CS251 Programming Languages **Handout # 31**
Randy Shull
Wellesley College

An Introduction to HOFL, a Higher-order Functional Language

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!

In this handout, we introduce the key features of the HOFL language in the context of examples. We will study the implementation of HOFL, particularly with regard to scoping issues, in a separate handout.

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.

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

P ∈ *Program*	
$P \rightarrow$ `(hofl (`I_{formal_1} ... I_{formal_n}`)` E_{body}`)`	Kernel Program
$P \rightarrow$ `(hofl (`I_{formal_1} ... I_{formal_n}`)` E_{body} $D_1 \ldots D_k$`)`	Sugared Program
D ∈ *Definition*	
$D \rightarrow$ `(def` I_{name} E_{body}`)`	Basic Definition
$D \rightarrow$ `(def (`$I_{fcnName}$ I_{formal_1} ... I_{formal_n}`)` E_{body}`)`	Sugared Function Definition
$D \rightarrow$ `(load filename)`	File Load
E ∈ *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 ... I_n`)` E_{body}`)`	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 ∈ *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 ∈ *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 ∈ *Primitive Operator*: e.g., `+`, `<=`, `and`, `not`, `prep`
F ∈ *Function Name*: e.g., `f`, `sqr`, `+-and-*`
I ∈ *Identifier*: e.g., `a`, `captain`, `fib_n-2`
N ∈ *Integer*: e.g., `3`, `-17`
B ∈ *Boolean*: `#t` and `#f`
C ∈ *Character*: `'a'`, `'B'`, `'7'`, `'\n'`, `''''\\'`
R ∈ *String*: `"foo"`, `"Hello there!"`, `"The string \"bar\""`

Figure 1: Grammar for the HOFL langauge.

