FOFL and FOBS: First-Order Functions

Revisions: Apr 24: In the interpreters for FOFL and FOBS in figures 2 and 3, the apply functions were missing checks that the number of formals equal the number of actuals.

We have seen the power of first-class higher-order functions in OCAML, and have seen how they can be implemented in the mini-language Hofl. The functions in most real programming languages (including C, PASCAL, and FORTRAN) are much more limited than those in OCAML and Hofl. In this handout, we explore two mini-languages with more limited kinds of functions:

1. FOFL (First-Order Functional Language) extends VALEX with first-order second-class global functions. Functions in FOFL are similar to those in C\(^1\).

2. FOBS (First-Order Block-Structure Language) extends FOFL with block structure – the ability to declare functions inside of other functions. Functions in FOBS are similar to those in PASCAL\(^2\).

1 C supports a limited form of first-class functions in the form of function pointers; this important feature is not modeled in FOFL.

2 PASCAL allows functions to be passed as arguments but not returned as results; this important feature is not modeled in FOBS.
Figure 1: Grammar for the FOFL language.
The globally declared functions in a FOFL program are mutually recursive:

```lisp
(fofl (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))))
)
```

Here is a simple example of list processing in FOFL:

```lisp
(fofl (a b) (sum (map-square (filter-even (range a b))))
  (def (sum ns)
    (if (empty? ns)
      0
      (+ (head ns)
        (sum (tail ns)))))
  (def (map-square ns)
    (if (empty? ns)
      #e
      (prep (* (head ns) (head ns))
        (map-square (tail ns)))))
  (def (filter-even ns)
    (if (empty? ns)
      #e
      (if (even? (head ns))
        (prep (head ns) (filter-even (tail ns)))
        (filter-even (tail ns)))))
  (def (range lo hi)
    (if (> lo hi)
      #e
      (prep lo (range (+ lo 1)) hi)))
  (def (even? n) (= 0 (% n 2))))
)
```

Functions in FOFL are second-class. Although they can be named by function declarations, they cannot be passed as arguments to other functions, returned as values from functions, stored in lists, or created anywhere other than in a top-level function declaration.
1.2 Namespaces

A programming language may have several different categories of names. Each such category is called a namespace. For example, JAVA has distinct namespaces for packages, classes, methods, instance variables, class variables, and method parameters/local variables.

In a language with multiple namespaces, the same name can simultaneously be used in different namespaces without any kind of naming conflict. For example, consider the following JAVA class declaration:

```java
public class Circle {
    // Instance variable of a Circle object.
    public double radius;

    // Constructor method for creating Circle objects.
    public Circle (double r) {
        this.radius = r;
    }

    // Instance method for scaling Circles.
    public Circle scale (double factor) {
        return new Circle(factor * this.radius);
    }
}
```

It turns out that we can rename every one of the names appearing in the above program to radius (as shown below) and the class will have the same meaning!

```java
public class radius {
    // Instance variable of a circle object.
    public double radius;

    // Constructor method for creating Circle objects.
    public radius (double radius) {
        this.radius = radius;
    }

    // Instance method for scaling Circles.
    public radius radius (double radius) {
        return new radius(radius * this.radius);
    }
}
```

Of course, in order to use the renamed class, we would need to change uses of the original class consistently. For instance, the expression (new Circle(10)).scale(2).radius would have to be renamed to (new radius(10)).radius(2).radius.

Although using the name radius to stand for entities in four different namespaces (class, instance variable, instance variable name, parameter name) would make the program very difficult for a human program to read, the JAVA compiler and JAVA bytecode interpreter treat the renamed program identically to the original.

JAVA has an unusually high number of namespaces. But many languages have at least two namespaces: one for functions, and one for variables. For instance, in this category are PASCAL and COMMON LISP, as well as the mini-languages FOFL and FOBS that we are studying. In
contrast, many functional languages, such as Scheme, ML, and Haskell (as well as the mini-language HoFL) have a single namespace for functions and variables. This is parsimonious with the first-classness of functions, which allows functions to be named like any other values.

As a somewhat silly example, consider the following working definition of a factorial function in FOFL:

```scheme
(def (fact fact)
  (if (= fact 0)
   1
   (* fact (fact (- fact 1)))))
```

In this example, there are two distinct entities named `fact`: the factorial function (in the function namespace) and the formal parameter of the factorial function (in the variable namespace). Because the namespaces are distinct, there is no confusion between the entities. If the same experiment were tried in HoFL, OCAML, Scheme, or C, however, the function would encounter an error when applied to a number because all occurrences of `fact` in the body — including the one in the operator position — would refer to a number.

