Functional languages Part I – functions

Michel Schinz (parts based on slides by Xavier Leroy)
Advanced Compiler Construction / 2006-04-28

Higher-order functions

Higher-order function

A **higher-order function** (**HOF**) is a function which either:

- takes another function as argument, or
- returns a function.

Many languages offer higher-order functions, but not all provide the same power...

HOFs in C

In C, it is possible to pass a function as an argument, and to return a function as a result.

However, C functions cannot be nested: they must all appear at the top level. This severely restricts their usefulness, but greatly simplifies their implementation – they can be represented as simple code pointers.

HOFs in functional languages

In functional languages – Scala, Scheme, OCaml, etc. – functions can be nested, and they can survive the scope which defined them.

This is very powerful as it permits the definition of functions which return “new” functions – *e.g.* function composition.

However, as we will see, it also complicates the representation of functions, as simple code pointers are no longer sufficient.

HOF example

To illustrate the issues related to the representation of functions in a functional language, we will use the following example:

```
(define make-adder
  (lambda (x)
    (lambda (y) (+ x y))))

(define increment (make-adder 1))
(increment 41) ⇒ 42

(define decrement (make-adder -1))
(decrement 42) ⇒ 41
```

Representing adder functions

To represent the functions returned by calls to `make-adder`, we basically have two choices:

- keep the code pointer representation for functions – but that implies runtime code generation!
- find another representation for functions, which does not depend on runtime code generation.

Closures

Closures

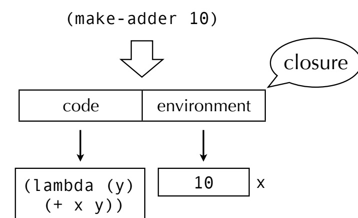
To adequately represent the function returned by `make-adder`, its code pointer must be augmented with the value of `x`.

Such a combination of a code pointer and an **environment** giving the values of the **free variables** – here `x` – is called a **closure**.

The name refers to the fact that the pair (code pointer, environment) is self-contained.

Closures

`(make-adder 10)`



The code of the closure must be evaluated in its environment, so that `x` is "known".

Closure introduction

Using closures instead of function pointers to represent functions changes the way they are manipulated at run time:

- function abstraction builds and returns a closure instead of a simple code pointer,
- function application passes the environment as an additional argument when calling the code pointer.

Closure representation

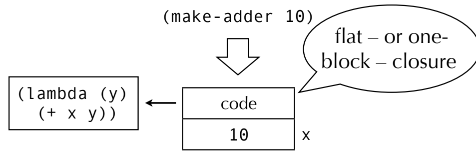
During function application, nothing is known about the closure being called – it can be any closure in the program.

The code pointer must therefore be at a known and constant location so that it can be extracted.

The *contents* of the environment, however, is not used during application itself: it will only be accessed by the function body. This provides some liberty to represent it.

Flat representation

In **flat** closures, the environment is “inlined” in the closure itself, instead of being referred from it. The closure plays the role of the environment.



Recursive closures

Recursive functions need access to their own closure. For example:

```
(define f
  (lambda (l) ... (map f l) ...))
```

How is this implemented?

Recursive closures

Recursive closures can be implemented in several ways:

- the closure – here `f` – can be treated as a free variable, and put in its own environment – leading to a cyclic closure,
- the closure can be rebuilt from scratch,
- with flat closures, the environment *is* the closure, and can be reused directly.

Mutually-recursive closures

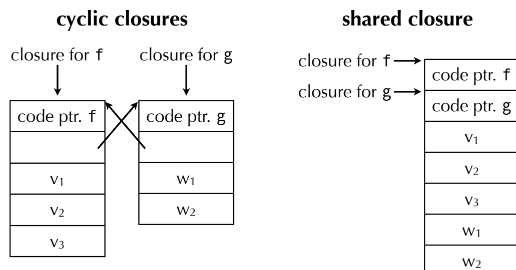
Mutually-recursive functions all need access to the closures of all the functions in the definition.

For example:

```
(letrec ((f (lambda (l) …(compose f g)…))
         (g (lambda (l) …(compose g f)…)))
  …)
```

Solutions: either use cyclic closures, or a single shared one with interior pointers.

