Closure conversion

or: values representation for functions

Advanced Compiler Construction

Michel Schinz – 2024–03–14

Functions

Functions

All languages offer functions, but with varying degrees of power, for example:

- in C, they can be passed as arguments and returned, but not nested,
- in functional languages, they can be nested and survive the scope that defined them.

Consequences:

- functions in C are less powerful than in functional languages (e.g. one can't define function composition),
- functions in C are easier to represent, as a code pointer is enough.

Example

The following L_3 example illustrates the challenges of representing functions in functional languages:

```
(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
```

Closures

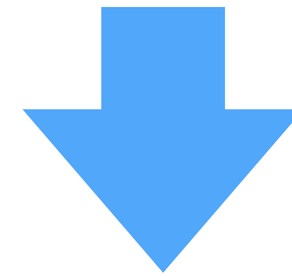
Closures

A simple code pointer cannot represent the functions returned by `make-adder` – at least not without run time code generation.

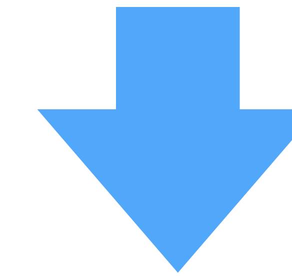
That code pointer must be paired with an **environment** giving the values of the free variables, here `x`, in what is called a **closure**.

