Static scoping
Scoping in Hofl

Theory of Programming Languages
Computer Science Department
Wellesley College

Table of contents

Introduction

Contour model

Substitution model

Environment model

Scoping in HOFL

In many cases, the connection between reference occurrences and binding occurrences is clear from the meaning of the binding constructs. For instance, in the HOFL abstraction

```
(fun (a b) (bind c (+ a b) (div c 2)))
```

it is clear that the `a` and `b` within `(+ a b)` refer to the parameters of the abstraction and that the `c` in `(div c 2)` refers to the variable introduced by the `bind` expression.

Scoping in HOFL

However, the situation becomes murkier in the presence of functions whose bodies have free variables. Consider the following HOFL program:

```
(hofl (a)
  (bind add-a (fun (x) (+ x a))
    (bind a (+ a 10)
      (add-a (* 2 a)))))
```

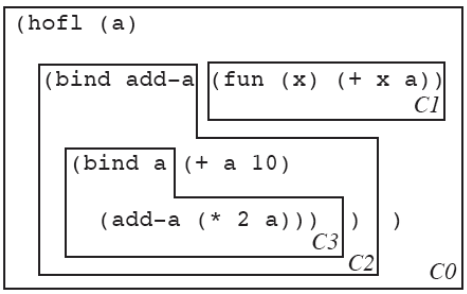
The `add-a` function is defined by the abstraction `(fun (x) (+ x a))`, which has a free variable `a`. The question is: which binding occurrence of `a` in the program does this free variable refer to? Does it refer to the program parameter `a` or the `a` introduced by the `bind` expression?

Scoping mechanisms

- A scoping mechanism determines the binding occurrence in a program associated with a free variable reference within a function body.
- We will study two scoping mechanisms in the context of the HOFL language: static scoping and dynamic scoping To simplify the discussion, we will initially consider
- HOFL programs that do not use the `bindrec` construct. We will study recursive bindings in more detail later the semester.

Static scoping: The contour model

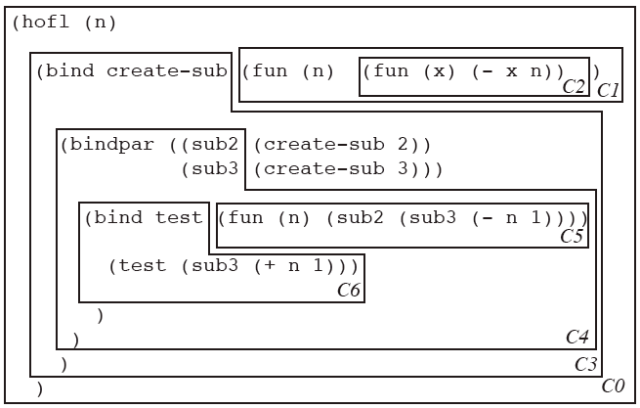
- In static scoping, the meaning of every variable reference is determined by the lexical contour boxes.



- For example, above is the contour diagram associated with the `add-a` example. The reference to `a` in the expression `(+ x a)` lies within contour boxes C_1 and C_0. C_1 does not bind `a`, but C_0 does, so the `a` in `(+ x a)` refers to the `a` bound by `(hofl (a) ... )`.

Lexical scoping

As another example, consider the contours associated with `create-sub`:



Static scoping is also known as lexical scoping because the meaning of any reference occurrence is apparent from the lexical structure of the program.

A substitution model for HOFL

The same substitution model used to explain the evaluation of OCAML, BINDEX, and VALEX can be used to explain the evaluation of statically scoped HOFL expressions that do not contain `bindrec`.

```
(hofl (a)
  (bind add-a (fun (x) (+ x a))
    (bind a (+ a 10)
      (add-a (* 2 a))))) run on [3]
; Here and below, assume a ''smart'' substitution that
; performs renaming only when variable capture is possible.
⇒ (bind add-a (fun (x) (+ x 3))
    (bind a (+ 3 10)
      (add-a (* 2 a))))
⇒* (bind a 13 ((fun (x) (+ x 3)) (* 2 a)))
⇒ ((fun (x) (+ x 3)) (* 2 13))
⇒ ((fun (x) (+ x 3)) 26)
⇒ (+ 26 3)
⇒ 29
```

