Closure conversion

or: data representation for functions

Advanced Compiler Construction

Michel Schinz – 2014-03-13

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, 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.

HOF example

To illustrate the issues related to the representation of functions in a functional language, we will use the following L_3 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
```

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.

Closures

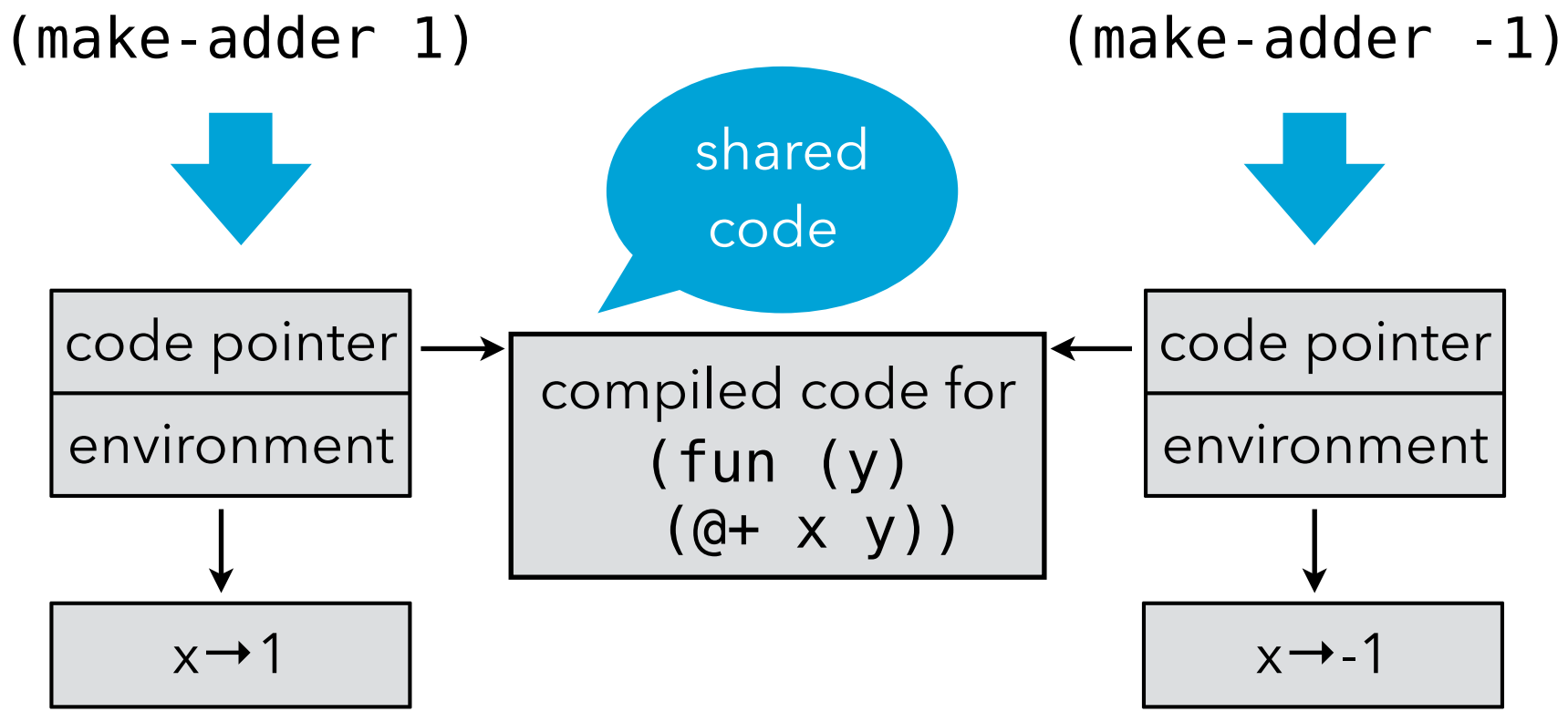
Closures

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 closed, i.e. self-contained.

Closures



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

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.

Representing closures

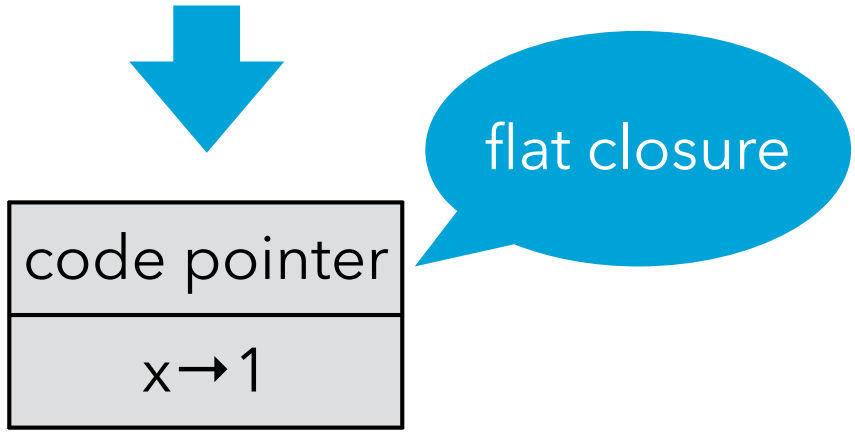
During function application, nothing is known about the closure being called – it can be any closure in the program. The code pointer must therefore be at a known and constant location so that it can be extracted.

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.

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



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.

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.

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.

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.

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.

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)
```