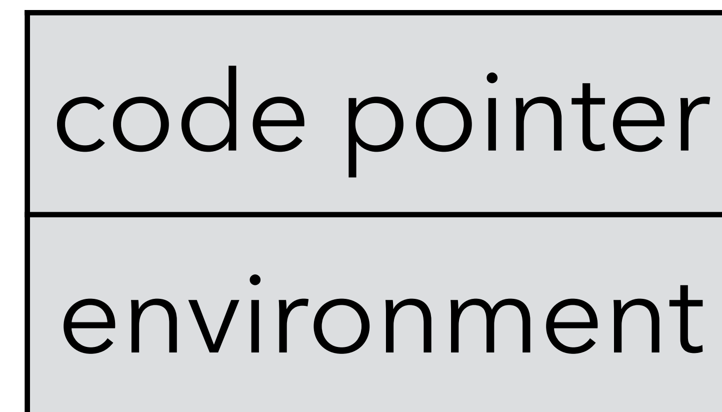
Closures

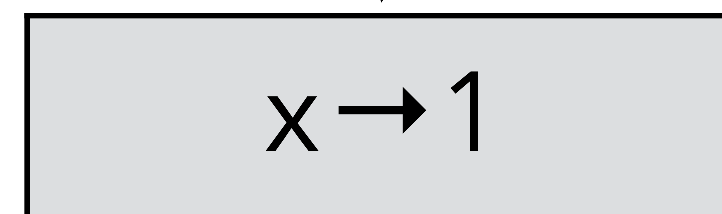
`(make-adder 1)`



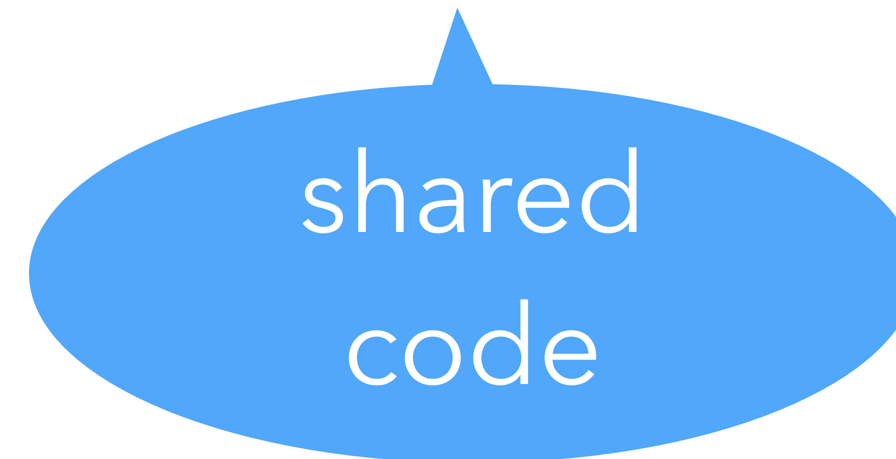
`(make-adder -1)`

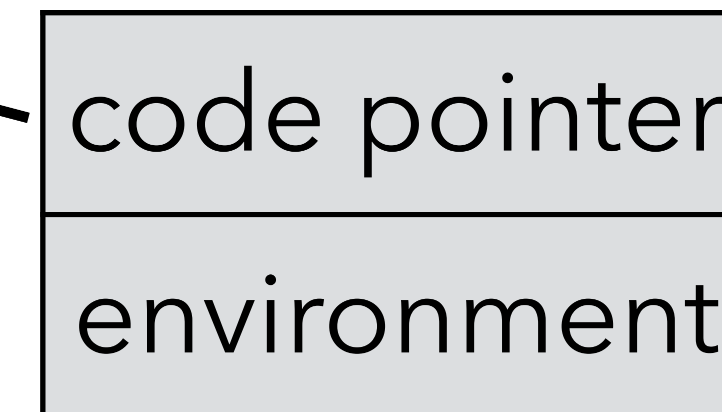


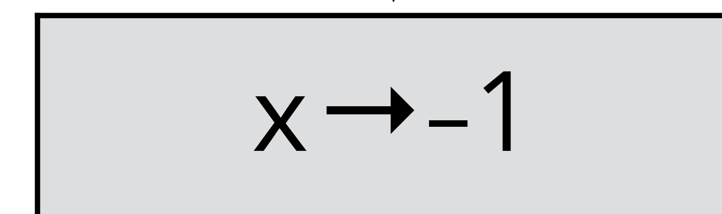




compiled code for
```
(fun (y)
  (@+ x y))
```







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

