时候，就可以选择调用传进来的具体实现了。

```
fn real_add(cls: Closure*, b: Int) -> Int {
  let cls = cls as AdderClosure* // 能传到这里的一定是我要的类型
  let a = cls->captured_a        // 这样就能拿到捕获的变量
  a + b                          // 普通地算出结果
}

fn call_closure(cls: Closure*, b: Int) -> Int {
  let func_ptr = cls->func       // load cls[0]
  (func_ptr)(cls, b)             // 调用闭包里存储的函数指针
}
```

总的来说

- 当遇到函数作为变量/参数/返回值使用时，把它转成闭包。
- 闭包的第一个字段是函数指针，其余的是它捕获的变量（记得做好布局）。
- 当调用闭包时，先从第一个字段取出指针并调用。
- 在闭包的函数实现内用到捕获的变量时，从闭包的其余字段中读。

其实……

- 很多编程语言不区分普通函数和闭包，所以所有函数都会被编译成闭包。
- 需要反过来区分哪些函数不是闭包，并转化成普通调用
    - 没有捕获变量，没有用作函数指针。

KNF 转换成 Closure IR

在写成 IR 的时候可以简单一点：暂时可以不用管具体的调用方式。

Closure IR 除了闭包以外和 KNF 基本一致。

```
fn knf_to_closure(env, e: Knf) -> ClosureIR {
  match e {
    LetRec(f, ty, body, cont) => {
      let body = knf_to_closure(env, body)  // 递归转换函数体
      let free_vars = fv(f).to_array()      // 记下自由变量
      let def = Func::{ name: f, ty, formal_fv: free_vars, body }
      env.add_function(def)                 // 把函数定义加到全局里面去
      MakeClosure({ name: f, actual_fv: free_vars },   // 赋值自由变量
          knf_to_closure(cont))
    }
    // ...
  }
}
```

Bonus：一些优化

- 如果闭包的所有调用点都不涉及动态调用，那么可以去掉结构体中的函数指针。（就像我们一开始做的那样）
- 如果闭包不会逃出定义它的环境以外，那么它捕获的变量就可以作为参数传入，而无需分配内存。

```
fn foo() {
  let i = 10
  fn bar() -> Int {
    i
  }
  bar()       // 这里还能访问到 i，所以可以不捕获它
}
```

静态调用的例子

```
fn parent() -> (Int) -> Int {
  let j = 2
  fn closure(i: Int) -> Int{
    i + j
  }

  closure(1);   // 静态调用
  closure
}

parent()(1);    // 动态调用
```

静态调用的例子

```
fn closure_static(i: Int, j: Int) -> Int{
  i + j
}

fn closure_dynamic(cls: Closure*, i: Int) -> Int {
  closure_static(i, cls->j)
}

fn parent() -> (Int) -> Int {
  let j = 2

  closure_static(1, j);   // 静态调用
  let cls = make_closure(closure_dynamic, j)
  cls
}

let cls = parent();
(cls.func)(1);            // 动态调用
```