CS251 Programming Languages **Handout # 34**
Prof. Lyn Turbak **April 21, 2005**
Wellesley College

Hofl: First-class Functions and Scoping

This is a preliminary version that does not include a discussion of bindrec.

HOFL (Higher Order Functional Language) is a language that extends VALEX with first-class functions and a recursive binding construct. We study HOFL to understand the design and implementation issues involving first-class functions, particularly the notions of static vs. dynamic scoping and recursive binding. Later, we will consider languages that support more restrictive notions of functions than HOFL.

Although HOFL is a "toy" language, it packs a good amount of expressive punch, and could be used for many "real" programming purposes. Indeed, it is very similar to the Scheme programming language, and it is powerful enough to write interpreters for all the mini-languages we have studied, including HOFL itself!

1 An Overview of HOFL

The HOFL language extends VALEX with the following features:

1. Anonymous first-class curried functions and a means of applying these functions;
2. A `bindrec` form for definining mutually recursive values (typically functions);
3. A `load` form for loading definitions from files.

The full grammar of HOFL is presented in Fig. 1. The syntactic sugar of HOFL is defined in Fig. 2.

1.1 Abstractions and Function Applications

In HOFL, anonymous first-class functions are created via

`(abs` I_{formal} E_{body}`)`

This denotes a function of a single argument I_{formal} that computes E_{body}. It corresponds to the OCAML notation `fun` I_{formal} `->` E_{body}.

Function application is expressed by the parenthesized notation `(`E_{rator} E_{rand}`)`, where E_{rator} is an arbitrary expression that denotes a function, and E_{rand} denotes the operand value to which the function is applied. For example:

```
hofl> ((abs x (* x x)) (+ 1 2))
9
hofl> ((abs f (f 5)) (abs x (* x x)))
25
hofl> ((abs f (f 5)) ((abs x (abs y (+ x y))) 12))
17
```

The second and third examples highlight the first-class nature of HOFL function values.

The notation `(fun (`$I_1 \ldots I_n$`)` E_{body}`)` is syntactic sugar for curried abstractions and the notation `(`E_{rator} $E_1 \ldots E_n$`)` for $n \geq 2$ is syntactic sugar for curried applications. For example,

$P \in$ *Program*	
$P \rightarrow$ `(hofl (`$I_{formal_1} \ldots I_{formal_n}$`)` E_{body}`)`	Kernel Program
$P \rightarrow$ `(hofl (`$I_{formal_1} \ldots I_{formal_n}$`)` E_{body} $D_1 \ldots D_k$`)`	Sugared Program
$D \in$ *Definition*	
$D \rightarrow$ `(def` I_{name} E_{body}`)`	Basic Definition
$D \rightarrow$ `(def (`$I_{fcnName}$ $I_{formal_1} \ldots I_{formal_n}$`)` E_{body}`)`	Sugared Function Definition
$D \rightarrow$ `(load filename)`	File Load
$E \in$ *Expression*	
Kernel Expressions:	
$E \rightarrow L$	Literal
$E \rightarrow I$	Variable Reference
$E \rightarrow$ `(if` E_{test} E_{then} E_{else}`)`	Conditional
$E \rightarrow$ `(`O_{rator} $E_{rand_1} \ldots E_{rand_n}$`)`	Primitive Application
$E \rightarrow$ `(abs` I_{formal} E_{body}`)`	Function Abstraction
$E \rightarrow$ `(`E_{rator} E_{rand}`)`	Function Application
$E \rightarrow$ `(bindrec ((`I_{name_1} E_{defn_1}`) ... (`I_{name_n} E_{defn_n}`))` E_{body}`)`	Local Recursion
Sugar Expressions:	
$E \rightarrow$ `(fun (`$I_1 \ldots I_n$`)` E_{body}`)`, where $n \geq 0$	Curried Function
$E \rightarrow$ `(`E_{rator} $E_{rand_1} \ldots E_{rand_n}$`)`, where $n \geq 2$	Curried Application
$E \rightarrow$ `(`E_{rator}`)`	Nullary Application
$E \rightarrow$ `(bind` I_{name} E_{defn} E_{body}`)`	Local Binding
$E \rightarrow$ `(bindseq ((`I_{name_1} E_{defn_1}`) ... (`I_{name_n} E_{defn_n}`))` E_{body}`)`	Sequential Binding
$E \rightarrow$ `(bindpar ((`I_{name_1} E_{defn_1}`) ... (`I_{name_n} E_{defn_n}`))` E_{body}`)`	Parallel Binding
$E \rightarrow$ `(&&` E_1 E_2`)`	Short-Circuit And
$E \rightarrow$ `(||` E_1 E_2`)`	Short-Circuit Or
$E \rightarrow$ `(cond (`E_{test_1} E_{body_1}`) ... (`E_{test_n} E_{body_n}`) (else` $E_{default}$`))`	Multi-branch Conditional
$E \rightarrow$ `(list` E_1 ... E_n`)`	List
$E \rightarrow$ `(quote` S`)`	Quoted Expression
$S \in$ *S-expression*	
$S \rightarrow N$	S-expression Integer
$S \rightarrow C$	S-expression Character
$S \rightarrow R$	S-expression String
$S \rightarrow I$	S-expression Symbol
$S \rightarrow$ `(`S_1 ... S_n`)`	S-expression List
$L \in$ *Literal*	
$L \rightarrow N$	Numeric Literal
$L \rightarrow B$	Boolean Literal
$L \rightarrow C$	Character Literal
$L \rightarrow R$	String Literal
$L \rightarrow$ (sym I)	Symbolic Literal
$L \rightarrow$ `#e`	Empty List Literal
$O \in$ *Primitive Operator*: e.g., `+`, `<=`, `and`, `not`, `prep`	
$F \in$ *Function Name*: e.g., `f`, `sqr`, `+-and-*`	
$I \in$ *Identifier*: e.g., `a`, `captain`, `fib_n-2`	
$N \in$ *Integer*: e.g., `3`, `-17`	
$B \in$ *Boolean*: `#t` and `#f`	
$C \in$ *Character*: `'a'`, `'B'`, `'7'`, `'\n'`, `''''\\'`	
$R \in$ *String*: `"foo"`, `"Hello there!"`, `"The string \"bar\""`	

Figure 1: Grammar for the HOFL langauge.