Introducing closures

Using closures instead of function pointers impacts:

- function abstraction, which must build and return a closure instead of a simple code pointer,
- function application, which must extract the code pointer from the closure, and invoke it with the environment as an additional argument.

Representing closures

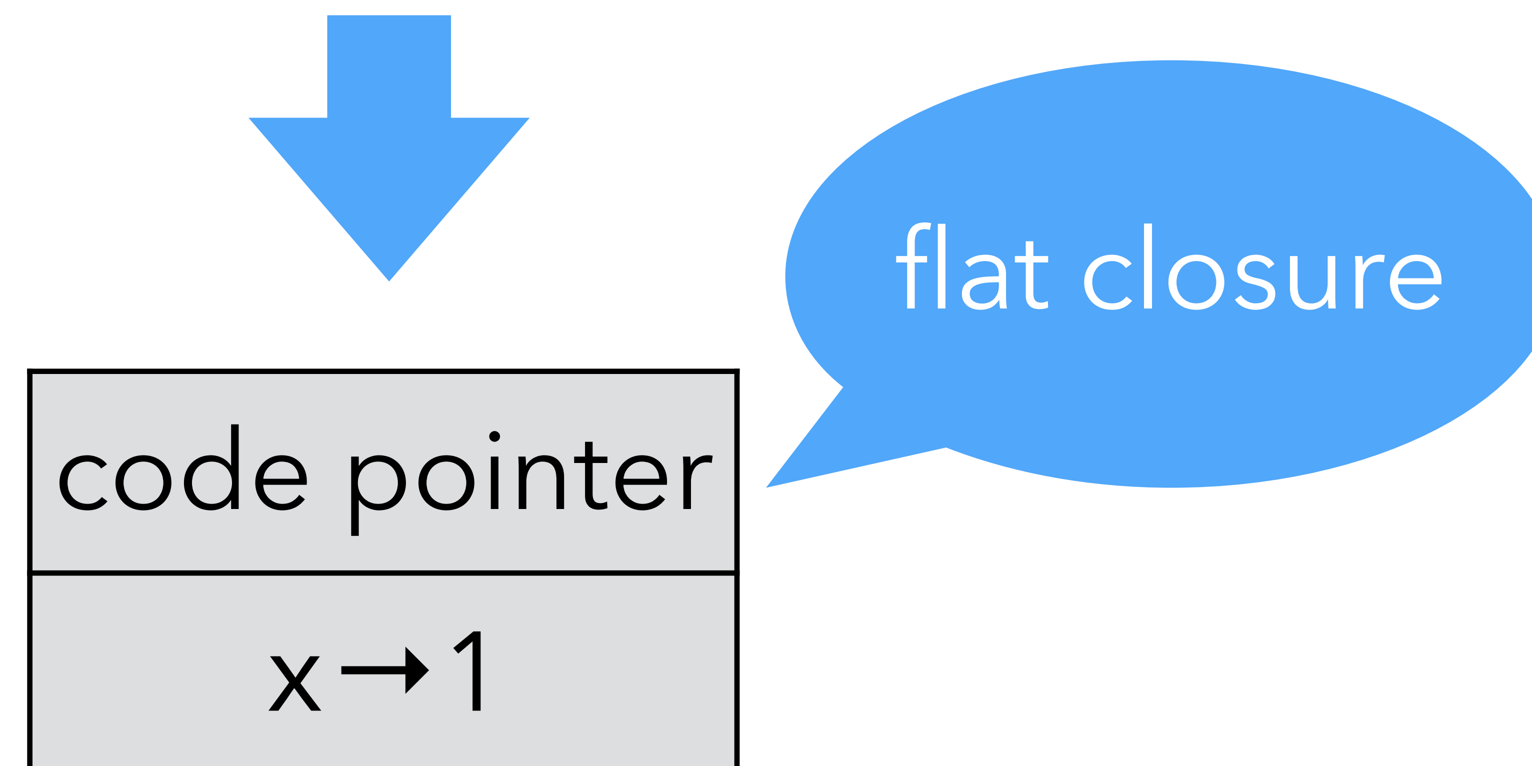
Notice that:

- since the code pointer is accessed during function application, it must be at a known location,
- since the environment is only accessed by the function itself, it can be laid out in an arbitrary way.

In particular, the environment can be "inlined" to obtain a flat closure.

Flat closures

In **flat** (or **one-block**) **closures**, the environment is “inlined” into the closure itself, instead of being referred from it. The closure itself plays the role of the environment.



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

Closures are introduced by a simplification phase called **closure conversion** that:

- takes a program in which functions can have free variables,
- produces a program where all functions are closed.

Source functions are represented as closures.

Note: This is just values representation for functions!

Free variables

The **free variables** of a function are the variables that are used in it but defined in some enclosing scope.

The `make-adder` example contains two functions:

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

The outer one has no free variable – it is closed.

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

Closing functions

Functions are closed by:

1. adding a parameter representing the environment, and
2. using it in the function’s body to access free variables.

Function abstraction and application must be adapted:

- 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.

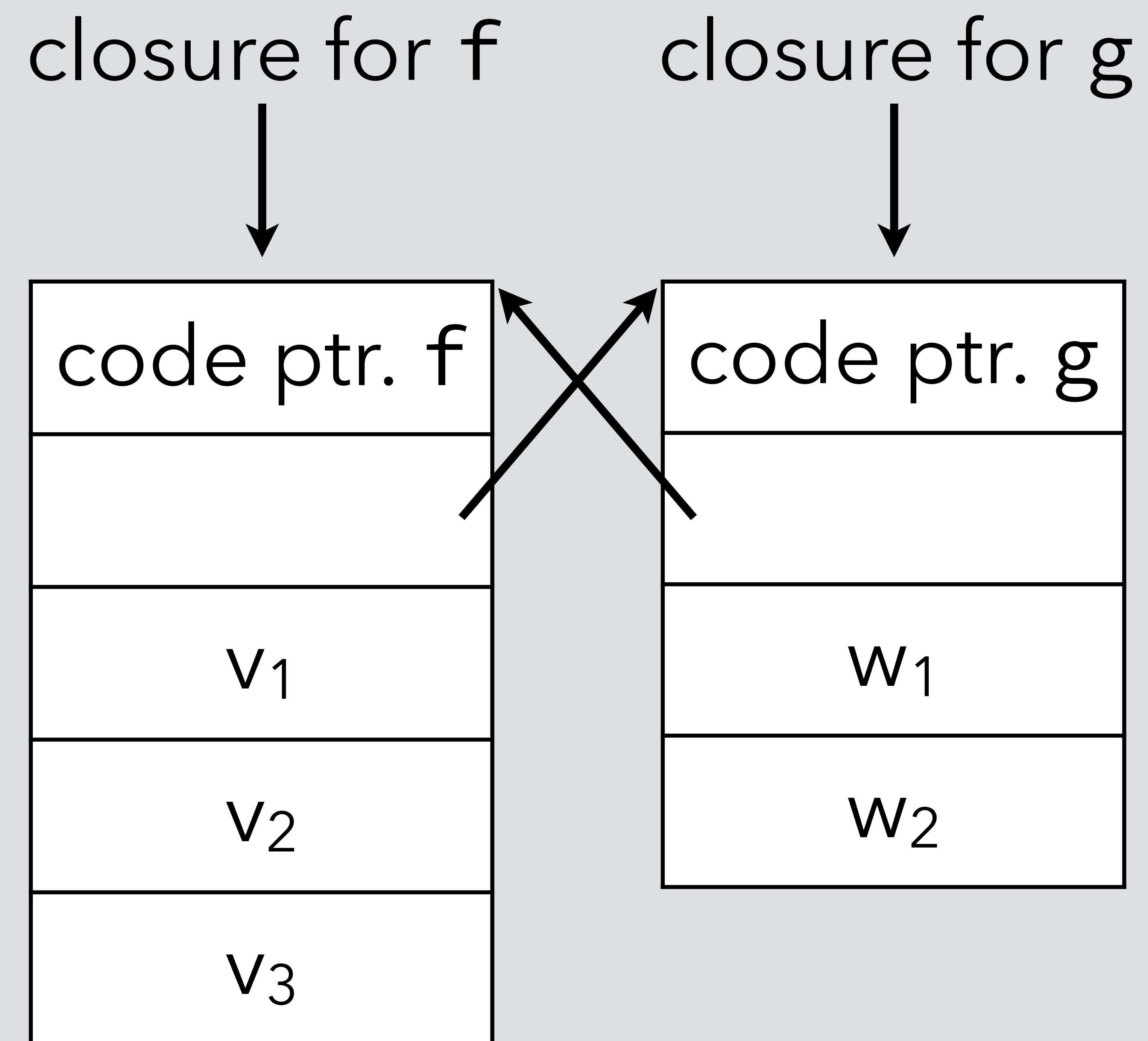
For example, in the following program, `f` needs access to the closure of `g`, and the other way around:

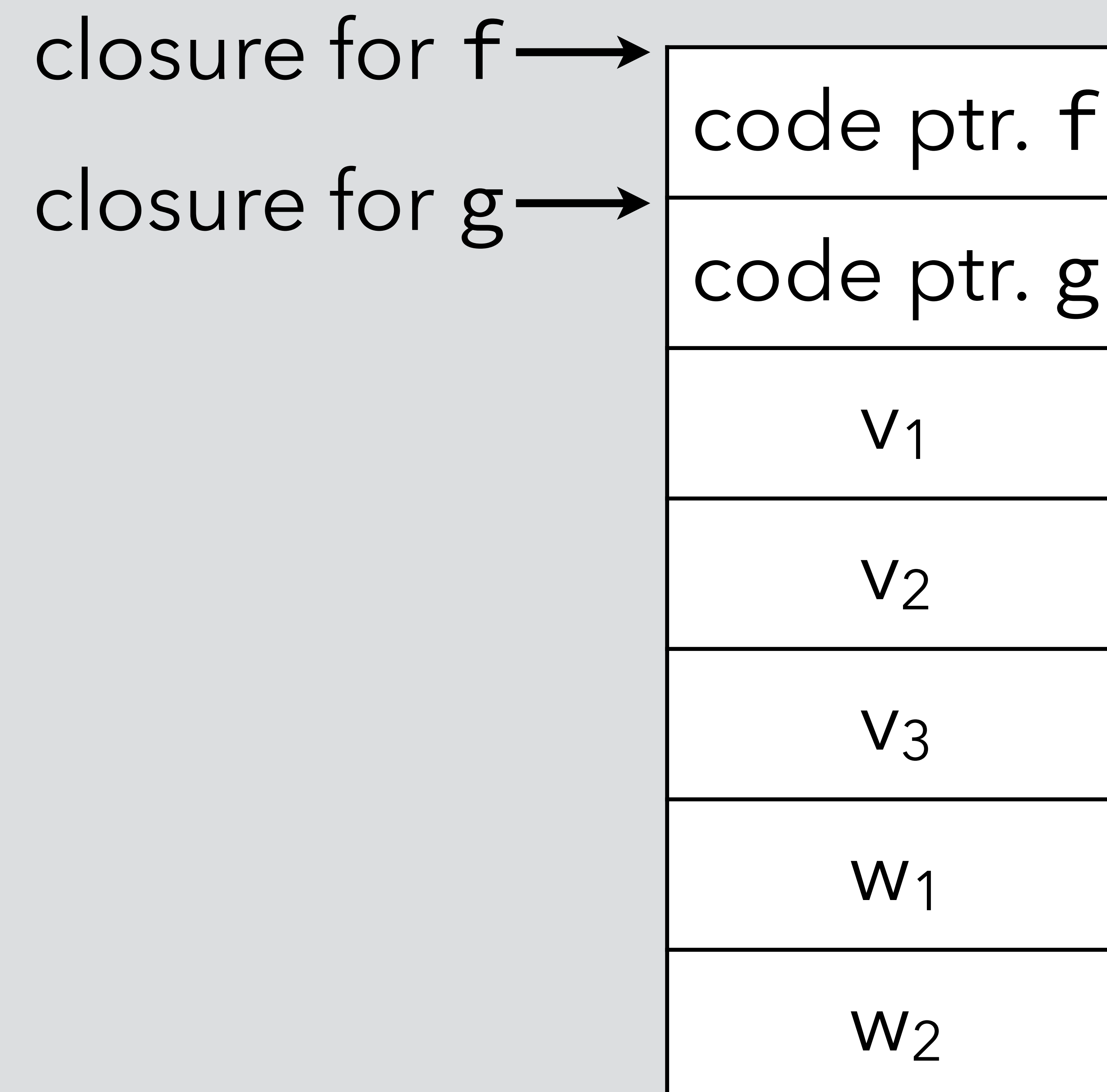
```
 (letrec ((f (fun (l) … (compose f g) …))
          (g (fun (l) … (compose g f) …)))
   …)
```

Solutions:

- use cyclic closures, or
- 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 chose `_function`.
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 values representation phase that takes care of representing function values.

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

CPS/L_3 free variables

F[`(let`$_p$ `((`n `(`p a_1 …`)))` e`)`] =

(F[e] \ { n }) ∪ F[a_1] ∪ …

F[`(let`$_c$ `((`c_1 `(cnt (`$n_{1,1}$ …`)` e_1`))` …`)` e`)`] =

F[e] ∪ (F[e_1] \ { $n_{1,1}$, … }) ∪ …

F[`(let`$_f$ `((`f_1 `(fun (`c_1 $n_{1,1}$ …`)` e_1`))` …`)` e`)`] =

(F[e] ∪ (F[e_1] \ { $n_{1,1}$, … }) ∪ …) \ { f_1, … }

F[`(app`$_c$ c a_1 …`)`] = F[a_1] ∪ …

F[`(app`$_f$ a c a_1 …`)`] = F[a] ∪ F[a_1] ∪ …

F[`(if (`p a_1 …`)` c_t c_e`)`] = F[a_1] ∪ …

F[`(halt` a`)`] = F[a]

F[n] = { n } *if* n *is a name*

F[l] = {} *if* l *is a literal*

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

Function definition

```
⟦(let_f ((f1 (fun (c1 n1,1 …) e1)) …) e)⟧ =
   (let_f ((w1 (fun (c1 env1 n1,1 …)
                 (let* ((v1 (block-get env1 1))
                        …)
                   ⟦e1⟧{f1→env1}{FV1(0)→v1}{…}))
           …)
     (let* ((f1 (block-alloc #__function |FV1|+1))
            …
            (t1 (block-set! f1 0 w1))
            (t2 (block-set! f1 1 FV1(0)))
            …)
       ⟦e⟧))
```

closed f_1

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:

```
⟦(appf a nc a1 ...)⟧ =
  (letp ((f (block-get ⟦a⟧ 0)))
    (appf f nc ⟦a⟧ ⟦a1⟧ ...))
```

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 `_function`.