As a second example, consider `create-sub`

```
(hofl (n)
  (bind create-sub (fun (n) (fun (x) (- x n)))
    (bindpar ((sub2 (create-sub 2))
              (sub3 (create-sub 3)))
      (bind test (fun (n) (sub2 (sub3 (- n 1))))
        (test (sub3 (+ n 1)))))))) run on [12]
⇒ (bind create-sub (fun (n) (fun (x) (- x n)))
    (bindpar ((sub2 (create-sub 2))
              (sub3 (create-sub 3)))
      (bind test (fun (n) (sub2 (sub3 (- n 1))))
        (test (sub3 (+ 12 1))))))
⇒* (bindpar ((sub2 ((fun (n) (fun (x) (- x n))) 2))
             (sub3 ((fun (n) (fun (x) (- x n))) 3)))
    (bind test (fun (n) (sub2 (sub3 (- n 1))))
      (test (sub3 13))))
⇒ (bindpar ((sub2 (fun (x) (- x 2)))
            (sub3 (fun (x) (- x 3))))
    (bind test (fun (n) (sub2 (sub3 (- n 1))))
      (test (sub3 13))))
⇒ (bind test (fun (n) ((fun (x) (- x 2)) ((fun (x) (- x 3)) (- n 1))))
    (test ((fun (x) (- x 3)) 13))))
⇒ ((fun (n) ((fun (x) (- x 2)) ((fun (x) (- x 3)) (- n 1))))
      ((fun (x) (- x 3)) 13))
```


The example concludes

```
⇒ ((fun (n) ((fun (x) (- x 2)) ((fun (x) (- x 3)) (- n 1))))
      ((fun (x) (- x 3)) 13))
⇒ ((fun (n) ((fun (x) (- x 2)) ((fun (x) (- x 3)) (- n 1)))) (- 13 3))
⇒ ((fun (n) ((fun (x) (- x 2)) ((fun (x) (- x 3)) (- n 1)))) 10)
⇒ ((fun (x) (- x 2)) ((fun (x) (- x 3)) (- 10 1)))
⇒ ((fun (x) (- x 2)) ((fun (x) (- x 3)) 9))
⇒ ((fun (x) (- x 2)) (- 9 3))
⇒ ((fun (x) (- x 2)) 6)
⇒ (- 6 2)
⇒ 4
```

Formalizing the HOFL *substitution model: Abstract syntax*

```
type var = string
type pgm = Pgm of var list * exp (* param names, body *)
and exp =
    Lit of valu (* integer, boolean, character, string, and list literals *)
  | Var of var (* variable reference *)
  | PrimApp of primop * exp list (* primitive application with rator, rands *)
  | If of exp * exp * exp (* conditional with test, then, else *)
  | Abs of var * exp (* function abstraction *)
  | App of exp * exp (* function application *)
  | Bindrec of var list * exp list * exp (* recursive bindings *)

and valu =
    Int of int
  | Bool of bool
  | Char of char
  | String of string
  | Symbol of string
  | List of valu list
  | Fun of var * exp * valu Env.env (* formal, body, and environment *)
    (* environment component is ignored in the substitution-model interpreter *

and primop = Primop of var * (valu list -> valu)
```


HOFL `subst` *function*

```
(* val subst : exp -> exp Env.env -> exp *)
let rec subst exp env =
  match exp with
    Lit i -> exp
  | Var v -> (match Env.lookup v env with Some e -> e | None -> exp)
  | PrimApp(op,rands) -> PrimApp(op, map (flip subst env) rands)
  | If(tst,thn,els) -> If(subst tst env, subst thn env, subst els env)
  | Abs(fml,body) ->
      let fml' = fresh fml in Abs(fml', subst (rename1 fml fml' body) env)
  | App(rator,rand) -> App(subst rator env, subst rand env)
  | Bindrec(names,defns,body) ->
      let names' = map fresh names in
        Bindrec(names', map (flip subst env) (map (renameAll names names') defn
                subst (renameAll names names' body) env)

and subst1 newexp name exp = subst exp (Env.make [name] [newexp])

and substAll newexps names exp = subst exp (Env.make names newexps)

and rename1 oldname newname exp = subst1 (Var newname) oldname exp

and renameAll olds news exp = substAll (map (fun s -> Var s) news) olds exp
```

Substitution model evaluator in Hofl*:* run

```
(* val run : Hofl.pgm -> int list -> valu *)
let rec run (Pgm(fmls,body)) ints =
  let flen = length fmls
  and ilen = length ints
  in
    if flen = ilen then
      eval (substAll (map (fun i -> Lit (Int i)) ints) fmls body)
    else
      raise (EvalError ("Program expected " ^ (string_of_int flen)
                        ^ " arguments but got " ^ (string_of_int ilen)))
```