$$
\begin{array}{rcl}
\texttt{(hofl (}I_{formal_1} \ldots I_{formal_n}\texttt{)}\ E_{body}\ \texttt{(def}\ I_1\ E_1\texttt{)}\ \ldots\ \texttt{(def}\ I_n\ E_n\texttt{))} & & \\
\qquad \rightsquigarrow\ \texttt{(hofl (}I_{formal_1} \ldots I_{formal_n}\texttt{)\ (bindrec ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{))} & & \\
\\
\texttt{(def (}I_{fcn}\ I_1 \ldots I_n\texttt{)}\ E_{body}\texttt{)} & \rightsquigarrow & \texttt{(def}\ I_{fcn}\ \texttt{(fun (}I_1 \ldots I_n\texttt{)}\ E_{body}\texttt{))} \\
\\
\texttt{(fun (}I_1\ I_2 \ldots I_n\texttt{)}\ E_{body}\texttt{)} & \rightsquigarrow & \texttt{(abs}\ I_1\ \texttt{(fun (}I_2 \ldots I_n\texttt{)}\ E_{body}\texttt{))}\text{, where } n \geq 2 \\
\texttt{(fun (}I\texttt{)}\ E_{body}\texttt{)} & \rightsquigarrow & \texttt{(abs}\ I\ E_{body}\texttt{)} \\
\texttt{(fun ()}\ E_{body}\texttt{)} & \rightsquigarrow & \texttt{(abs}\ I\ E_{body}\texttt{)}\text{, where } I \text{ is fresh} \\
\\
\texttt{(}E_{rator}\ E_{rand_1}\ E_{rand_2} \ldots E_{rand_n}\texttt{)} & \rightsquigarrow & \texttt{((}E_{rator}\ E_{rand_1}\texttt{)}\ E_{rand_2} \ldots E_{rand_n}\texttt{)}\text{, where } n \geq 2 \\
\texttt{(}E_{rator}\texttt{)} & \rightsquigarrow & \texttt{(}E_{rator}\ \texttt{\#f)} \\
\\
\texttt{(bind}\ I_{name}\ E_{defn}\ E_{body}\texttt{)} & \rightsquigarrow & \texttt{((abs}\ I_{name}\ E_{body}\texttt{)}\ E_{defn}\texttt{)} \\
\\
\texttt{(bindpar ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)} & \rightsquigarrow & \texttt{((fun (}I_1 \ldots I_n\texttt{)}\ E_{body}\texttt{)}\ E_1\ \ldots\ E_n\texttt{)} \\
\\
\texttt{(bindseq ((}I\ E\texttt{)}\ \ldots\texttt{)}\ E_{body}\texttt{)} & \rightsquigarrow & \texttt{(bind}\ I\ E\ \texttt{(bindseq (}\ldots\texttt{)}\ E_{body}\texttt{))} \\
\texttt{(bindseq ()}\ E_{body}\texttt{)} & \rightsquigarrow & E_{body} \\
\\
\texttt{(\&\&}\ E_{rand1}\ E_{rand2}\texttt{)} & \rightsquigarrow & \texttt{(if}\ E_{rand1}\ E_{rand2}\ \texttt{\#f)} \\
\\
\texttt{(||}\ E_{rand1}\ E_{rand2}\texttt{)} & \rightsquigarrow & \texttt{(if}\ E_{rand1}\ \texttt{\#t}\ E_{rand2}\texttt{)} \\
\\
\texttt{(cond (else}\ E_{default}\texttt{))} & \rightsquigarrow & E_{default} \\
\texttt{(cond (}E_{test}\ E_{default}\texttt{)}\ \ldots\texttt{)} & \rightsquigarrow & \texttt{(if}\ E_{test}\ E_{default}\ \texttt{(cond}\ \ldots\texttt{))} \\
\\
\texttt{(list)} & \rightsquigarrow & \texttt{\#e} \\
\texttt{(list}\ E_{hd}\ \ldots\texttt{)} & \rightsquigarrow & \texttt{(prep}\ E_{hd}\ \texttt{(list}\ \ldots\texttt{))} \\
\\
\texttt{(quote}\ \textit{int}\texttt{))} & \rightsquigarrow & \textit{int} \\
\texttt{(quote}\ \textit{char}\texttt{))} & \rightsquigarrow & \textit{char} \\
\texttt{(quote}\ \textit{string}\texttt{))} & \rightsquigarrow & \textit{string} \\
\texttt{(quote \#t)} & \rightsquigarrow & \texttt{\#t} \\
\texttt{(quote \#f)} & \rightsquigarrow & \texttt{\#f} \\
\texttt{(quote \#e)} & \rightsquigarrow & \texttt{\#e} \\
\texttt{(quote}\ \textit{sym}\texttt{)} & \rightsquigarrow & \texttt{(sym}\ \textit{sym}\texttt{)} \\
\texttt{(quote (}\textit{sexp1}\ \ldots\ \textit{sexpn}\texttt{))} & \rightsquigarrow & \texttt{(list (quote}\ \textit{sexp1}\texttt{)}\ \ldots\ \texttt{(quote}\ \textit{sexpn}\texttt{))}
\end{array}
$$

Figure 2: Desugaring rules for HOFL.

```
((fun (a b x) (+ (* a x) b)) 2 3 4)
```

is syntactic sugar for

```
((((abs a (abs b (abs x (+ (* a x) b)))) 2) 3) 4)
```

Nullary functions and applications are also defined as sugar. For example, `((fun () `E`))` is syntactic sugar for `((abs `I E`) #f)`, where I is a fresh variable. Note that `#f` is used as an arbitrary argument value in this desugaring.

In HOFL, `bind` is not a kernel form but is syntactic sugar for the application of a manifest abstraction. Unlike in VALEX, in HOFL the `bindpar` desugaring can be expressed via a high-level rule (also involving the application of a manifest abstraction). For example,

```
(bind c (+ a b) (* c c))
```

is sugar for

```
((abs c (* c c)) (+ a b))
```

and

```
(bindpar ((a (+ a b)) (b (- a b))) (* a b))
```

is sugar for

```
((fun (a b) (* a b)) (+ a b) (- a b)),
```

which is itself sugar for

```
(((abs a (abs b (* a b))) (+ a b)) (- a b)).
```

1.2 Local Recursive Bindings

Singly and mutually recursive functions can be defined anywhere (not just at top level) via the `bindrec` construct:

`(bindrec ((`I_{name_1} E_{defn_1}`) ... (`I_{name_n} E_{defn_n}`)) `E_{body}`)`

The `bindrec` construct is similar to `bindpar` and `bindseq` except that the scope of $I_{name}1 \ldots I_{name}$n includes *all* definition expressions $E_{defn}1 \ldots E_{defn}$n as well as E_{body}. For example, here is a definition of a recursive factorial function:

```
(hofl (x)
  (bindrec ((fact (abs (n)
                    (if (= n 0)
                        1
                        (* n (fact (- n 1)))))))
    (fact x)))
```

Here is an example involving the mutual recursion of two functions, `even?` and `odd?`:

```
(hofl (n)
  (bindrec ((even? (abs (x)
                     (if (= x 0)
                         #t
                         (odd? (- x 1)))))
            (odd? (abs (y)
                    (if (= y 0)
                        #f
                        (even? (- y 1))))))
    (list (even? n) (odd? n))))
```

To emphasize that `bindrec` need not be at top-level, here is program that abstracts over the `even?/odd?` example from above:

```
(hofl (n)
  (bind tester (abs (bool)
                 (bindrec ((test1 (fun (x)
                                    (if (= x 0)
                                        bool
                                        (test2 (- x 1)))))
                           (test2 (abs (y)
                                    (if (= y 0)
                                        (not bool)
                                        (test1 (- y 1))))))))
    (list ((tester #t) n) ((tester #f) n))))
```

To simplify the definition of values, especially functions, at top-level HOFL supports syntactic sugar for top-level program definitions. For example, the `fact` and `even?/odd?` examples can also be expressed as follows:

```
(hofl (x) (fact x)
  (def (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))))

(hofl (n) (list (even? n) (odd? n))
  (def (even? x)
    (if (= x 0)
        #t
        (odd? (- x 1))))
  (def (odd? y)
    (if (= y 0)
        #f
        (even? (- y 1)))))
```

The HOFL read-eval-print loop (REPL) accepts definitions as well as expressions. All definitions are considered to be mutually recursive. Any expression submitted to the REPL is evaluated in the context of a `bindrec` derived from all the definitions submitted so far. If there has been more than one definition with a given name, the most recent definition with that name is used. For example, consider the following sequence of REPL interactions:

```
hofl> (def three (+ 1 2))
three
```

For a definition, the response of the interpreter is the defined name. This can be viewed as an

acknowledgement that the definition has been submitted. The body expression of the definition is *not* evaluated yet, so if it contains an error or infinite loop, there will be no indication of this until an expression is submitted to the REPL later.

```
hofl> (+ three 4)
7
```

When the above expression is submitted, the result is the value of the following expression:

```
(bindrec ((three (+ 1 2)))
  (+ three 4))
```

Now let's define a function and then invoke it:

```
hofl> (def (sq x) (* x x))
sq
hofl> (sq three)
9
```

The value *9* is the result of evaluating the following expression:

```
(bindrec ((three (+ 1 2))
          (sq (abs x (* x x))))
  (sq three))
```

Let's define one more function and invoke it:

```
hofl> (def (sos a b) (+ (sq a) (sq b)))
sos
hofl> (sos three 4)
25
```

The value *25* is the resulf of evaluating the following expression:

```
(bindrec ((three (+ 1 2))
          (sq (abs x (* x x)))
          (sos (abs a (abs b (+ (sq a) (sq b))))))
  ((sos three) 4))
```

Note that it wasn't necessary to define `sq` before `sos`. They could have been defined in the opposite order, as long as no attempt was made to evaluate an expression containing `sos` before `sq` was defined.