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

```
(def function?
  (fun (o)
    (and (@block? o)
         (@= #__function (@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 below.

Translation inefficiencies

Applied to the CPS version of the `make-adder` example:

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

the simple translation creates a closure for the outer function, which is not needed as it is closed and never used as a value.

(Even if it **escaped**, i.e. was used as a value, the call could avoid going through the closure, as it is a **known function** here).

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 extracts the free variables from the environment and passes them as 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 function definition

```
⟦(letf ((f1 (fun (c1 n1,1 ...) e1)) ...) e)⟧ =
  (letf ((w1 (fun (c1 n1,1 ... u1 ...)
                  ⟦e1⟧{FV1(0)→u1}{...}))
         (s1 (fun (c1 env1 n11 ...)
                  (let* ((v1 (block-get env1 1))
                         ...)
                    (appf w1 c1 n11 ... v1 ...))))
         ...)
    (let* ((f1 (block-alloc #_function |FV1|+1))
           ...
           (t1 (block-set! f1 0 s1))
           ...)
      ⟦e⟧))
```

worker

wrapper

Improved function application

When translating function application, if the function being applied is **known** (i.e. bound by an enclosing `let_f`), its worker can be used directly:

$[\![$`(app_f a n_c a_1 …)`$]\!]$ = *if* a *is the name of a known function, with worker* a_w

```
(app_f a_w n_c ⟦a_1⟧ … FV_n(0) …)
```

otherwise, the closure has to be used, as before:

$[\![$`(app_f a n_c a_1 …)`$]\!]$ = *otherwise*

```
(let_p ((f (block-get ⟦a⟧ 0)))
  (app_f f n_c ⟦a⟧ ⟦a_1⟧ …))
```

Free variables

The improved translation makes the computation of free variables slightly more difficult, because:

- if a function `f` calls a known function `g`, it has to pass it its free variables as arguments,
- the free variables of `g` now become free variables of `f`,
- they must therefore be added to `f`'s arguments, impacting its callers – which could include `g`,
- and so on…

Hoisting
CPS/L_3 functions

Function hoisting

After closure conversion, all functions in the program are closed, and can therefore be hoisted to a single outer `let`$_f$.

Afterwards, the program has the following simple form:

(**let**$_f$ (*all functions of the program*)
  main program code)

where *main program code* does not contain any function definition (`let`$_f$ expression).

This simplifies the shape of the program and the job of later phases.

CPS/L_3 hoisting (1)

```
⟦(letₚ ((n (p a₁ …))) e)⟧ =
   (let_f (fs)
     (letₚ ((n (p a₁ …))) e’))
   if ⟦e⟧ = (let_f (fs) e’)
```

CPS/L$_3$ hoisting (2)

⟦`(let`$_c$ `((`c_1 `(cnt (`$n_{1,1}$ …`)` e_1`))` …`)` e`)`⟧ =

  `(let`$_f$ `(`fs_1 … fs`)`

    `(let`$_c$ `((`c_1 `(cnt (`$n_{1,1}$ …`)` e_1'`))` …`)` e'`))`

  if ⟦e_i⟧ = `(let`$_f$ `(`fs_i`)` e_i'`)`

  and ⟦e⟧ = `(let`$_f$ `(`fs`)` e'`)`

⟦`(let`$_f$ `((`f_1 `(fun (`$n_{1,1}$ …`)` e_1`))` …`)` e`)`⟧ =

  `(let`$_f$ `((`f_1 `(fun (`$n_{1,1}$ …`)` e_1'`))` … fs_1 … fs`)` e'`)`

  if ⟦e_i⟧ = `(let`$_f$ `(`fs_i`)` e_i'`)`

  and ⟦e⟧ = `(let`$_f$ `(`fs`)` e'`)`

⟦e⟧ *when* e *is any other kind of expression* =

  `(let`$_f$ `()` e`)`

Closures and objects

Closures and objects

A closure can be seen as an object with:

- a single method, containing the code of the closure,
- a set of fields: the environment.

Therefore, anonymous nested classes can be used to simulate closures!

Problem: the syntax is too heavyweight to be used often. Java (≥ 8), Scala, etc. offer special syntax for anonymous functions, which are translated to nested classes.

Adder maker 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)
```

Translated adder

(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)