Mutually-recursive closures



Translating closures

Closure conversion

In a compiler, closures can be implemented by a simplification phase, called **closure conversion**.

Closure conversion transforms a program in which functions can be nested and have free variables into an equivalent one containing only top-level – and hence closed – functions.

The output of closure conversion is therefore a program in which functions can be represented as code pointers!

The two aspects of closure conversion

Closure conversion can be split in two phases:

- the closing of functions, through the introduction of environments,
- the hoisting of nested, closed functions to the top level.

We will examine them later, but we first need to define the concept of free variable.

Free variables

The **free variables** of a function are the variables which are used but not defined in that function – *i.e.* they are defined in some enclosing scope.

Notice that this concept is relative: in a correct program, all variables are defined somewhere, so they are never free in an absolute sense, but only with respect to some context.

Global variables are never considered to be free, since they are available everywhere.

Free variables example

Our adder example contains two functions, corresponding to the two occurrences of the `lambda` keyword:

```
(define make-adder
  (lambda (x)
    (lambda (y) (+ x y))))
```

The outer one does not have any free variable – it is a **closed function**, like all top-level functions – while the inner one has a single free variable: `x`.

Closing functions

Functions are closed by adding a parameter representing the environment, and using it in the function’s body to access free variables.

Function abstraction and application must of course be adapted to create and pass that environment: abstraction must create and initialise the closure and its environment, while application must extract the environment and pass it as an additional parameter.

Closing example

```
(define make-adder
  (lambda (x)
    (lambda (y) (+ x y))))
```

⇩

```
(define make-adder
  (let ((closure0 ($alloc 1)))
    ($set closure0 0
          (lambda (env0 x)
            (let ((closure1 ($alloc 2)))
              ($set closure1 0
                    (lambda (env1 y) (+ ($get env1 1) y)))
              ($set closure1 1 x)
              closure1)))
    closure0))
```

closure for `make-adder`

closure for anonymous adder

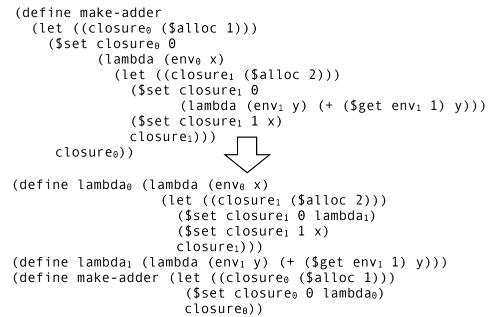
Hoisting functions

Once they are closed, nested anonymous functions can be easily be hoisted to the top level and given an arbitrary name.

The original occurrence of the nested function is simply replaced by that name.

After hoisting, all functions appearing in the program are at the top-level, and are of course closed. Therefore, they can be represented by simple code pointers, as in C.

Hoisting example



```
(define make-adder
  (let ((closure0 ($alloc 1)))
    ($set closure0 0
          (lambda (env0 x)
            (let ((closure1 ($alloc 2)))
              ($set closure1 0
                     (lambda (env1 y) (+ ($get env1 1) y)))
              ($set closure1 1 x)
              closure1)))
     closure0))

(define lambda0 (lambda (env0 x)
                  (let ((closure1 ($alloc 2)))
                    ($set closure1 0 lambda1)
                    ($set closure1 1 x)
                    closure1)))
(define lambda1 (lambda (env1 y) (+ ($get env1 1) y)))
(define make-adder (let ((closure0 ($alloc 1)))
                     ($set closure0 0 lambda0)
                     closure0))
```


Closure conversion for minischeme

Closure conversion for minischeme

As we have seen, closure conversion can be performed by first closing functions, and then hoisting nested functions to the top level.

We will look in detail at the closing part for minischeme, which we will specify as a function `C` mapping potentially-open terms to closed ones.

For that, we first need to define a function `F` mapping a term to the set of its free variables.

Free variables for minischeme

F[(`define` name value)] = ∅

F[(`lambda` (v_1 ...) $body_1$ ...)] =
 (F[$body_1$] ∪ F[$body_2$] ∪ ...) \ { v_1, ... }

F[(`let` ((v_1 e_1) ...) $body_1$ ...)] =
 (F[e_1] ∪ ... ∪ F[$body_1$] ∪ ...) \ { v_1, ... }

F[(`if` e_1 e_2 e_3)] = F[e_1] ∪ F[e_2] ∪ F[e_3]

F[(e_1 e_2 ...)] = F[e_1] ∪ F[e_2] ∪ ...