1.3 Loading Definitions From Files

Typing sequences of definitions into the HOFL REPL can be tedious for any program that contains more than a few definitions. To facilitate the construction and testing of complex programs, HOFL supports the loading of definitions from files. Suppose that *filename* is a string literal (i.e., a character sequence delimited by double quotes) naming a file that contains a sequence of HOFL definitions. In the REPL, entering the directive `(load` *`filename`*`)` has the same effect as manually entering all the definitions in the file named *filename*. For example, suppose that the file named `"option.hfl"` contains the definitions in Fig. 3 and `"list-utils.hfl"` contains the definitions in Fig. 4. Then we can have the following REPL interactions:

```
(def none (sym *none*)) ; Use the symbol *NONE* to represent the none value.

(def (none? v)
  (if (sym? v)
      (sym= v none)
      #f))

(def (some? v) (not (none? v)))
```

Figure 3: The contents of the file `"option.hfl"` which contains an OCAML-like option data structure express in HOFL.

```
hofl> (load "option.hfl")
none
none?
some?
```

When a `load` directive is entered, the names of all definitions in the loaded file are displayed. These definitions are not evaluated yet, only collected for later.

```
hofl> (none? none)
#t

hofl> (some? none)
#f

hofl> (load "list-utils.hfl")
length
rev
nth
first
second
third
fourth
map
filter
gen
range
foldr
foldr2

hofl> (range 3 7)
(list 3 4 5 6 7)

hofl> (map (fun (x) (* x x)) (range 3 7))
(list 9 16 25 36 49)

hofl> (foldr (fun (a b) (+ a b)) 0 (range 3 7))
25

hofl> (filter some? (map (fun (x) (if (= 0 (% x 2)) x none)) (range 3 7)))p
(list 4 6)
```

In HOFL, a `load` directive may appear whereever a definition may appear. It denotes the sequence of definitions contains in the named file. For example, loaded files may themselves contain `load` directives for loading other files. The environment implementation in Fig. 5 loads the

```
(def (length xs)
  (if (empty? xs)
      0
      (+ 1 (length (tail xs)))))

(def (rev xs)
  (bindrec ((loop (fun (old new)
                    (if (empty? old)
                        new
                        (loop (tail old) (prep (head old) new))))))
    (loop xs #e)))

(def (nth n xs) ; Returns the nth element of a list (1-indexed)
  (if (= n 1)
      (head xs)
      (nth (- n 1) (tail xs))))

(def first (nth 1))
(def second (nth 2))
(def third (nth 3))
(def fourth (nth 4))

(def (map f xs)
  (if (empty? xs)
      #e
      (prep (f (head xs))
            (map f (tail xs)))))

(def (filter pred xs)
  (cond ((empty? xs) #e)
        ((pred (head xs))
         (prep (head xs) (filter pred (tail xs))))
        (else (filter pred (tail xs)))))

(def (gen next done? seed)
  (if (done? seed)
      #e
      (prep seed (gen next done? (next seed)))))

(def (range lo hi) (gen (fun (x) (+ x 1)) (fun (y) (> y hi)) lo))

(def (foldr binop null xs)
  (if (empty? xs)
      null
      (binop (head xs)
             (foldr binop null (tail xs)))))

(def (foldr2 ternop null xs ys)
  (if (|| (empty? xs) (empty? ys))
      null
      (ternop (head xs)
              (head ys)
              (foldr2 ternop null (tail xs) (tail ys)))))
```

Figure 4: The contents of the file `"list-utils.hfl"` which contains some classic list functions expressed in HOFL.

```
(load "option.hfl")
(load "list-utils.hfl")

(def env-empty (fun (name) none))

(def (env-bind name val env)
  (fun (n)
    (if (sym= n name)
        val
        (env n))))

(def (env-bind-all names vals env)
  (foldr2 env-bind env names vals))

(def (env-lookup name env) (env name))
```

Figure 5: The contents of the file `"env.hfl"`, which contains an functional implementation of environments expressed in HOFL.

files `"option.hfl"` and `"list-utils.hfl"`. `load` directives may also appear directly in a HOFL program. For example:

```
(hofl (a b)
  (filter some? (map (fun (x) (if (= 0 (% x 2)) x none)) (range a b)))
  (load "option.hfl")
  (load "list-utils.hfl"))
```

When applied to the argument list `[3;7]`, this program yields a HOFL list containing the integers `4` and `6`

1.4 A BINDEX Interpreter Written in HOFL

To illustrate that HOFL is suitable for defining complex programs, in Figs. 6 and 7 we present a complete interpreter for the BINDEX language written in HOFL. BINDEX expressions and programs are represented as tree structures encoded via HOFL lists, symbols, and integers. For example, the BINDEX averaging program can be expressed as the following HOFL list:

```
(list (sym bindex)
      (list (sym a) (sym b)) ; formals
      (list (sym /) ; body
            (list (sym +) (sym a) (sym b))
            2))
```

HOFL's LISP-inspired `quote` sugar allows such BINDEX programs to be written more perspicuously. For example, the above can be expressed as follows using `quote`:

```
(quote (bindex (a b) (/ (+ a b) 2)))
```

Here are some examples of the HOFL-based BINDEX interpreter in action:

```
hofl> (load "bindex.hfl")
... lots of names omitted ...
hofl> (run (quote (bindex (x) (* x x))) (list 5))
25
```

```
hofl> (run (quote (bindex (a b) (/ (+ a b) 2))) (list 5 15))
10
```

It is not difficult to extend the BINDEX interpreter to be a full-fledged HOFL interpreter. If we did this, we would have a HOFL interpreter defined in HOFL. An interpreter for a language written in that language is called a **meta-circular** interpreter. There is nothing strange or ill-defined about a meta-circular interpreter. Keep in mind that in order to execute a meta-circular interpreter for a language L, we must have an existing working implementation of L. For example, to execute a meta-circular HOFL interpreter, we could use a HOFL interpreter defined in OCAML.

```
(load "env.hfl") ; This also loads list-utils.hfl and option.hfl

(def (run pgm args)
  (bind fmls (pgm-formals pgm)
    (if (not (= (length fmls) (length args)))
        (error "Mismatch between expected and actual arguments"
               (list fmls args))
        (eval (pgm-body pgm)
              (env-bind-all fmls args env-empty)))))

(def (eval exp env)
  (cond ((lit? exp) (lit-value exp))
        ((var? exp)
         (bind val (env-lookup (var-name exp) env)
           (if (none? val)
               (error "Unbound variable" exp)
               val)))
        ((binapp? exp)
         (binapply (binapp-op exp)
                   (eval (binapp-rand1 exp) env)
                   (eval (binapp-rand2 exp) env)))
        ((bind? exp)
         (eval (bind-body exp)
               (env-bind (bind-name exp)
                         (eval (bind-defn exp) env)
                         env)))
        (else (error "Invalid expression" exp))))

(def (binapply op x  y)
  (cond ((sym= op (sym +)) (+ x y))
        ((sym= op (sym -)) (- x y))
        ((sym= op (sym *)) (* x y))
        ((sym= op (sym /)) (if (= y 0) (error "Div by 0" x) (/ x y)))
        ((sym= op (sym %)) (if (= y 0) (error "Rem by 0" x) (% x y)))
        (else (error "Invalid binop" op))))
```

Figure 6: Environment model interpreter for BINDEX expressed in HOFL, part 1.

2 Scoping Mechanisms

In order to understand a program, it is essential to understand the meaning of every name. This requires being able to reliably answer the following question: given a reference occurrence of a

```
;;;-------------------------------------------------------------------
;;; Abstract syntax

;;; Programs

(def (pgm? exp)
 (&& (list? exp)
     (&& (= (length exp) 3)
         (sym= (first exp) (sym bindex)))))

(def (pgm-formals exp) (second exp))
(def (pgm-body exp) (third exp))

;;; Expressions

;; Literals
(def (lit? exp) (int? exp))
(def (lit-value exp) exp)

;; Variables
(def (var? exp) (sym? exp))
(def (var-name exp) exp)

;; Binary Applications
(def (binapp? exp)
  (&& (list? exp)
      (&& (= (length exp) 3)
          (binop? (first exp)))))

(def (binapp-op exp) (first exp))
(def (binapp-rand1 exp) (second exp))
(def (binapp-rand2 exp) (third exp))

;; Local Bindings
(def (bind? exp)
  (&& (list? exp)
      (&& (= (length exp) 4)
          (&& (sym=? (first exp) (sym bind))
              (sym? (second exp))))))

(def (bind-name exp) (second name))
(def (bind-defn exp) (third name))
(def (bind-body exp) (fourth name))

;; Binary Operators
(def (binop? exp)
  (|| (sym= exp (sym +))
      (|| (sym= exp (sym -))
          (|| (sym= exp (sym *))
              (|| (sym= exp (sym /))
                  (sym= exp (sym %)))))))
```

Figure 7: Environment model interpreter for BINDEX expressed in HOFL, part 2.

name, which binding occurrence does it refer to?

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.

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?

A **scoping mechanism** determines the binding occurrence in a program associated with a free variable reference within a function body. In languages with block structure[1] and/or higher-order functions, it is common to encounter functions with free variables. Understanding the scoping mechanisms of such languages is a prerequisite to understand the meanings of programs written in these languages.

We will study two scoping mechanisms in the context of the HOFL language: **static scoping** and **dynamic scoping**. To simplify the discussion, here will will only consider HOFL programs that do not use the `bindrec` construct. We will study recursive bindings in more detail later.

