Closure conversion

Michel Schinz – parts based on slides by X. Leroy
Advanced compiler construction, 2008-04-04

Higher-order functions

Higher-order function

A **higher-order function** (**HOF**) is a function that 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 that defined them.

This is very powerful as it permits the definition of functions that return “new” functions – *e.g.* functional 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 Scheme 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 `make-adder`, we basically have two choices:

1. Keep the code pointer representation for functions. However, that implies run-time code generation, as each function returned by `make-adder` is different!
2. Find another representation for functions, which does not depend on run-time code generation.

Closures

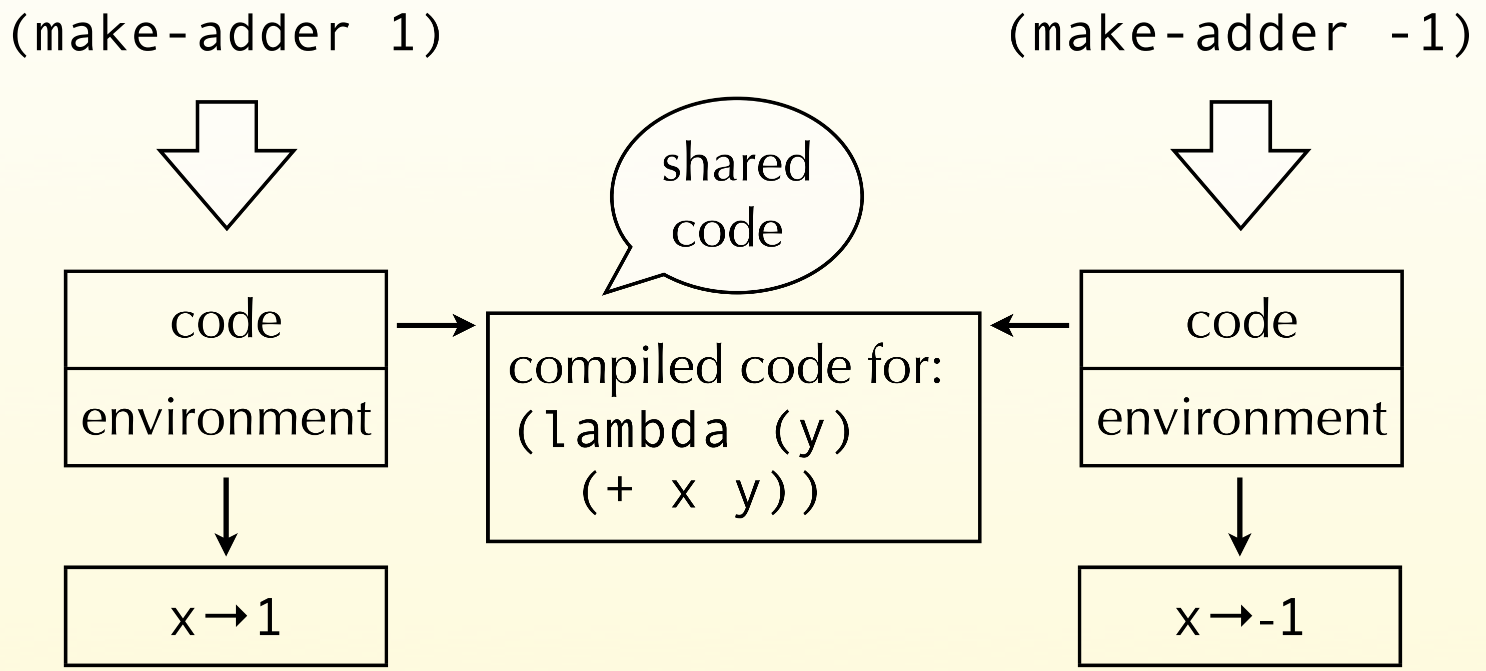
Closures

To adequately represent the functions returned by `make-adder`, their 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 variable(s) – here `x` – is called a **closure**.

The name refers to the fact that the pair composed of the code pointer and the environment is self-contained.

Closure



The code of a closure must be evaluated in its environment, so that `x` is “known”.

Introducing closures

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 extracts the code pointer from the closure, and invokes it with the environment as an additional argument.