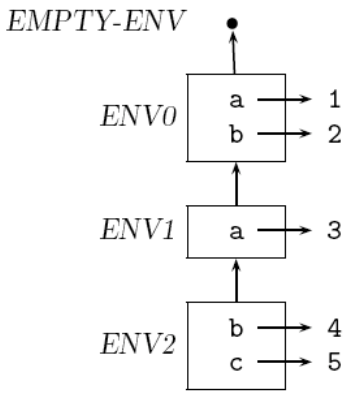

Substitution model evaluator in Hofl*:* eval

```
(* val eval : Hofl.exp -> valu *)
and eval exp =
  match exp with
    Lit v -> v
  | Var name -> raise (EvalError("Unbound variable: " ^ name))
  | PrimApp(op, rands) -> (primopFunction op) (map eval rands)
  | If(tst,thn,els) ->
     (match eval tst with
       Bool b -> if b then eval thn else eval els
     | v -> raise (EvalError ("Non-boolean test value "
                              ^ (valuToString v)
                              ^ " in if expression"))
     )
  | Abs(fml,body) -> Fun(fml,body,Env.empty) (* No env needed in subst model *)
  | App(rator,rand) -> apply (eval rator) (eval rand)
  | Bindrec(names,defns,body) -> ... see discussion of bindrec ...

and apply fcn arg =
  match fcn with
    Fun(fml,body,_) -> eval (subst1 (Lit arg) fml body)
                       (* Lit converts any argument valu
                          into a literal for purposes of substitution *)
  | _ -> raise (EvalError ("Non-function rator in application: "
```

HOFL *environment model*

- We would like to be able to explain static scoping within the environment model of evaluation.
- To do so, it is helpful to draw an environment as a linked chain of environment frames, where each frame has a set of name/value bindings and each frame has a single parent frame.
- There is a distinguished empty frame that terminates the chain, much as an empty list terminates a linked list.



The secret to life

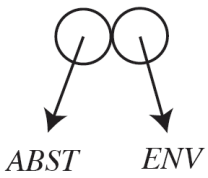
It turns out that any scoping mechanism is determined by how the following two questions are answered within the environment model:

1. What is the result of evaluating an abstraction in an environment?
2. When creating a frame to model the application of a function to arguments, what should the parent frame of the new frame be?

For static scoping

In the case of static scoping, answering these questions yields the following rules:

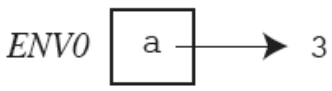
1. Evaluating an abstraction *ABS* in an environment *ENV* returns a closure that pairs together *ABS* and *ENV*. The closure "remembers" that *ENV* is the environment in which the free variables of *ABS* should be looked up.



2. To apply a closure to arguments, create a new frame that contains the formal parameters of the abstraction of the closure bound to the argument values. The parent of this new frame should be the environment remembered by the closure. That is, the new frame should extend the environment where the closure was born, not (necessarily) the environment in which the closure was called.

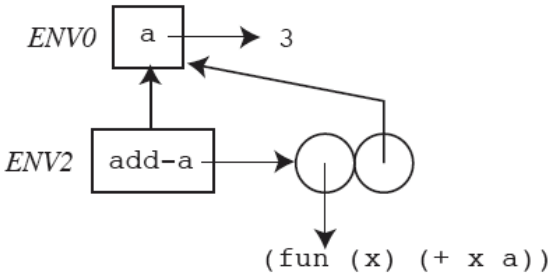
Return of `add-a`

We will show these rules in the context of using the environment model to explain executions of the two programs from above. First, consider running the `add-a` program on the input 3. This evaluates the body of the `add-a` program in an environment ENV_0 binding a to 3:



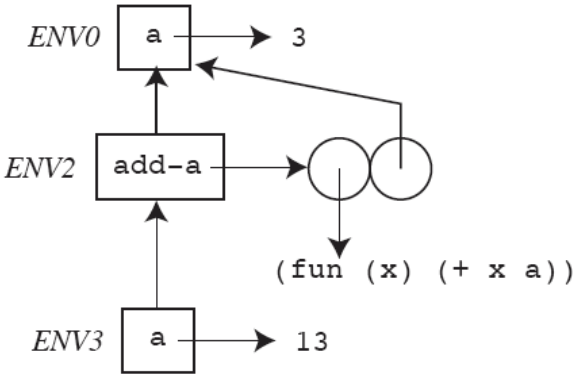
Evaluting the `bind add-a`

To evaluate the `(bind add-a ... )` expression, we first evaluate `(fun (x) (+ x a))` in ENV_0. According to rule 1 from above, this should yield a closure pairing the abstraction with ENV_0. A new frame ENV_2 should then be created binding add-a to the closure:



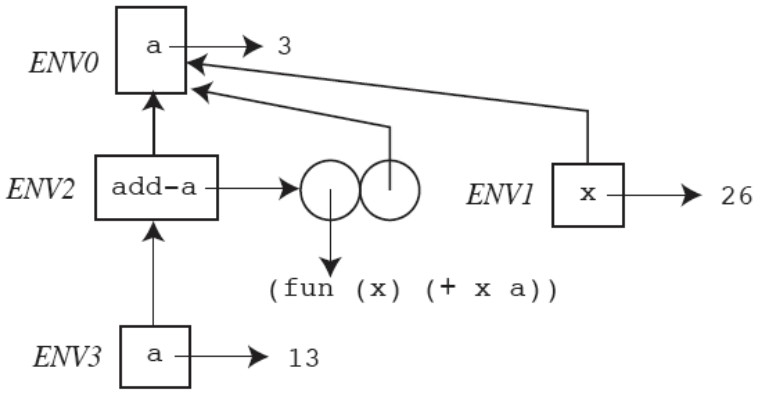
Next comes the expression `bind a`

Next the expression `(bind a ...)` is evaluated in ENV_2. First the definition `(+ a 10)` is evaluated in ENV_1, yielding 13. Then a new frame ENV_3 is created that binds a to 13:



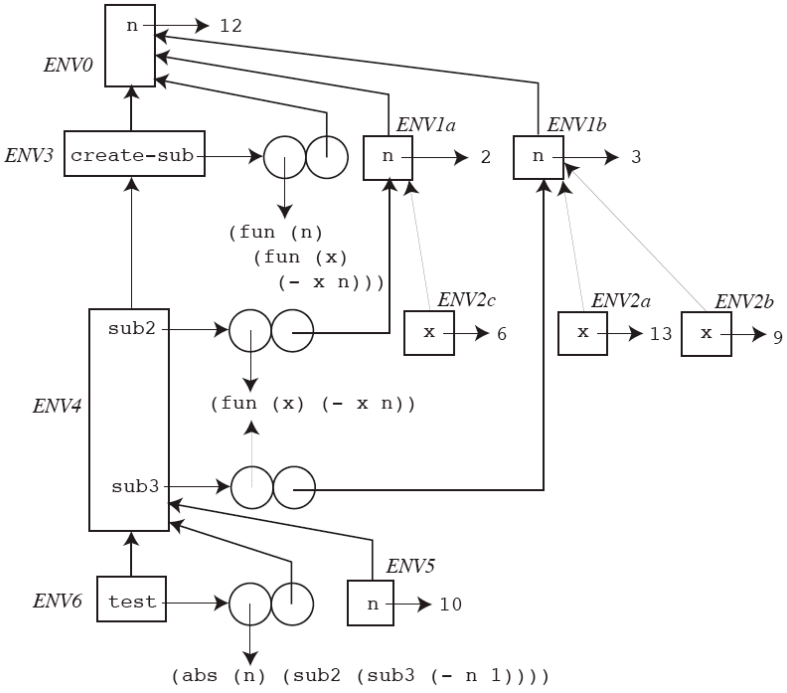
`bindrec` *deserves special attention*

Finally the function application `(add-a (* 2 a))` is evaluated in ENV_3.



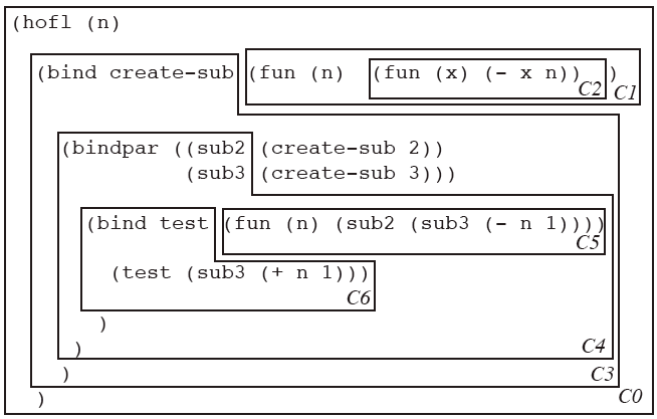
As the final step, the abstraction body `(+ x a)` is evaluated in ENV_1. Since `x` evaluates to 26 in ENV_3 and a evaluates to 3, the final answer is 29.

We could play the same game with `create-sub`



Notice that ...

The environment names have been chosen to underscore the fact that whenever environment frame ENV_i has a parent pointer to environment frame ENV_j in the environment model, the corresponding contour C_i is nested directly inside of C_j within the contour model.



Interpreter implementation of the environment model

Rules 1 and 2 of the previous section are easy to implement in an environment model interpreter.

```
(* val eval : Hofl.exp -> valu Env.env -> valu *)
and eval exp env =
  match exp with
      :
  | Abs(fml,body) -> Fun(fml,body,env) (* make a closure *)
  | App(rator,rand) -> apply (eval rator env) (eval rand env)
      :

and apply fcn arg =
  match fcn with
    Fun(fml,body,senv) -> eval body (Env.bind fml arg senv)
  | _ -> raise (EvalError ("Non-function rator in application: "
                           ^ (valuToString fcn)))
```

Note that it is not necessary to pass `env` as an argument to `funapply`, because static scoping dictates that the call-time environment plays no role in applying the function.