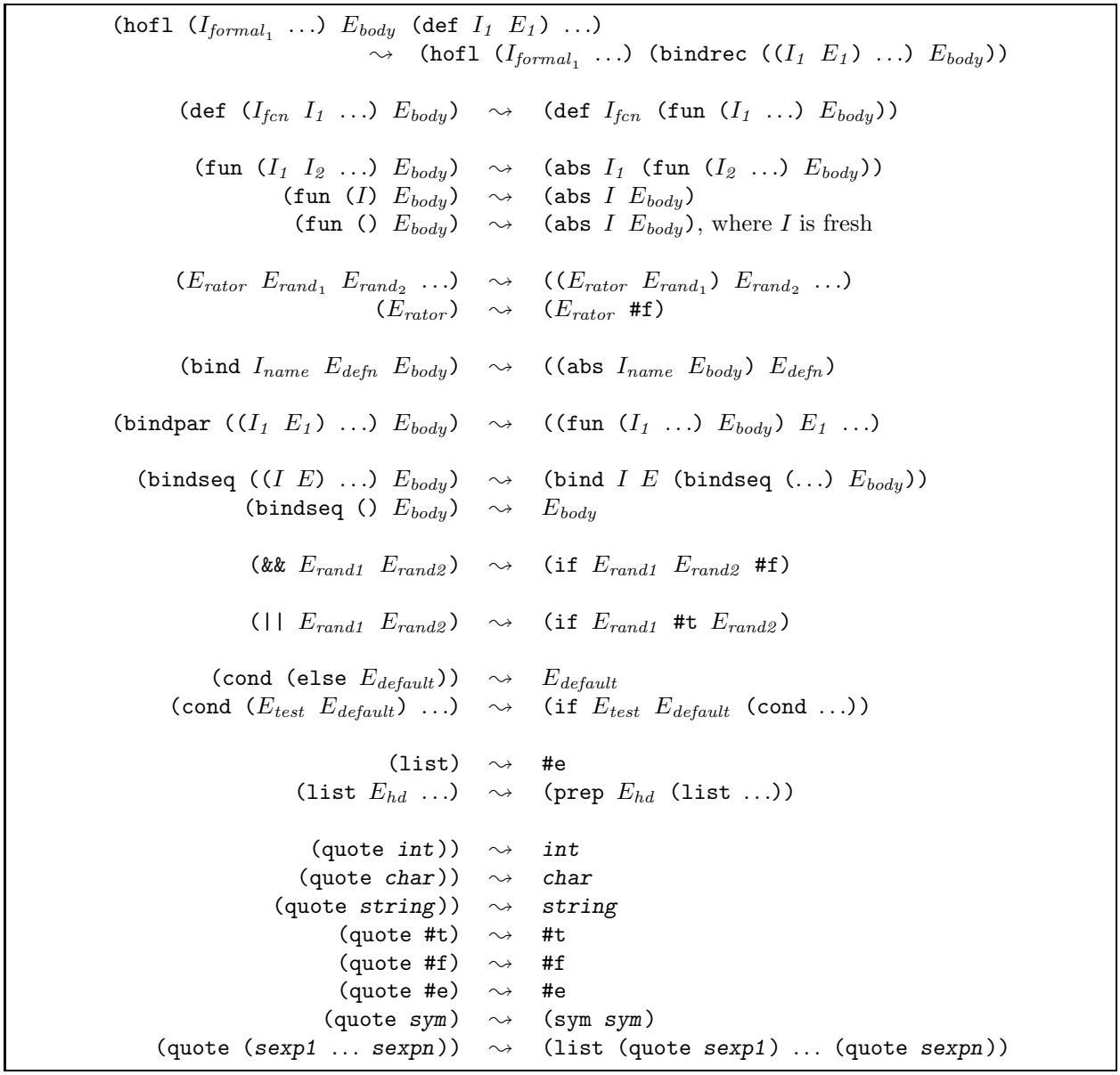

Figure 2: Desugaring rules for HOFL.

```
((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. For example,

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

is sugar for

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

Unlike in VALEX, in HOFL the `bindpar` desugaring need not be handled specially in the parser because it can be expressed via a high-level desugaring rule (also involving the application of a manifest abstraction). For example:

```
(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))
```

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

The scope of the names bound by `bindrec` (`even?` and `odd?` in the latter 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. 3. 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.

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

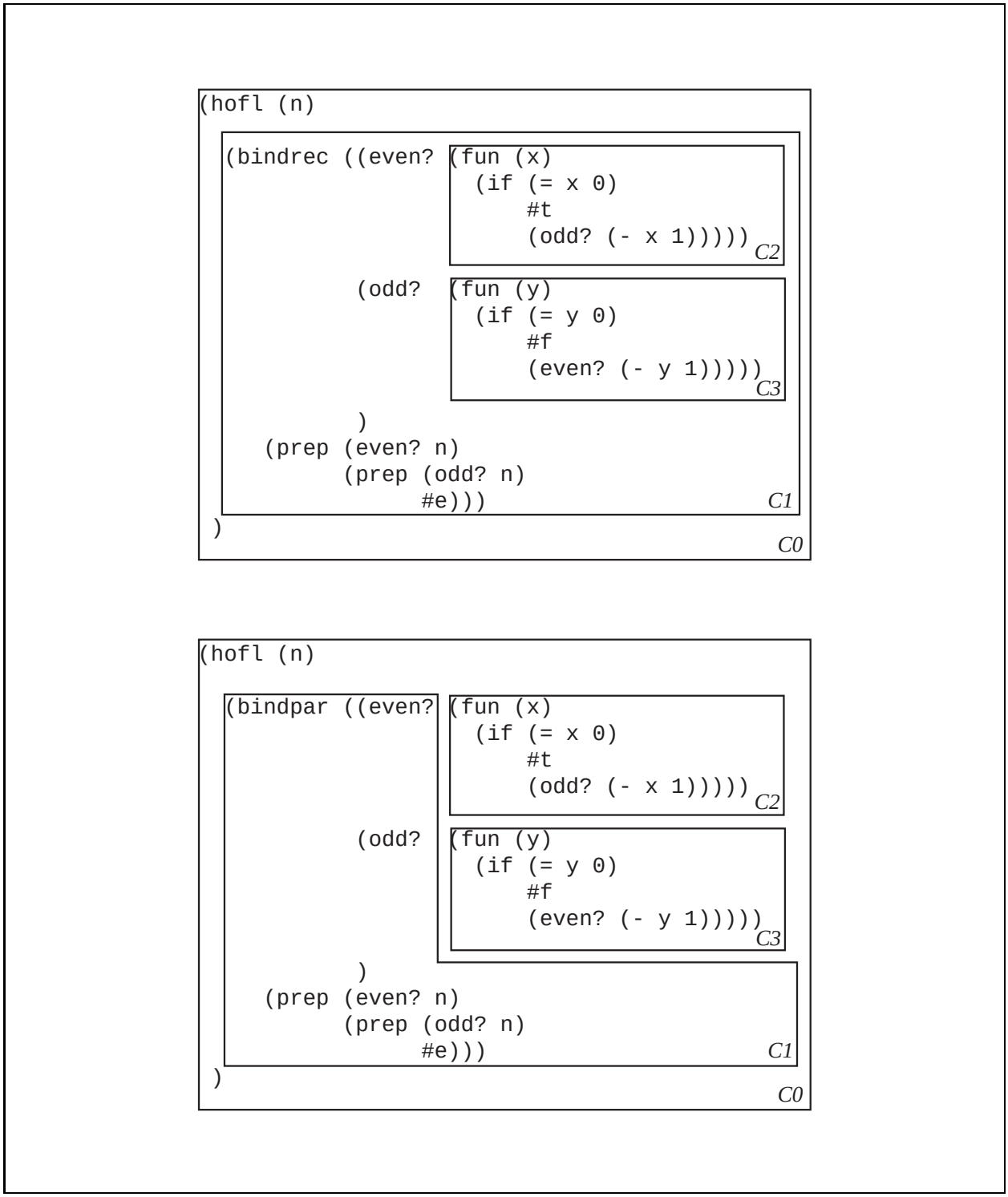

Figure 3: Lexical contours for versions of the `even?`/`odd?` program using `bindrec` and `bindpar`.