Representing closures

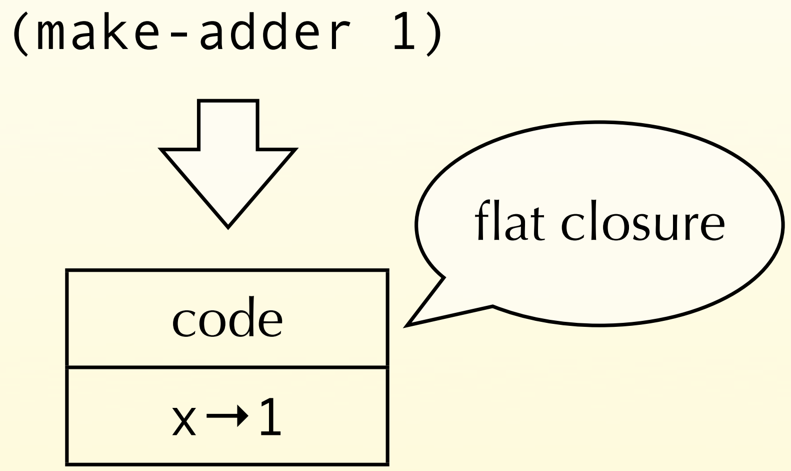
During function application, nothing is known about the closure being called – it can be any closure in the program.

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

The values contained in the environment, however, are not used during application itself: they will only be accessed by the function body. This provides some freedom to place them.

Flat closures

In **flat** (or **one-block**) **closures**, the environment is “inlined” into 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) ...))
```

Several techniques can be used to give a closure access to itself:

1. the closure – here `f` – can be treated as a free variable, and put in its own environment – leading to a cyclic closure,
2. the closure can be rebuilt from scratch,
3. 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, in the following program, `f` needs access to the closure of `g`, and the other way around:

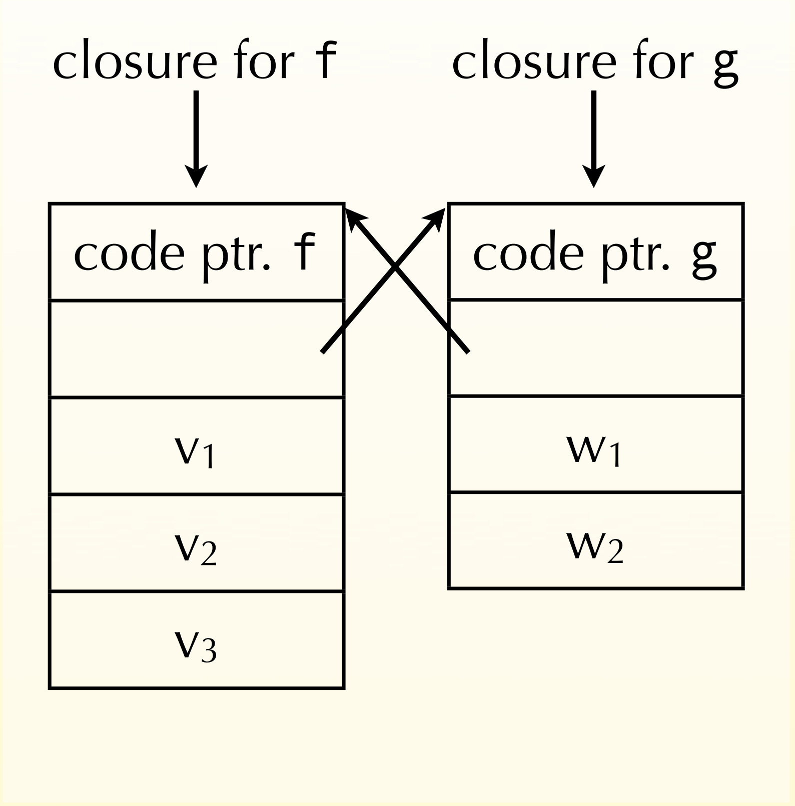
```
(letrec ((f (lambda (l) …(compose f g)…))
         (g (lambda (l) …(compose g f)…)))
  …)
```

Solutions:

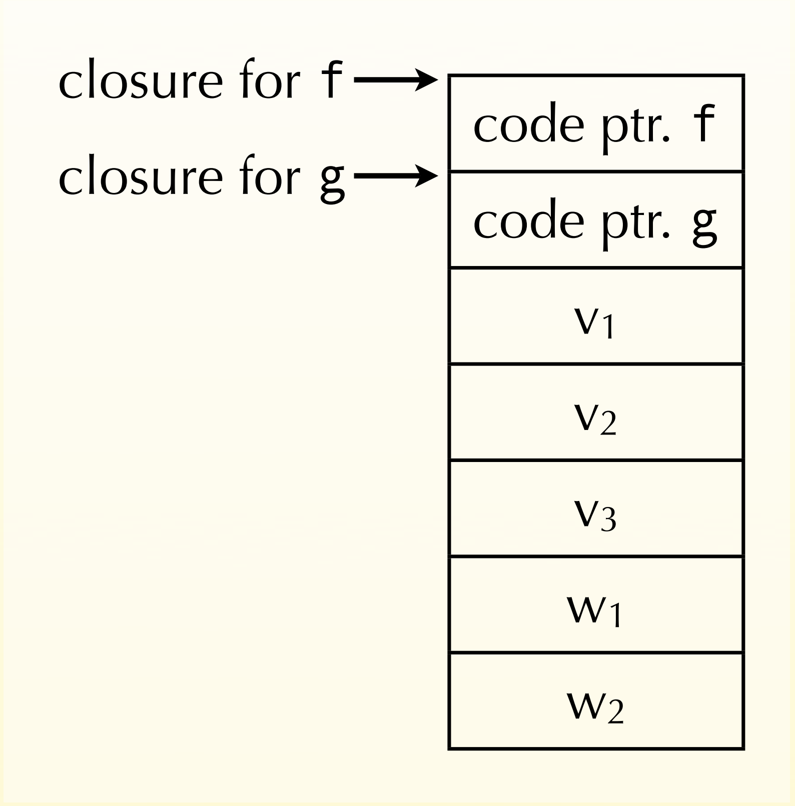
1. use cyclic closures, or
2. share a single closure with interior pointers (note: interior pointers make the job of the GC harder).

Mutually-recursive closures

cyclic closures



shared closure



Compiling 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 have free variables into an equivalent one containing only closed functions.

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

Free variables

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

Global variables are never considered 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. The inner one has a single free variable: `x`.

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. The inner one has a single free variable: `x`.

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. 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 accordingly:

- abstraction must create and initialise the closure and its environment,
- 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
  (vector (lambda (env1 x)
            (vector (lambda (env2 y)
                      (+ (vector-ref env2 1) y))
                    x))))
```