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.

Mutually-recursive closures

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

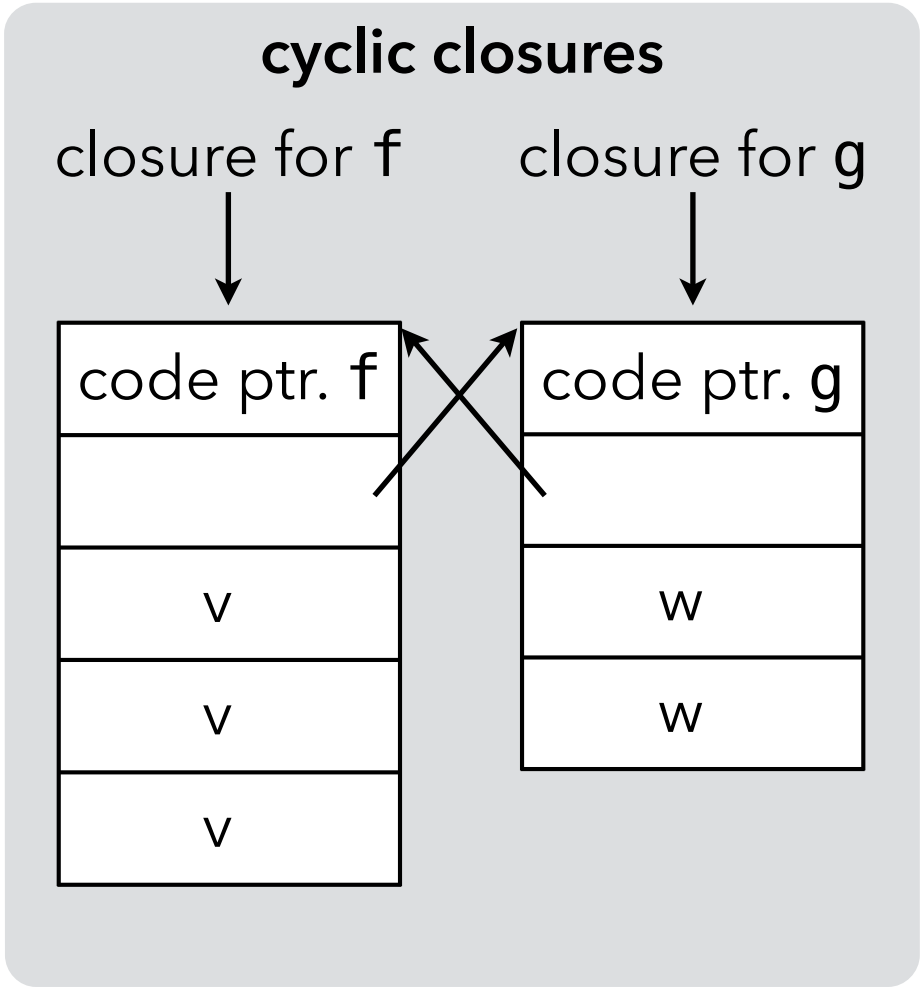
For example, in the following program, `f` needs access to the closure of `g`, and the other way around:

