

# Closure conversion or: data representation for functions

Advanced Compiler Construction  
Michel Schinz - 2013-03-14

1

# Higher-order functions

2

## 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...

3

## 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.

4

# HOFs in functional languages

In functional languages – Scala, OCaml, Haskell, 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.

5

# HOF example

To illustrate the issues related to the representation of functions in a functional language, we will use the following L<sub>3</sub> example:

```
(def make-adder
  (fun (x)
    (fun (y) (@+ x y))))
(def increment (make-adder 1))
(increment 41) ⇒ 42

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

6

# Representing adders

To represent the functions returned by `make-adder`, there are basically two choices:

1. Use simple code pointers. Unfortunately, this 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.

7

# Closures

8

# Closures

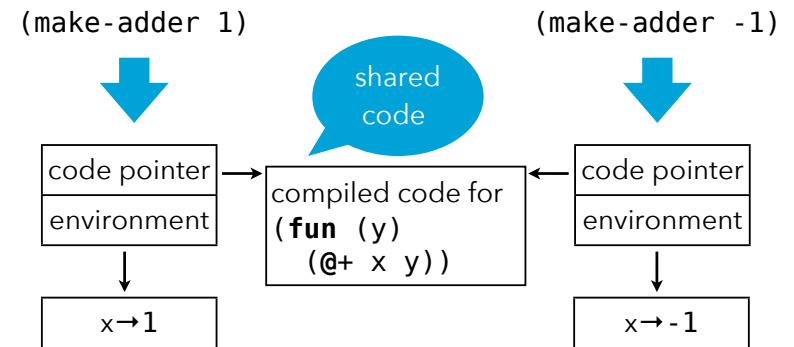
To adequately represent the functions returned by `make-adder`, their code pointer must be augmented with the value of `x`.

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 closed, i.e. self-contained.

9

# Closures



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

10

# Introducing closures

Using closures instead of function pointers changes the way functions 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.

11

# 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.

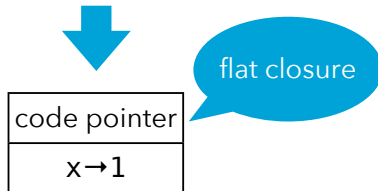
The values contained in the environment, however, are not used during application itself: they will only be accessed by the function body. This provides some freedom to place them.

12

## 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.

(make-adder 1)



13

## Exercise

Given the following  $L_3$  composition function:

```
(def compose  
  (fun (f g)  
    (fun (x) (f (g x))))))
```

draw the flat closure returned by the application

```
(compose succ twice)
```

assuming that **succ** and **twice** are two functions defined in an enclosing scope.

14

## Compiling closures

15

## 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.

16

## Closure conversion

Closure conversion is nothing more than data representation for functions: it encodes the high-level notion of functions of the source language using the low-level concepts of the target language - in this case heap-allocated blocks and code pointers.

17

## 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.

The **make-adder** example contains two functions:

```
(def make-adder
  (fun (x)
    (fun (y) (@+ x y))))
```

The outer one does not have any free variable: it is a closed function. The inner one has a single free variable: **x**.

18

## 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 initialize the closure,
- application must pass the environment as an additional parameter.

19

## Closing example

Assuming the existence of abstract **closure-make** and **closure-get** functions, a closure conversion phase could transform the **make-adder** example as follows:

```
(def make-adder (fun (x)
                  (fun (y) (@+ x y))))
```

```
(make-adder 1)
```



```
(def make-adder
  (closure-make
   (fun (env1 x)
     (closure-make
      (fun (env2 y)
        (@+ (closure-get env2 1) y)
          x))))
   ((closure-get make-adder 0) make-adder 1))
```

20

## Recursive closures

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