F[v] = { v } if v is local, and ∅ if v is global.

Closing minischeme functions (1)

Closing minischeme constructs which do not deal with functions or variables is trivial:

C[(`define` name value)] =
 (`define` name C[value])

C[(`let` ((v_1 e_1) ...) $body_1$ ...)] =
 (`let` ((v_1 C[e_1]) ...) C[$body_1$] ...)

C[(`if` e_1 e_2 e_3)] = (`if` C[e_1] C[e_2] C[e_3])

Closing minischeme functions (2)

Abstraction and application are more interesting:

```
C[(lambda (v1 ...) body1 ...)] =
  (let ((closure ($alloc |F[body1 ...]| + 1)))
    ($set closure 0
          (lambda (env v1 ...) C[body1] ...))
    ($set closure 1 F[body1 ...]@1)
    ...
    closure)

C[(e1 e2 ...)] =
  (let ((closure C[e1]))
    (($get closure 0) closure C[e2] ...))
```


Closing minischeme functions (3)

Finally, the translation of variables must distinguish three cases:

C[v] = v
if v is not a free variable,

C[v] = `($get env i)`
if v is a free variable stored at index i in the environment,

C[v] = `closure`
if v refers to the closure being defined.

Hoisting for minischeme

Hoisting consists in lifting closed nested functions to the top-level, naming them in the process.

Like closing, hoisting can be specified as a function, say H, mapping a term which potentially contains nested functions, to a new version of that term without nested functions – plus a list of additional definitions.

The definition of that function is left as an exercise.

Closures and objects

Closures and objects

There is a strong similarity between closures and objects: closures can be seen as objects with a single method – containing the code of the closure – and a set of fields – the environment.

In Java, the ability to define nested classes can be used to simulate closures, but the syntax is too heavyweight to be used often.

In Scala, a special syntax exists for anonymous functions, which are translated to nested classes.

Closures in Scala (1)

To see how closures are handled in Scala, we will look at how the compiler translates the Scala equivalent of the `make-adder` function:

```
def makeAdder(x: Int): Int=>Int =
  { y: Int => x+y }
val increment = makeAdder(1)
increment(41)
```

Closures in Scala (2)

In a first phase, the anonymous function is turned into an anonymous class of type `Function1` – the type of functions with one argument. This class is equipped with a single `apply` method.

```
def makeAdder(x: Int): Function1[Int,Int]=
  new Function1[Int,Int] {
    def apply(y: Int): Int = x+y
  }
val increment = makeAdder(1)
increment.apply(41)
```


Closures in Scala (3)

In a second phase, the anonymous class is named.

```
def makeAdder(x: Int):Function1[Int,Int]={
  class Anon extends Object
             with Function1[Int,Int] {
    def apply(y: Int): Int = x+y
  }
  new Anon
}
val increment = makeAdder(1)
increment.apply(41)
```


Closures in Scala (4)

In a third phase, the `Anon` class is closed and hoisted to the top level.

```
class Anon(x:Int) extends Object
                   with Function1[Int,Int{
  def apply(y: Int): Int = x+y
}
def makeAdder(x: Int):Function1[Int,Int]={
  new Anon(x)
}
val increment = makeAdder(1)
increment.apply(41)
```


Closures in Scala (5)

Finally, the constructor of `Anon` is made explicit.

```
class Anon extends Object
           with Function1[Int,Int] {
  private var x: Int = _;
  def this(x0: Int) { this.x = x0 }
  def apply(y: Int): Int = x+y
}
def makeAdder(x: Int):Function1[Int,Int]={
  new Anon(x)
}
val increment = makeAdder(1)
increment.apply(41)
```


Summary

In C, all functions have to be at the top level, and can therefore be represented as code pointers.

Functional languages allow functions to be nested and to survive the scope which created them. They have to be represented by a closure, which pairs a code pointer with an environment, giving the values of the code's free variables.

Closures can be implemented by a program transformation called closure conversion.