3 Static Scoping

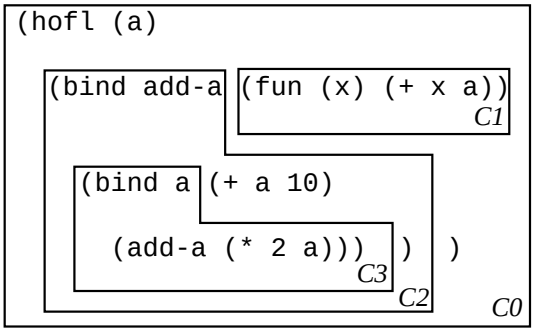
3.1 Contour Model

In **static scoping**, the meaning of every variable reference is determined by the lexical contour boxes introduced in Handout #28 on Bindex. To determine the binding occurrence of any reference occurrence of a name, find the innermost contour enclosing the reference occurrence that binds the name. This is the desired binding occurrence.

For example, below 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) ... )`. Similarly, it can be determined that:

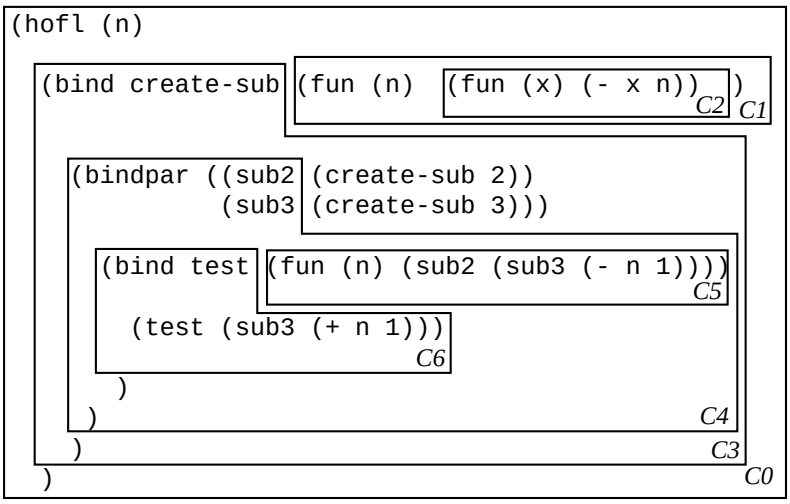
- the `a` in `(+ a 10)` refers to the `a` bound by `(hofl (a) ... )`;
- the `a` in `(* 2 a)` refers the `a` bound by `(bind a ... )`;
- the `x` in `(+ x a)` refers to the `x` bound by `(abs (x) ... )`.
- the `add-a` in `(add-a (* 2 a))` refers to the `add-a` bound by `(bind add-a ... )`.

[1] A language has block structure if functions can be declared locally within other functions. As we shall see later, a language can have block structure without having first-class functions.



Because the meaning of any reference occurrence is apparent from the lexical structure of the program, static scoping is also known as **lexical scoping**.

As another example of a contour diagram, consider the contours associated with the following program containing a `create-sub` function:



By the rules of static scope:

- the `n` in `(- x n)` refers to the `n` bound by the `(fun (n) ... )` of `create-sub`;
- the `n` in `(- n 1)` refers to the `n` bound by the `(fun (n) ... )` of `test`;
- the `n` in `(+ n 1)` refers to the `n` bound by `(hofl (n) ... )`.

3.2 Substitution Model

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`. (Handling `bindrec` is tricky in the substitution model, and will be considered later.)

For example, suppose we run the program containing the `add-a` function on the input 3. Then the substitution process yields:

```
(hofl (a)
  (bind add-a (fun (x) (+ x a))
    (bind a (+ a 10)
      (add-a (* 2 a))))) run on [3]
⇒ (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, suppose we run the program containing the `create-sub` function on the input 12. Then the substitution process yields:

```
(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))
⇒ ((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
```

We can formalize the HOFL substitution model by defining a substitution model evaluator in OCAML. Fig. 9 presents the abstract syntax and values used by the evaluator as well as the definition of substitution. The evaluator itself is presented in Fig. **??**. The third component of a `Fun` value, an environment, is not used in the substitution model but plays a very important role in the environment model. The omitted `bindrec` case will be explained later.

```
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 *)

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

let primopName (Primop(name,_)) = name
let primopFunction (Primop(_,fcn)) = fcn

(* 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') defns),
                subst (renameAll names names' body) env)

(* val subst1 : exp -> var -> exp -> exp *)
and subst1 newexp name exp = subst exp (Env.make [name] [newexp])

(* val substAll: exp list -> var list -> exp -> exp *)
and substAll newexps names exp = subst exp (Env.make names newexps)

(* val rename1 : var -> var -> exp -> exp *)
and rename1 oldname newname exp = subst1 (Var newname) oldname exp

(* val renameAll : var list -> var list -> exp -> exp *)
and renameAll olds news exp = substAll (map (fun s -> Var s) news) olds exp
```

Figure 8: OCAML data types for the abstract syntax of HOFL.

```
(* val run : Hofl.pgm -> int list -> int *)
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)))