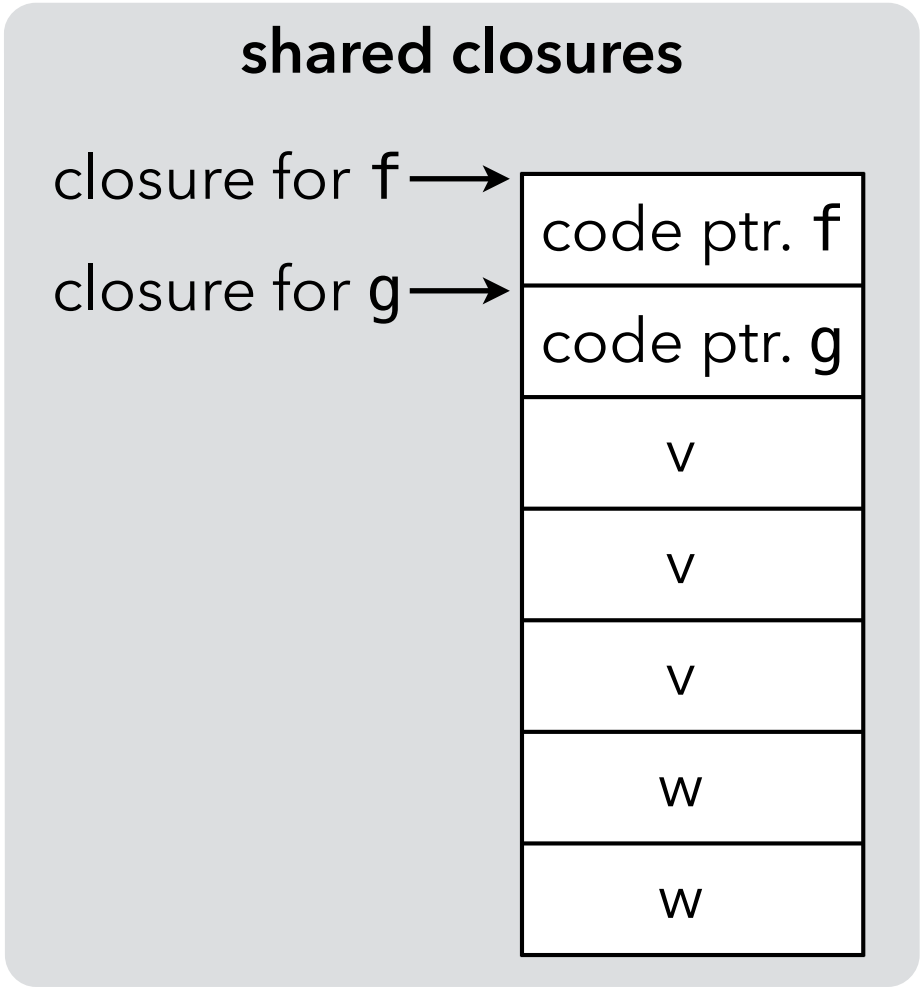
```
(letrec ((f (fun (l) … (compose f g) …))
         (g (fun (l) … (compose g f) …)))
  …)
```

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.

Mutually-recursive closures





CPS/L_3
closure conversion

Functions in CPS/L_3

In the L_3 compiler, we represent L_3 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.

CPS/L_3 closure conversion

In the L_3 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.

CPS/L_3 free variables

The F function computes the free variables of a CPS/L_3 term:

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

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

$F[(\texttt{let}_c\ ((n\ (\texttt{cnt}\ (a_1\ \ldots)\ b)))\ e)] =$
$F[e] \cup (F[b] \setminus \{ a_1, \ldots \})$

$F[(\texttt{let}_f\ ((f_1\ (\texttt{fun}\ (c_1\ n_{1,1}\ \ldots)\ e_1))\ \ldots)\ e)] =$
$(F[e] \cup (F[e_1] \setminus \{ n_{1,1}, \ldots \}) \cup \ldots) \setminus \{ f_1, \ldots \}$

$F[(\texttt{app}_c\ c\ n_1\ \ldots)] = \{ n_1, \ldots \}$

$F[(\texttt{app}_f\ f\ c\ n_1\ \ldots)] = \{ f, n_1, \ldots \}$

$F[(\texttt{if}\ (p\ n_1\ \ldots)\ ct\ cf)] = \{ n_1, \ldots \}$

Note: CPS/L_3 scoping rules ensure that continuation variables are never free, so we ignore them.

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)))
  …)
```

Function definition

closed version of f_1

```
⟦(letf ((f1 (fun (c1 n1,1 ...) e1)) ...) e)⟧ =
  (letf ((w1 (fun (c1 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 $F[e_i] \setminus \{ f_i, n_{i,1}, \ldots \}$

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:

$[\![(\text{app}_f\ n\ n_c\ n_1\ \ldots)]\!] =$

```
(letₚ ((f (block-get n 0)))
  (appf f nc 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)?

Improving CPS/L_3 closure conversion

Translation inefficiencies

The translation just presented is suboptimal in two respects:

1. it always creates closures, even for functions that are never used as values (i.e. only applied),
2. it always performs calls through the closure, thereby making all calls indirect.

These problems could be solved by later optimizations or by a better version of the translation sketched in the next slides.

Translation inefficiencies

The inefficiency of the simple translation can be observed on the `make-adder` example:

```
(def make-adder (fun (x)
                     (fun (y) (@+ x y))))
(make-adder 1)
```

Applied to the CPS version of that program, the simple translation creates a closure for both functions. However, the outer one is closed and never used as a value, therefore no closure needs to be created for it.

Notice that even if the outer function **escaped** (i.e. was used as a value) and needed a closure, the call to it could avoid fetching its code pointer from the closure, as here it is a **known function**.

Improved translation

The simple translation translates a source function into one target function and one closure.

The improved translation splits the target function in two:

1. the **wrapper**, which simply extracts the free variables from the environment and passes them as additional arguments to the worker,
2. the **worker**, which takes the free variables as additional arguments and does the real work.

The wrapper is put in the closure.

The worker is used directly whenever the source function is applied to arguments instead of being used as a value.

Improved abstraction

```
⟦(letf ((f1 (fun (c1 n1,1 ...) e1)) ...) e)⟧ =
  (letf ((w1 (fun (c1 n1,1 ... u1 ...)
               ⟦e1⟧{FV1(0)→u1}{...}))
         (s1 (fun (c1 env1 n1,1 ...)
               (let* ((v1 (block-get env1 1))
                      ...)
                 (appf w1 c1 n1,1 ... v1 ...))))
         ...)
    (let* ((f1 (block-alloc-202 |FV1|+1))
           ...
           (t1 (block-set! f1 0 s1))
           ...)
      ⟦e⟧))
```

worker

wrapper

Improved application

When translating function application, if the function being applied is **known** (i.e. is bound by an enclosing $\texttt{let}_f$), the worker of that function can be used directly, without going through the closure:

$[\![(\texttt{app}_f\ n\ n_c\ n_1\ \ldots)]\!]$ = *if* n *is a known function with worker* n_w

```
(appf  nw nc n n1 … FVn(0) …)
```

$[\![(\texttt{app}_f\ n\ n_c\ n_1\ \ldots)]\!]$ = *otherwise*

```
(letp ((f (block-get n 0)))
  (appf f nc n n1 …))
```

Free variables

The improved translation makes the computation of free variables slightly more difficult.

That's because when a function f calls a known function g, it has to pass it its free variables as additional arguments.

The free variables of g now become free variables of f.

These new free variables must be added to f's arguments, which impacts its callers – which could include g if the two are mutually-recursive. And so on...

Hoisting CPS/L_3 functions

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.

CPS/L_3 hoisting (1)

$[\![(\texttt{let}_{\texttt{l}}\ ((\text{n l}))\ \text{e})]\!] =$

$\quad(\texttt{let}_{\texttt{f}}\ (\text{fs})$

$\quad\quad(\texttt{let}_{\texttt{l}}\ ((\text{n l}))\ \text{e}'))$

$\quad\text{if } [\![\text{e}]\!] = (\texttt{let}_{\texttt{f}}\ (\text{fs})\ \text{e}')$

$[\![(\texttt{let}_{\texttt{p}}\ ((\text{n}\ (\text{p}\ \text{n}_1\ \ldots)))\ \text{e})]\!] =$

$\quad(\texttt{let}_{\texttt{f}}\ (\text{fs})$

$\quad\quad(\texttt{let}_{\texttt{p}}\ ((\text{n}\ (\text{p}\ \text{n}_1\ \ldots)))\ \text{e}'))$

$\quad\text{if } [\![\text{e}]\!] = (\texttt{let}_{\texttt{f}}\ (\text{fs})\ \text{e}')$

CPS/L_3 hoisting (2)

$[\![(\texttt{let}_c\ ((c_1\ (\texttt{cnt}\ (n_{1,1}\ \ldots)\ e_1))\ \ldots)\ e)]\!] =$

$\quad(\texttt{let}_f\ (fs_1\ \ldots\ fs)$

$\quad\quad(\texttt{let}_c\ ((c_1\ (\texttt{cnt}\ (n_{1,1}\ \ldots)\ e_1'))))\ e'))$

$\quad$if $[\![e_i]\!] = (\texttt{let}_f\ (fs_i)\ e_i')$

$\quad$and $[\![e]\!] = (\texttt{let}_f\ (fs)\ e')$

$[\![(\texttt{let}_f\ ((f_1\ (\texttt{fun}\ (n_{1,1}\ \ldots)\ e_1))\ \ldots)\ e)]\!] =$

$\quad(\texttt{let}_f\ ((f_1\ (\texttt{fun}\ (n_{1,1}\ \ldots)\ e_1')\ \ldots\ fs_1\ \ldots\ fs)\ e')$

$\quad$if $[\![e_i]\!] = (\texttt{let}_f\ (fs_i)\ e_i')$

$\quad$and $[\![e]\!] = (\texttt{let}_f\ (fs)\ e')$

$[\![e]\!]$ *when* e *is any other kind of expression* =

$\quad(\texttt{let}_f\ ()\ e)$

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.

`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)
```

`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)
```

env. initialization

env. extraction

closure creation

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