```
(letrec ((f (fun (l) ... (map f l) ...)))  
  ...)
```

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

- 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.

21

## 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:

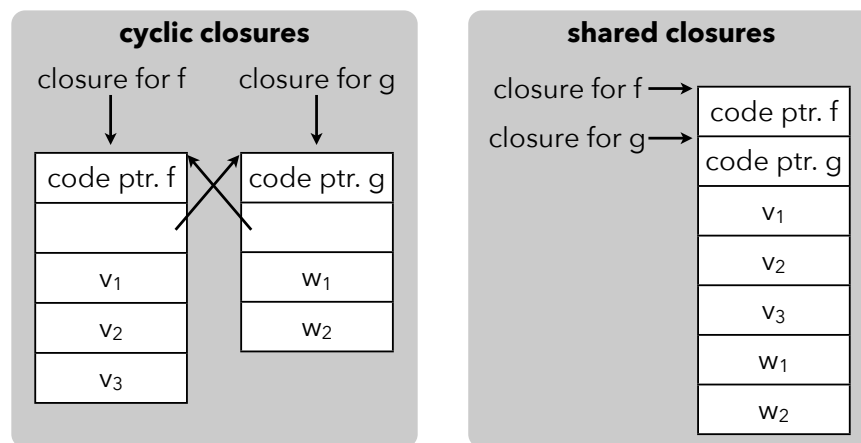
```
(letrec ((f (fun (l) ... (compose f g) ...))  
        (g (fun (l) ... (compose g f) ...)))  
  ...)
```

Solutions:

1. use cyclic closures, or
2. share a single closure with interior pointers - but note that the resulting interior pointers make the job of the garbage collector harder.

22

## Mutually-recursive closures



23

## CPS/L<sub>3</sub> closure conversion

24

## Functions in CPS/L<sub>3</sub>

In the L<sub>3</sub> compiler, we represent L<sub>3</sub> functions using flat closures.

Flat closures are simply blocks tagged with a tag reserved for functions - we choose 202. The first element of the block contains the code pointer while the other elements - if any - contain the environment of the closure.

25

## CPS/L<sub>3</sub> closure conversion

In the L<sub>3</sub> compiler, closure conversion is not a separate phase. Rather, it is the part of the data conversion phase that takes care of representing functions.

Closure conversion is therefore specified exactly like the data representation phase.

26

## CPS/L<sub>3</sub> free variables

The F function computes the free variables of a CPS/L<sub>3</sub> term:

$$F[(\text{let}_l ((n l)) e)] = F[e] \setminus \{n\}$$

$$F[(\text{let}_p ((n (p n_1 \dots))) e)] = \\ (F[e] \setminus \{n\}) \cup \{n_1, \dots\}$$

$$F[(\text{let}_k ((n (\text{cont } (a_1 \dots) b))) e)] = \\ F[e] \cup (F[b] \setminus \{a_1, \dots\})$$

$$F[(\text{let}_f ((f_1 (\text{fun } (k_1 n_{1,1} \dots) e_1)) \dots) e)] = \\ (F[e] \cup (F[e_1] \setminus \{n_{1,1}, \dots\}) \cup \dots) \setminus \{f_1, \dots\}$$

$$F[(\text{app}_k k n_1 \dots)] = \{n_1, \dots\}$$

$$F[(\text{app}_f f k n_1 \dots)] = \{f, n_1, \dots\}$$

$$F[(\text{if } (p n_1 \dots) kt kf)] = \{n_1, \dots\}$$

Note: CPS/L<sub>3</sub> scoping rules ensure that continuation variables are never free, so we ignore them.

27

## Notation

To simplify some of the following slides, we assume that integer literals can be used as arguments of primitives. For example, we write:

```
(letp ((n (block-get b 1))) ...)
```

instead of:

```
(let* ((c1 1)
      (n (block-get b c1)))
  ...)
```

28

# Function definition

closed version of  $f_1$

```

[[letf ((f1 (fun (k1 n1,1 ...) e1)) ...) e)] =
  (letf ((w1 (fun (k1 env1 n1,1 ...)
    (let* ((v1 (block-get env1 1))
      ...)
      [[e1]{f1→env1}{FV1(0)→v1}{...}])
    ...))
    (let* ((f1 (block-alloc-202 |FV1|+1))
      ...
      (t1 (block-set! f1 0 w1))
      (t2 (block-set! f1 1 FV1(0)))
      ...))
      [[e]])
  
```

closure initialization

closure allocation

$FV_i$  = an (arbitrary) ordering of the set  $F[e_i] \setminus \{f_i, n_{i,1}, \dots\}$

# Function application

Function application has to be transformed in order to extract the code pointer from the closure and pass the closure as the first argument after the return continuation:

```

[[appf n nk n1 ...]] =
  (letp ((f (block-get n 0)))
    (appf f nk n n1 ...))
  
```

# Function test

Functions being represented as tagged blocks, checking that an arbitrary object is a function amounts to checking that it is a tagged block and if it is, that its tag is 202.

This can be done directly in  $L_3$ , as a library function:

```

(def function?
  (fun (o)
    (and (@block? o)
      (@= 202 (@block-tag o))))))
  
```

# Exercise

We have seen two techniques to represent the closures of mutually-recursive functions: cyclic closures and shared closures.

Which of these two techniques does our transformation use (explain)?



# Hoisting CPS/L<sub>3</sub> functions

33

## Function hoisting

After closure conversion, all functions in the program are closed. Therefore, it is possible to **hoist** them all to a single outer `letf`.

Once this is done, the program has the following simple form:

```
(letf (all functions of the program)
  main program code)
```

where the main program code does not contain any function definition (`letf` expression).

Hoisting functions to the top level simplifies the shape of the program and can make the job of later phases - e.g. assembly code generation - easier.

34

## CPS/L<sub>3</sub> hoisting (1)

```
[[letl ((n l)) e]] =
  (letf (fs)
    (letl ((n l)) e'))
if [[e] = (letf (fs) e')]
[[letp ((n (p n1 ...))) e]] =
  (letf (fs)
    (letp ((n (p n1 ...))) e'))
if [[e] = (letf (fs) e')]
```

35

## CPS/L<sub>3</sub> hoisting (2)

```
[[letk ((n (cont (n1 ...) b))) e]] =
  (letf (fs1 fs2)
    (letk ((n (cont (n1 ...) b')) e'))
if [[b] = (letf (fs1) b')]
and [[e] = (letf (fs2) e')]
[[letf ((f1 (fun (n1,1 ...) e1)) ...) e]] =
  (letf ((f1 (fun (n1,1 ...) e1) ... fs1 ... fs) e')
if [[ei] = (letf (fsi) ei')]
and [[e] = (letf (fs) e')]
[[e] when e is any other kind of expression =
  (letf () e)
```

36

# Closures and objects

37

## 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.

38

## makeAdder in Scala

To see how closures are handled in Scala, let's look at how the translation of the Scala equivalent of the `make-adder` function:

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

39

## makeAdder translated

(Hoisted) closure class: the code is in the `apply` method, the environment in the object itself: it's a flat closure.

```
class Anon extends Function1[Int,Int] {  
  private val x: Int;  
  def this(x: Int) = { this.x = x }  
  def apply(y: Int): Int = this.x + y  
}
```

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

closure application (the closure is passed implicitly as `this`)

40