(* 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 true -> eval thn
     | Bool false -> 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 (including lists & functions)
                          into a literal for purposes of substitution *)
  | _ -> raise (EvalError ("Non-function rator in application: "
                           ^ (valuToString fcn)))
```

Figure 9: Substitution model evaluator in HOFL.

3.3 Environment Model

We would like to be able to explain static scoping within the environment model of evaluation. Most evaluation rules of the environment model are independent of the scoping mechanism. Such rules are shown in Fig. 10.

Program Running Rule

- To run a HOFL program `(hofl (`I_1 ... I_n`)` E_{body}`)` on integers $i_1, \ldots, i_k$, return the result of evaluating E_{body} in an environment that binds the formal parameter names I_1 ...I_n respectively to the integer values $i_1, \ldots, i_k$.

Expression Evaluation Rules

- To evaluate a literal expression in any environment, return the value of the literal.
- To evaluate a variable reference expression I expression in environment ENV, return the value of looking up I in ENV. If I is not bound in ENV, signal an unbound variable error.
- To evaluate the conditional expression `(if` E_1 E_2 E_3`)` in environment ENV, we first evaluated E_1 in ENV. If the result is true, we return the result of evaluating E_2 in ENV; else we return the result of evaluating E_3 in ENV.
- To evaluate the primitive application `(`O_{rator} E_1 ... E_n`)` in environment ENV, we must first evaluate the operand expressions E_1 through E_n in ENV. We then return the result of applying the primitive operator primop to the resulting operand values.
- To evaluate the function application `(`E_0 E_1 ... E_n`)` in environment ENV, we must first evaluate the expressions E_0 through E_n in ENV. We then return the result of applying the function value to the operand values. (The details of what it means to apply a function is at the heart of scoping and, as we shall see, differs among scoping mechanisms.)

Although `bind`, `bindrec`, and `bindseq` can all be "desugared away", it is convenient to imagine that there are rules for evaluating these constructs directly:

- Evaluating `(bind` E_{name} E_{defn} E_{body}`)` in environment ENV is the result of evaluating E_{body} in the environment that results from extending ENV with a frame containing a single binding between E_{name} and the result of evaluating E_{defn} in ENV.
- A `bindpar` is evaluated similarly to `bind`, except that the new frame contains one binding for each of the name/defn pairs in the `bindpar`. As in `bind`, all defns of `bindpar` are evaluated in the original frame, not the extension.
- A `bindseq` expression should be evaluated as if it were a sequence of nested `bind`s.

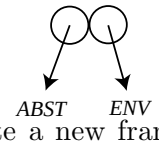
Figure 10: Environment model evaluation rules that are independent of the scoping mechanism.

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?

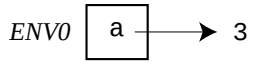
In the case of static scoping, answering these questions yields the following rules:

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; it is like an "umbilical cord" that connects the abstraction to its place of birth. We shall draw closures as a pair of circles, where the left circle points to the abstraction and the right circle points to the environment:

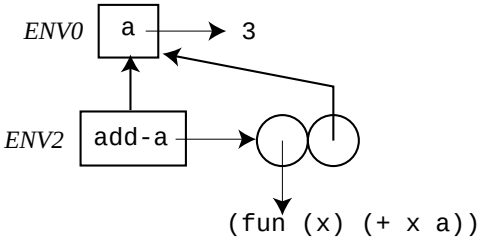


2. To apply a closure to arguments, create a new frame that contains the formal parameters of the abstraction of the closure bound to the argument values. The parent of this new frame should be the environment remembered by the closure. That is, the new frame should extend the environment where the closure was born, not (necessarily) the environment in which the closure was called. This creates the right environment for evaluating the body of the abstraction as implied by static scoping: the first frame in the environment contains the bindings for the formal parameters, and the rest of the frames contain the bindings for the free variables.

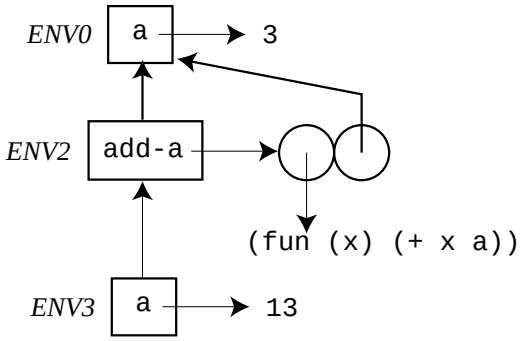
We will show these rules in the context of using the environment model to explain executions of the two programs from above. First, consider running the `add-a` program on the input 3. This evaluates the body of the `add-a` program in an environment ENV_0 binding a to 3:



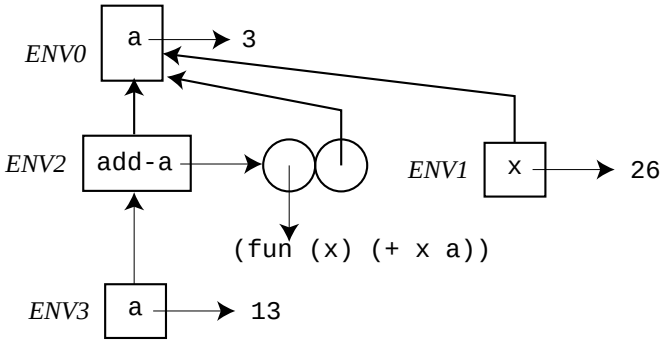
To evaluate the `(bind add-a ... )` expression, we first evaluate `(fun (x) (+ x a))` in ENV_0. According to rule 1 from above, this should yield a closure pairing the abstraction with ENV_0. A new frame ENV_2 should then be created binding add-a to the closure:



Next 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:

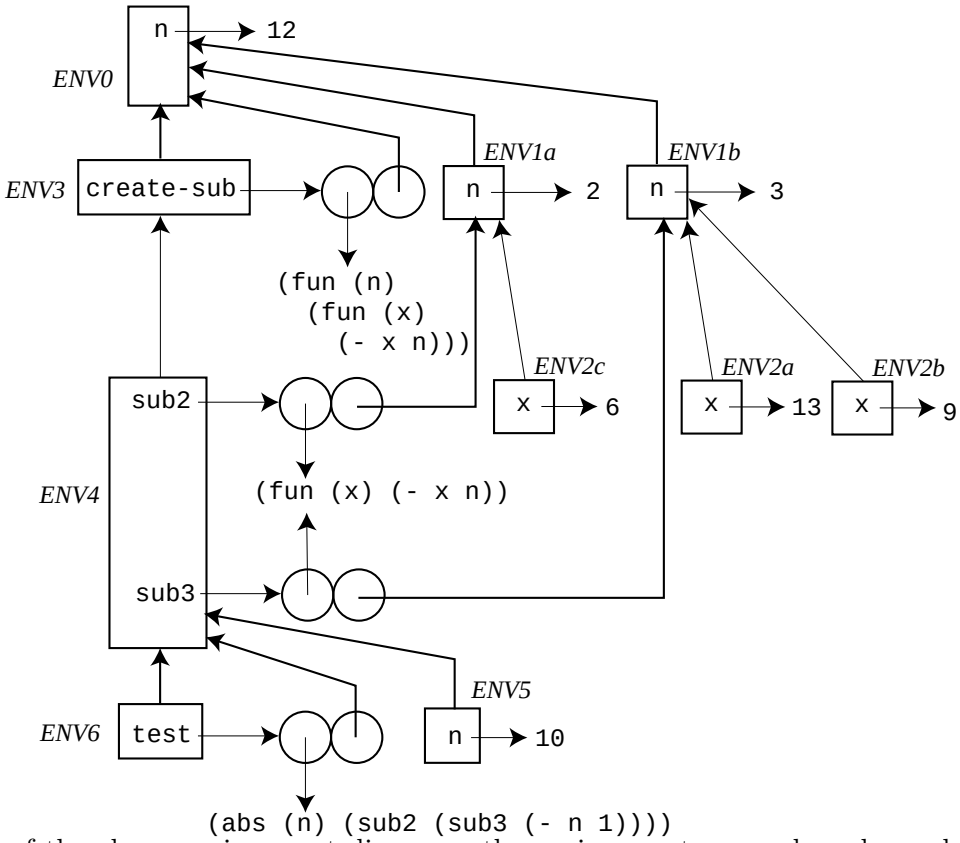


Finally the function application `(add-a (* 2 a))` is evaluated in ENV_3. First, the subexpressions `add-a` and `(* 2 a)` must be evaluated in ENV_3; these evaluations yield the add-a closure and 26, respectively. Next, the closure is applied to 26. This creates a new frame ENV_1 binding `x` to 26; by rule 2 from above, the parent of this frame is ENV_0, the environment of closure; the environment ENV_3 of the function application is simply not involved in this decision.



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.

As a second example of static scoping in the environment model, consider running the `create-sub` program from the previous section on the input 12. Below is an environment diagram showing all environments created during the evaluation of this program. You should study this diagram carefully and understand why the parent pointer of each environment frame is the way it is. The final answer of the program (which is not shown in the environment model itself) is 4.



In both of the above environment diagrams, the environment names have been chosen to underscore a critical fact that relates the environment diagrams to the contour diagrams. 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. For example, the environment chain $ENV_6 \rightarrow ENV_4 \rightarrow ENV_3 \rightarrow ENV_0$ models the contour nesting $C_6 \rightarrow C_4 \rightarrow C_3 \rightarrow C_0$, and the environment chains $ENV_{2\mathrm{c}} \rightarrow ENV_{1\mathrm{a}} \rightarrow ENV_0$, $ENV_{2\mathrm{a}} \rightarrow ENV_{1\mathrm{b}} \rightarrow ENV_0$, and $ENV_{2\mathrm{b}} \rightarrow ENV_{1\mathrm{b}} \rightarrow ENV_0$ model the contour nesting $C_2 \rightarrow C_1 \rightarrow C_0$.

These correspondences are not coincidental, but by design. Since static scoping is defined by the contour diagrams, the environment model must somehow encode the nesting of contours. The environment component of closures is the mechanism by which this correspondence is achieved. The environment component of a closure is guaranteed to point to an environment ENV_{birth} that models the contour enclosing the abstraction of the closure. When the closure is applied, the newly constructed frame extends ENV_{birth} with a new frame that introduces bindings for the parameters of the abstraction. These are exactly the bindings implied by the contour of the abstraction. Any expression in the body of the abstraction is then evaluated relative to the extended environment.

3.4 Interpreter Implementation of Environment Model

Rules 1 and 2 of the previous section are easy to implement in an environment model interpreter. The implementation is shown in Figure 11. 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.

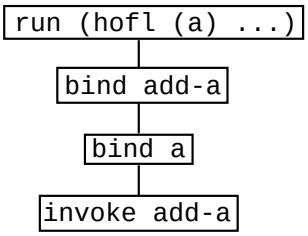
```
(* val eval : Hofl.exp -> valu Env.env -> valu *)
and eval exp env =
  match exp with
  ⋮
  | Abs(fml,body) -> Fun(fml,body,env) (* make a closure *)
  | App(rator,rand) -> apply (eval rator env) (eval rand env)
  ⋮

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

Figure 11: Essence of static scoping in HOFL.