```
(hofl (a b)
  (bindrec
      (
       (map (fun (f xs)
              (if (empty? xs)
                  #e
                  (prep (f (head xs))
                        (map f (tail xs))))))

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

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

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

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

       (sq (fun (i) (* i i)))

       (even? (fun (n) (= (% n 2) 0)))
       )
    (foldr (fun (x y) (+ x y))
           0
           (map sq (filter even? (range a b))))))
```

Figure 4: A sample HOFL program with higher-order list-processing functions.

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

Because HOFL supports recursion and higher-order functions, we can use it to define all the higher-order functions and list-processing functions that we've investigated in OCAML. For example, what does the HOFL program in Fig. 4 do?

4 Definitions in HOFL Programs

To simplify the definition of values, especially functions, in HOFL programs and in the interactive HOFL interpreter, 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, which results from collecting the original expression and two definitions into a `bindrec` and desugaring:

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

Let's define one more function and invoke it:

```
hofl> (def (sum-squares-between lo hi)
        (if (> lo hi)
            0
            (+ (sq lo) (sum-squares-between (+ lo 1) hi))))
sum-squares-between
hofl> (sum-squares-between three 5)
50
```

The value *50* is the result of evaluating the following expression, which results from collecting the original expression and three definitions into a `bindrec` and desugaring:

```
(bindrec ((three (+ 1 2))
          (sq (abs x (* x x)))
          (sum-squares-between
            (abs lo
              (abs hi
                (if (> lo hi)
                    0
                    (+ (sq lo) ((sum-squares-between (+ lo 1)) hi)))))))
  ((sum-squares-between three) 5))
```

It isn't necessary to define `sq` before `sum-square-between`. The definitions can appear in any order, as long as no attempt is made to find the value of a defined name before it is defined.

5 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. 5 and `"list-utils.hfl"` contains the definitions in Fig. 6. Then we can have the following REPL interactions:

```
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
```

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

```
hofl> (load "list-utils.hfl")
length
rev
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)))
(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. 7 loads the 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`.

HOFL is even powerful enough for writing interpreters. For example, we can write an INTEX, BINDEX, and VALEX interpreters in HOFL. We can even (gasp!) we write a HOFL interpreter in HOFL. Such an interpreter is known as a **metacircular interpreter**.

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

(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 6: 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 7: The contents of the file `"env.hfl"`, which contains a functional implementation of environments expressed in HOFL.

6 A Substitution-Model Interpreter For HOFL

Now we study a substitution model interpreter for HOFL. This interpreter merely mechanizes the substitution-model reasoning we've been using for OCAML all semester.

Fig. 8 presents the OCAML datatypes for the HOFL abstract syntax. Note that `Abs` and `Fun` are essential the same thing, except that the first is an `exp` while the second is a `valu`.

Fig. 9 presents the substitution-model interpreter for HOFL.