Closing example

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

⇩

closure for `make-adder`

```
(define make-adder
  (vector (lambda (env1 x)
            (vector (lambda (env2 y)
                      (+ (vector-ref env2 1) y))
                    x))))
```

Closing example

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

```
(define make-adder
  (vector (lambda (env1 x)
            (vector (lambda (env2 y)
                      (+ (vector-ref env2 1) y))
                    x))))
```

closure for make-adder

closure for anonymous adder

Closure conversion for minischeme

Minischeme closure conversion

As we have seen, closure conversion consists in closing functions by passing them an environment containig the values of their free variables.

We will specify the closing of minischeme functions 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.

Note: to simplify presentation, we assume in the following slides that all variables in a program have a unique name.

Minischeme free variables

$F[$`(lambda (`v_1 … `)` $body_1$ … `)`$] =$
$(F[body_1] \cup F[body_2] \cup \ldots) \setminus \{ v_1, \ldots \}$
$F[$`(if` e_1 e_2 e_3`)`$] = F[e_1] \cup F[e_2] \cup F[e_3]$
$F[$`(`e_1 e_2 …`)`$] = F[e_1] \cup F[e_2] \cup \ldots$
$F[v]$ when v is local $= \{ v \}$
$F[v]$ when v is global or a primitive $= \varnothing$

Note: since a `let` form is equivalent to the application of an anonymous function, it is easy to deduce the rule to compute its free variables from the rules above. This is left as an exercise.

Closing minischeme functions

Closing minischeme constructs that 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])

C[x] where x is a number or identifier =
 x

Closing minischeme functions

Abstraction is closed by creating and returning the closure, represented as a vector:

C[(lambda (v_1 …) $body_1$ …)] =
  (vector (lambda ($\underline{\texttt{env}}$ v_1 …) ϵ[C[$body_1$],F,$\underline{\texttt{env}}$] …)
          F_1 F_2 …)

fresh variable

where

- ϵ[*t*,*f*,*e*] transforms *t* by replacing all occurrences of the variables of *f* by accesses to corresponding slots in the environment *e*.
- F = F[(lambda (v_1 …) $body_1$ …)] and F_i is its *i*th component.

Closing minischeme functions

Finally, application extracts the code pointer from the closure, and invokes it with the closure itself as the first argument, followed by the other arguments:

C[(e_1 e_2 …)] when e_1 is not a primitive =
  `(let ((closure` C[e_1]`))`
    `((vector-ref closure 0) closure` C[e_2] …`))`

C[(e_1 e_2 …)] when e_1 is a primitive =
  (e_1 C[e_2] …)

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

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

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 containing the code of the anonymous function.

```
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

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

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

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 that 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, which takes a program where functional values have to be represented as closures and returns an equivalent program where they can be represented as simple code pointers.