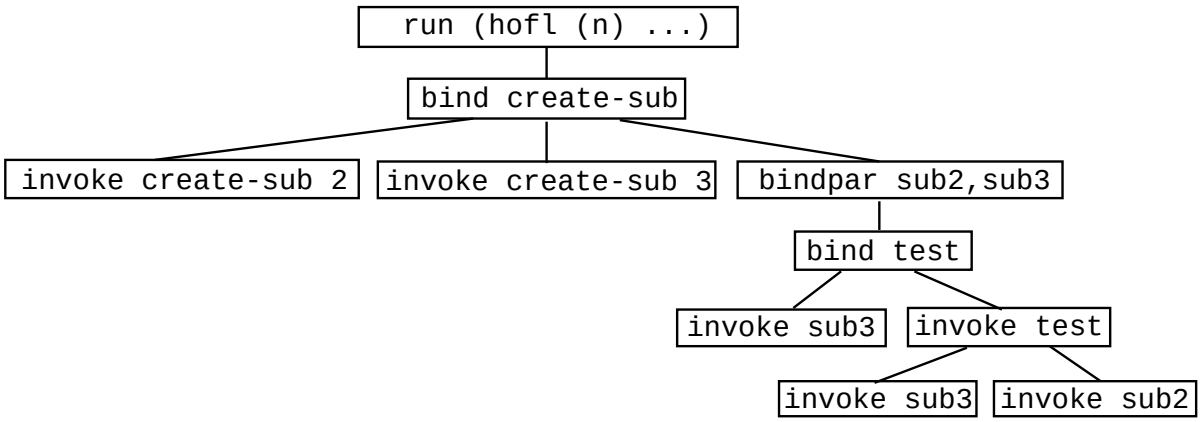
4 Dynamic Scoping

4.1 Environment Model

In dynamic scoping, environments follow the shape of the invocation tree for executing the program. Recall that an invocation tree has one node for every function invocation in the program, and that each node has as its children the nodes for function invocations made directly within in its body, ordered from left to right by the time of invocation (earlier invocations to the left). Since `bind` desugars into a function application, we will assume that the invocation tree contains nodes for `bind` expressions as well. We will also consider the execution of the top-level program to be a kind of function application, and its corresponding node will be the root of the invocation tree. For example, here is the invocation tree for the `add-a` program:



As a second example, here is the invocation tree for the `create-sub` program:



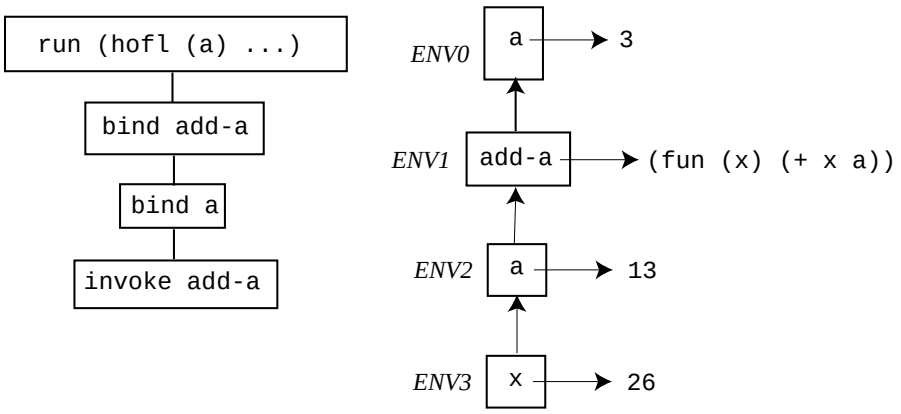
Note: in some cases (but not the above two), the shape of the invocation tree may depend on the values of the arguments at certain nodes, which in turn depends on the scoping mechanism. So the invocation tree cannot in general be drawn without fleshing out the details of the scoping mechanism.

The key rules for dynamic scoping are as follows:

1. Evaluating an abstraction *ABS* in an environment *ENV* just returns *ABS*. In dynamic scoping, there there is no need to pair the abstraction with its environment of creation.

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 in which the function application is being evaluated - that is, the environment of the invocation (call), not the environment of creation. This means that the free variables in the abstraction body will be looked up in the environment where the function is called.

Consider the environment model showing the execution of the `add-a` program on the argument 3 in a dynamically scoped version of Hofl. According to the above rules, the following environments are created:



The key differences from the statically scoped evaluation are (1) the name `add-a` is bound to an abstraction, not a closure and (2) the parent frame of ENV_3 is ENV_2, not ENV_0. This means that the evaluation of `(+ x a)` in ENV_3 will yield 39 under dynamic scoping, as compared to 29 under static scoping.

Figure 12 shows an environment diagram showing the environments created when the `create-sub` program is run on the input 12. The top of the figure also includes a copy of the invocation tree to emphasize that in dynamic scope the tree of environment frames has *exactly* the same shape as the invocation tree. You should study the environment diagram and justify the target of each parent pointer. Under dynamic scoping, the first invocation of `sub3` (on 13) yields 1 because the `n` used in the subtraction is the program parameter `n` (which is 12) rather than the 3 used as an argument to `create-sub` when creating `sub3`. The second invocation of `sub3` (on 0) yields -1 because the n found this time is the argument 1 to test. The invocation of `sub2` (on -1) finds that `n` is this same 1, and returns -2 as the final result of the program.

4.2 Interpreter Implementation

The two rules of the dynamic scoping mechanism are easy to encode in the environment model. The implementation is shown in Figure 11. For the first rules, the evaluation of an abstraction just returns the abstraction. For the second rules, the application of a function passes the call-time environment to funapply-dynamic, where it is used as the parent of the environment frame created for the application.

5 Recursive Bindings

5.1 The `bindrec` Construct

HOFL's `bindrec` construct allows creating mutually recursive structures. For example, here is the classic `even?/odd?` mutual recursion example expressed in HOFL:

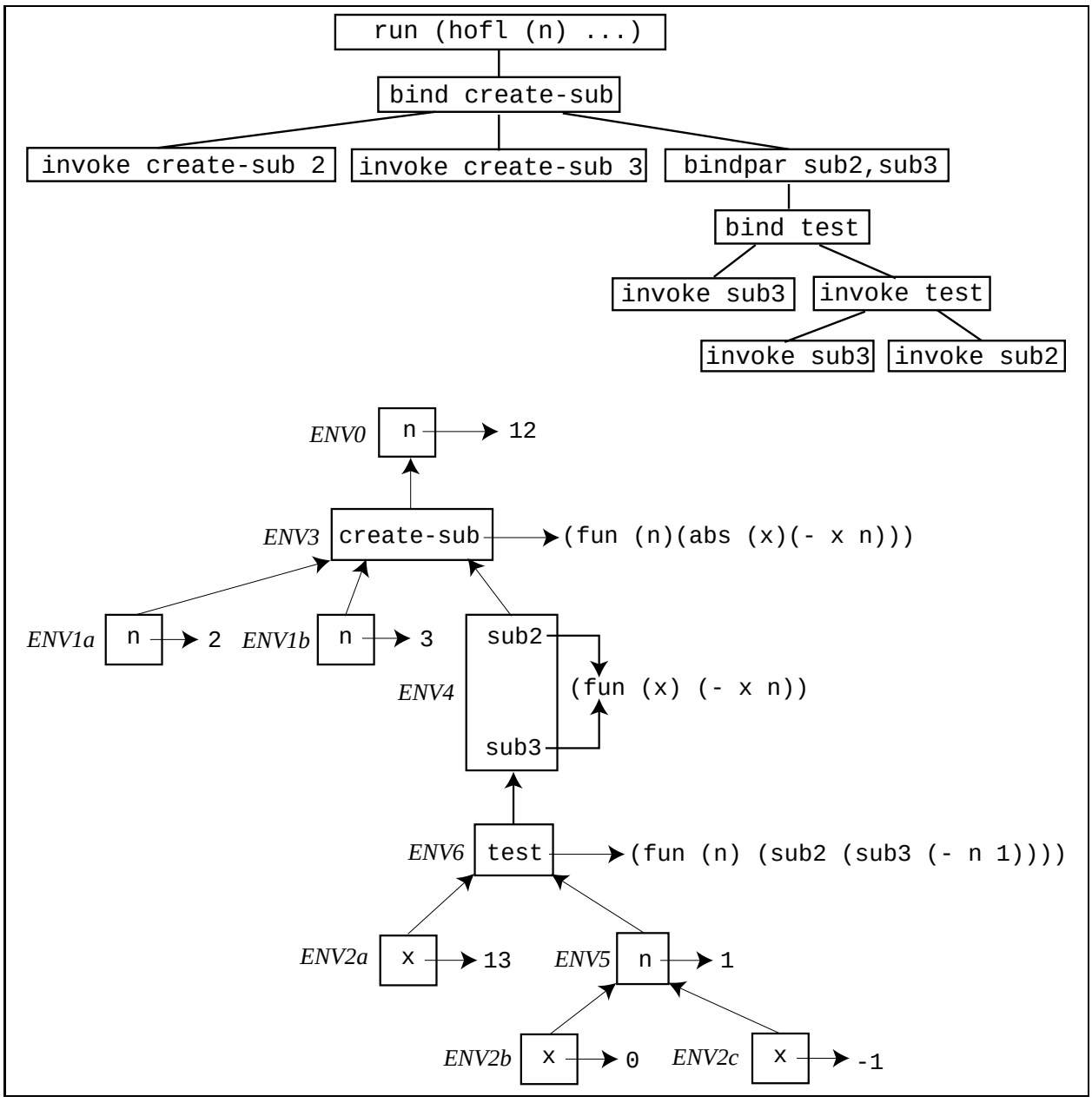


Figure 12: Invocation tree and environment diagram for the `create-sub` program run on 12.

```
(* 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) env
  ⋮

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

Figure 13: Essence of dynamic scoping in HOFL.