Here are the highlights of the interpreter:

- An abstraction is evaluated to a function value simply by changing the `Abs` tag to a `Fun` tag.
- An application is evaluated first by recursively evaluating both the rator and rand, and then applying the function that is the value of the rator to the argument that is the value of the rand. The application process, performed by the `apply` helper function, substitutes the argument for the formal parameter in the body, and then evaluates the result. Sound familiar?
- In a `bindrec`, each recursive definition is replaced by a copy of the definition in which each reference to a `bindrec`-bound name is replaced by an expression that wraps the name in a new `bindrec` with the same bindings. This has the effect of propagating the recursive nature of the `bindrec` to each reference to a `bindrec`-bound name.

To see how `bindrec` works in practice, consider the following version of the `even?`/`odd?` program from above that has a simpler body:

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

Suppose we introduce the abbreviation E for the abstraction

```
type var = string

type pgm = Pgm of var list * exp (* param names, body *)

and exp =
    Lit of valu (* value 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

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

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

let primopFunction (Primop(_,fcn)) = fcn
```

Figure 8: HOFL abstract syntax expressed with OCAML dataypes.

```
(abs x
  (if (= x 0)
      #t
      (odd? (- x 1))))
```

and the abbrevation O for the abstraction

```
(abs y
  (if (= y 0)
      #f
      (even? (- y 1))))
```

Then Fig. 10 shows the substitution model evaluation of the program on the input 3. Note how the substitution that wraps each `bindrec`-bound name in a fresh `bindrec` allows the abstractions for the recursive functions to be unwound one level at a time, giving rise to the desired behavior for the recursive functions.

```
(* 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) (* Just change from exp to valu *)
  | App(rator,rand) -> apply (eval rator) (eval rand)
  | Bindrec(names,defns,body) ->
      eval (substAll (map (fun defn -> Bindrec(names,defns,defn))
                          defns)
                     names
                     body)

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: HOFL abstract syntax expressed with OCAML dataypes.

Abbreviations

E = `(abs x (if (= x 0) #t (odd? (- x 1))))`
O = `(abs y (if (= y 0) #f (even? (- y 1))))`

Example

`(hofl (n) (bindrec ((even? E) (odd? O)) (even? n)))` run on [3]

```
; Here and below, assume a ''smart'' substitution that
; performs renaming only when variable capture is possible.

⇒ (bindrec ((even? E) (odd? O)) (even? 3))
⇒ (bindrec ((even? E) (odd? O)) ((abs x (if (= x 0) #t (odd? (- x 1)))) 3))
⇒ ((abs x (if (= x 0) #t ((bindrec ((even? E) (odd? O)) O) (- x 1)))) 3)
⇒ (if (= 3 0) #t ((bindrec ((even? E) (odd? O)) O) (- 3 1)))
⇒ (if #f #t ((bindrec ((even? E) (odd? O)) O) (- 3 1)))

⇒ ((bindrec ((even? E) (odd? O)) O) (- 3 1))
⇒ ((abs y (if (= y 0) #f ((bindrec ((even? E) (odd? O)) E) (- y 1)))) (- 3 1))
⇒ ((abs y (if (= y 0) #f ((bindrec ((even? E) (odd? O)) E) (- y 1)))) 2)
⇒ (if (= 2 0) #f ((bindrec ((even? E) (odd? O)) E) (- 2 1)))
⇒ (if #f #f ((bindrec ((even? E) (odd? O)) E) (- 2 1)))

⇒ ((bindrec ((even? E) (odd? O)) E) (- 2 1))
⇒ ((abs x (if (= x 0) #t ((bindrec ((even? E) (odd? O)) O) (- x 1)))) (- 2 1))
⇒ ((abs x (if (= x 0) #t ((bindrec ((even? E) (odd? O)) O) (- x 1)))) 1)
⇒ (if (= 1 0) #t ((bindrec ((even? E) (odd? O)) O) (- 1 1)))
⇒ (if #f #t ((bindrec ((even? E) (odd? O)) O) (- 1 1)))

⇒ ((bindrec ((even? E) (odd? O)) O) (- 1 1))
⇒ ((abs y (if (= y 0) #f ((bindrec ((even? E) (odd? O)) E) (- y 1)))) (- 1 1))
⇒ ((abs y (if (= y 0) #f ((bindrec ((even? E) (odd? O)) E) (- y 1)))) 0)
⇒ (if (= 0 0) #f ((bindrec ((even? E) (odd? O)) E) (- 0 1)))
⇒ (if #t #f ((bindrec ((even? E) (odd? O)) E) (- 0 1)))
⇒ #f
```

Figure 10: Example evaluation involving `bindrec` in the substitution-model interpreter.