```
(hofl (n)
  (bindrec ((even? (abs (x)
                     (if (= x 0)
                         #t
                         (odd? (- x 1)))))
            (odd? (abs (y)
                    (if (= y 0)
                        #f
                        (even? (- y 1))))))
      (prep (even? n)
            (prep (odd? n)
                  #e))))
```

The scope of the names bound by `bindrec` (`even?` and `odd?` in this case) includes not only the body of the `bindrec` expression, but also the definition expressions bound to the names. This distinguishes `bindrec` from `bindpar`, where the scope of the names would include the body, but not the definitions. The difference between the scoping of `bindrec` and `bindpar` can be seen in the two contour diagrams in Fig. **??**. In the `bindrec` expresion, the reference occurrence of `odd?` within the `even?` abstraction has the binding name `odd?` as its binding occurrence; the case is similar for `even?`. However, when `bindrec` is changed to `bindpar` in this program, the names `odd?` and `even?` within the definitions become unbound variables. If `bindrec` were changed to `bindseq`, the occurrence of `even?` in the second binding would reference the declaration of `even?` in the first, but the occurrence of `odd?` in the first binding would still be unbound.

5.2 Environment Model Evaluation of `bindrec`

5.2.1 High-level Model

How is `bindrec` handled in the environment model? We do it in three stages:

1. Create an empty environment frame that will contain the recursive bindings, and set its parent pointer to be the environment in which the bindrec expression is evaluated.

2. Evaluate each of the definition expressions with respect to the empty environment. If evaluating any of the definition expressions requires the value of one of the recursively bound

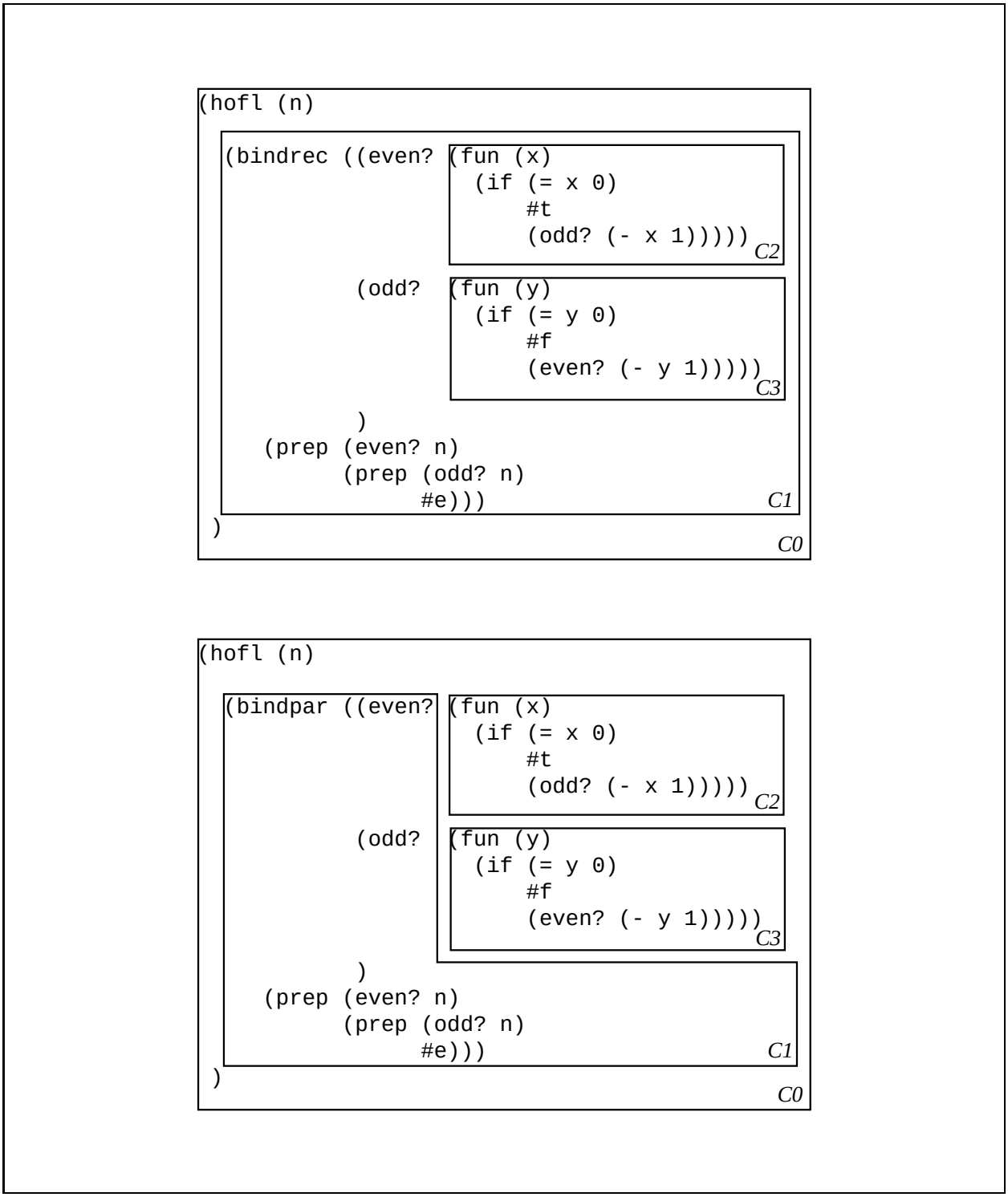
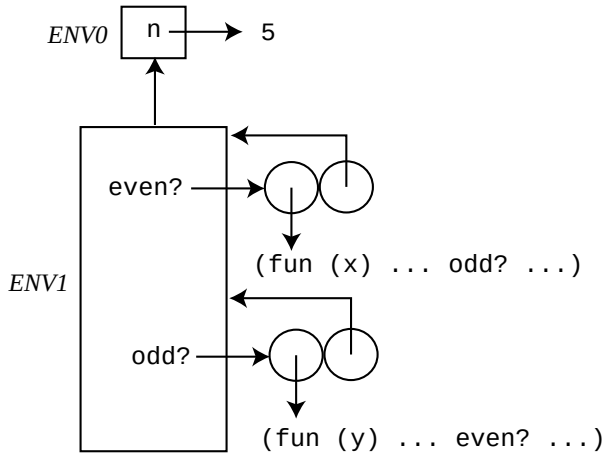


Figure 14: Lexical contours for versions of the `even?`/`odd?` program using `bindrec` and `bindpar`.

variables, the evaluation process is said to encounter a **black hole** and the `bindrec` is considered ill-defined.

3. Populate the new frame with bindings between the binding names and the values computed in step 2. Adding the bindings effectively "ties the knot" of recursion by making cycles in the graph structure of the environment diagram.

The result of this process for the `even?`/`odd?` example is shown below, where it is assumed that the program is called on the argument 5. The body of the program would be evaluated in environment ENV_1 constructed by the `bindrec` expression. Since the environment frames for containing `x` and `y` would all have ENV_1 as their parent pointer, the references to odd? and even? in these environments would be well-defined.



In order for `bindrec` to be meaningful, the definition expressions cannot require immediate evaluation of the `bindrec`-bound variables (else a black hole would be encountered). For example, the following `bindrec` example clearly doesn't work because in the process of determining the value of `x`, the value `x` is required before it has been determined:

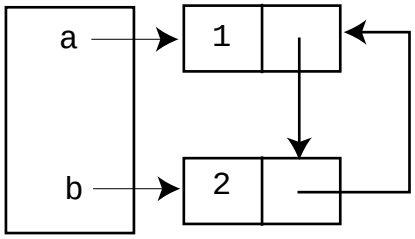
```
(bindrec ((x (+ x 1)))
  (* x 2))
```

In contrast, in the `even?`/`odd?` example we are not asking for the values of `even?` and `odd?` in the process of evaluating the definitions. Rather the definitions are abstractions that will refer to `even?` and `odd?` at a later time, when they are invoked. Abstractions serve as a sort of delaying mechanism that make the recursive bindings sensible.

As a more subtle example of a meaningless `bindrec`, consider the following:

```
(bindrec ((a (prep 1 b))
          (b (prep 2 a)))
  b)
```

Unlike the above case, here we can imagine that the definition might mean something sensible. Indeed in so-called call-by-need (a.k.a lazy) languages (such as Haskell), the above definitions are very sensible, and stand for the following list structure:



However, call-by-value (a.k.a. strict or eager) languages (such as HOFL, OCAML, SCHEME, JAVA, C, etc.) require that all definitions be completely evaluated to values before they can be bound to a name or inserted in a data structure. In this class of languages, the attempt to evaluate `(preps 1 b)` fails because the value of `b` cannot be determined.

Nevertheless, by using the delaying power of abstractions, we can get something close to the above cyclic structure in HOFL. In the following program, the references to the recursive bindings `one-two` and `two-one` are "protected" within abstractions of zero variables (which are known as thunks). Any attempt to use the delayed variables requires applying the thunks to zero arguments (as in the expression `((snd stream))` within the prefix function).

```
(hofl (n)
  (bindpar ((pair (fun (a b) (list a b)))
            (fst (fun (pair) (head pair)))
            (snd (fun (pair) (head (tail pair)))))
    (bindrec ((one-two (pair 1 (fun () two-one)))
              (two-one (pair 2 (fun () one-two)))
              (prefix (fun (num stream)
                        (if (= num 0)
                            (empty)
                            (prep (fst stream)
                                  (prefix (- num 1)
                                          ((snd stream))))))))
      (prefix n one-two))))
```

When the above program is applied to the input 5, the result is `(list 1 2 1 2 1)`.

5.2.2 Implementing `bindrec`

Implementing the "knot-tying" aspect of the recursive bindings of bindrec within the `eval` function of the statically scoped HOFL interpreter proves to be rather tricky. We will consider a sequence of incorrect definitions for the `bindrec` clause on the path to developing some correct ones.

Here is a first attempt:

```
(* Broken Attempt 1 *)
| Bindrec(names,defns,body) ->
  eval body
       (Env.bindAll names
                    (map (fun defn -> eval defn ???)
                         defns)
                    env)
```

There is a problem here: what should the environment *???* be? It shouldn't be `env` but the new environment that results from extending `env` with the recursive bindings. But the new environment has no name in the above clause.

A second attempt uses OCAML's `let` to name the result of `Env.bindAll`:

```
(* Broken Attempt 2 *)
| Bindrec(names,defns,body) ->
  eval body
       let newEnv = Env.bindAll names
                                (map (fun defn -> eval defn newEnv)
                                     defns)
                                env
       in newEnv
```

This attempt fails because, by the scoping rules of `let`, `newEnv` is an unbound variable in `Env.bindAll ...` .

A third attempt replaces `let` with `let rec`:

```
(* Broken Attempt 3 *)
| Bindrec(names,defns,body) ->
  eval body
       let rec newEnv =
         Env.bindAll names
                     (map (fun defn -> eval defn newEnv)
                          defns)
                     env
       in newEnv
```

The above clause attempts to use the knot-tying ability of OCAML's own recursive binding construct, `let rec`, to implement HOFL's recursive binding construct. Now the `newEnv` within `Env.bindAll ...` is indeed correctly scoped. Unfortunately, there are still two problems:

1. The OCAML `let rec` construct can only be used to define recursive functions. It cannot be used to define more general recursive values (such as the `one-two` example above).

2. Even if OCAML *did* allow more general recursive values to be defined via `let rec`, because OCAML is a call-by-value language, we would still come face to face with the same sort of problem encountered in the recursive list example from above. That is, the `eval` within the functional argument to `map` requires that all its arguments be evaluated to values before it is invoked. But its `newEnv` argument is defined to be the result of a computation that depends on the result returned by invocations of this occurrence of `eval`. This leads to an irresolvable set of constraints: `eval` must return before it can be invoked!

We can fix the problem in the same way we fixed the recursive list problem: by using thunks to delay evaluation of the recursive bound variable. In particular, rather than storing the result of evaluating the definition in the environment, we can store in the environment a thunk for evaluating the definition:

```
(* Complex (but working) Attempt 4 *)
| Bindrec(names,defns,body) ->
  eval body
       let rec newEnv =
         Env.bindAll names
                     (map (fun defn -> (fun () -> eval defn newEnv))
                          defns)
                     env
       in newEnv
```

Once we do this, we must ensure (1) that *all* entities stored in the environments used by `eval` are thunks and (2) that whenever a thunk is looked up in the environment, it should be "dethunked"

- i.e., applied to zero arguments to retrieve its value. This makes sense if you think in terms of types. Point (1) says that the type of environments is effectively changed from `var -> valu` to `var -> unit -> value`, where `unit` is the type of `()`. Point (2) says that since the result of an environment lookup is now a function of type `unit -> valu`, it must be applied to zero arguments in order to get a value.

Although the above approach "works", it is less elegant and less efficient than we'd like it to be. A solution that is both more elegant and more efficient is possible if we adopt the environment signature and implementation in Fig. **??**. This supports operations for binding both regular values (`bind` and `bindAll`) as well as thunked values (`bindThunk` and `bindAllThunks`) in such a way that the regular values may be looked up efficiently without any dethunking. Furthermore, it supports a abstract fixed point operator, `fix`, that internally uses OCAML's `let rec` construct to find the fixed point. This is possible because the implementation uses functions to represent environments, and `let rec` can be used to define recursive functions.

Using the extended environment module, we can implement `bindrec` as follows:

```
(* Final Version *)
| Bindrec(names,defns,body) ->
    eval body
         (Env.fix (fun e ->
                   (Env.bindAllThunks names
                                      (map (fun defn ->
                                             (fun () -> eval defn e))
                                           defns)
                                      env)))
```

5.3 Fixed Points and the Y Operator

With all the complexity surrounding the implementation of `bindrec` in HOFL, it may be surprising that it is possible to define recursive functions in HOFL without `bindrec`! While `bindrec` is convenient for defining such functions, it is not necessary.

```
module type ENV = sig
  type 'a env
  val empty: 'a env
  val bind : string -> 'a -> 'a env -> 'a env
  val bindAll : string list -> 'a list -> 'a env -> 'a env
  val make : string list -> 'a list -> 'a env
  val lookup : string -> 'a env -> 'a option
  val bindThunk : string -> (unit -> 'a) -> 'a env -> 'a env
  val bindAllThunks : string list -> (unit -> 'a) list -> 'a env -> 'a env
  val merge : 'a env -> 'a env -> 'a env
  val fix : ('a env -> 'a env) -> 'a env
  (* for testing *)
  val fromFun : (string -> 'a option) -> 'a env
  val toFun : 'a env -> (string -> 'a option)
end

module Env : ENV = struct

  type 'a env = string -> 'a option

  let empty = fun s -> None

  let bind name valu env =
    fun s -> if s = name then Some valu else env s

  let bindAll names vals env = ListUtils.foldr2 bind env names vals

  let make names vals = bindAll names vals empty

  let lookup name env = env name

  let bindThunk name valuThunk env =
    fun s -> if s = name then Some (valuThunk ()) else env s

  let bindAllThunks names valThunks env =
    ListUtils.foldr2 bindThunk env names valThunks

  let merge env1 env2 =
    fun s -> (match env1 s with
                None -> env2 s
              | some -> some)

  let fix gen = let rec envfix s = (gen envfix) s in envfix

  let fromFun f = f

  let toFun f = f

end
```

Figure 15: An environment signature and an function-based implementation of this signature.

```
(def y (fun (g)
         ((fun (s) (fun (x) ((g (s s)) x)))
          (fun (s) (fun (x) ((g (s s)) x))))))

(def fact-gen (fun (f)
                      (fun (n)
                    (if (= n 0)
                      1
                      (* n (f (- n 1)))))))

(def fact (y fact-gen))

(def church-pair (fun (x y) (fun (f) (f x y))))

(def church-fst (fun (p) (p (fun (x y) x))))

(def church-snd (fun (p) (p (fun (x y) y))))

(def even-odd-gen (fun (p)
                    (church-pair
                       (fun (x) ; even?
                        (if (= x 0)
                            #t
                            ((church-snd p) (- x 1))))
                      (fun (y) ; odd?
                        (if (= y 0)
                            #f
                            ((church-fst p) (- y 1)))))))

(def even? (church-fst (y even-odd-gen)))

(def odd? (church-snd (y even-odd-gen)))
```