### 1.3 Evaluation and Scope in FOFL

A complete environment model evaluator for FOFL is presented in Fig. 2. Highlights of the evaluator include:

- There are three environments:
  1. The **function environment** `fenv` represents the function namespace. Function names are looked up here in the `apply` function.
  2. The **value environment** `venv` represents the value namespace that binds program parameters, function parameters, and `bind` names. Variable names are looked up here in the `Var` clause of `eval`.
  3. The **global environment** `genv` is that portion of the value environment that binds only the program parameters.

- Just as in HoFL, FOFL has scoping mechanisms that determine the meaning of a free variable in a function. `env-run` is parameterized over a `scope` parameter that determines the parent environment of the application frame created by a function application. The parent environment is determined from `genv` and `venv`. This supports the two traditional forms of scoping:
  - **static**: The parent environment is `genv`.
  - **dynamic**: The parent environment is `venv`.

It also allows for some non-traditional forms of scoping:

- **empty**: The parent environment is the empty environment.
- **merged**: The parent environment is the result of merging `genv` and `venv`.

Note that another option is to make the parent environment the result of merging `venv` and `genv`; but this is equivalent to dynamic scope.
(* Model a Fofl scoping mechanism as a way of combining global and local environments *)

```ocaml
type scoping = valu Env.env (* global parameter environment *)
  * valu Env.env (* local parameter environment *)
  -> valu Env.env
```

(* val run : scoping -> Fofl.pgm -> int list -> valu *)

(* This function is an example of block structure *)

```ocaml
let rec run scope (Pgm(fmls, body, fcns)) ints =
```

```ocaml
  let flen = length fmls
  and ilen = length ints
  in if flen <> ilen then
    raise (EvalError ("Program expected " ^ (string_of_int flen)
                           " arguments but got " ^ (string_of_int ilen)))
  else
    let genv = Env.make fmls (map (fun i -> Int i) ints) (* global param env *)
    and fenv = Env.make (map fcnName fcns) fcns (* function env *)
    in let rec eval exp venv (* current variable env *) =
```

```ocaml
    match exp with
    | Lit v -> v
    | Var name ->
        (match Env.lookup name venv with
          Some(i) -> i
        | None -> raise (EvalError("Unbound variable: " ^ name)))
    | PrimApp(op, rands) -> (primopFunction op) (map (flip eval venv) rands)
    | If(tst, thn, els) ->
        (match eval tst venv with
          Bool true -> eval thn venv
        | Bool false -> eval els venv
        | v -> raise (EvalError ("Non-boolean test value " ^ (valuToString v)
                               ^ " in if expression")))
    | Bind(name, defn, body) -> eval body (Env.bind name (eval defn venv) venv)
    | App(fname, rands) -> apply fname (map (flip eval venv) rands) venv
```

```ocaml
    and apply fname actuals venv =
      match Env.lookup fname fenv with
      | None -> raise (EvalError("unknown function " ^ fname))
      | Some (Fcn(name,formals,body)) ->
          let flen = length formals and alen = length actuals
          in if flen <> alen then
              raise (EvalError ("Function " ^ name ^ " expected "
                                ^ (string_of_int flen) ^ " arguments but got "
                                ^ (string_of_int alen)))
            else eval body (Env.bindAll formals actuals (scope genv venv))
        in eval body genv
```

(* Scoping mechanisms *)

```ocaml
let static = fun genv venv -> genv
let dynamic = fun genv venv -> venv
(* "weird" scopes *)
let empty = fun genv venv -> Env.empty
let merged = fun genv venv -> Env.merge genv venv
(* Note that Env.merge genv venv is equivalent to dynamic scope *)
```

Figure 2: An environment model evaluator for FOFL parameterized over the scoping mechanism (scope) for free value variables in a function.
As examples of FOFL scoping, consider running the following two programs on the argument list [3] under the four different scoping mechanisms:

; Program 1
(fofl y) (try 100)
  (def (add-y z) (* x y))
  (def (try y) (add-y (* 2 y)))
)

; Program 2
(fofl a) (test (* 100 a) (* 10 a))
  (def (linear x) (+ (* a x) b))
  (def (test a b) (linear 2))
)

<table>
<thead>
<tr>
<th>Scope</th>
<th>Value in Program 1</th>
<th>Value in Program 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dynamic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>merged</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2 FOBS

FOBS extends FOFL with block structure – the ability to locally declare within functions any kind of declaration that can be made at top-level. In particular, since FOFL supports top-level function declarations, FOBS allows functions to be declared within functions. Here we first explore block structure in the context of HOFL, and then discuss block structure in FOFL.

2.1 Block Structure in HOFL

Block structure in HOFL is realized via two constructs:

- HOFL’s abs construct allows creating functions anywhere in a program, even within other functions.

- HOFL’s bindrec construct allows creating collections of mutually recursive functions anywhere in a program, even within other functions.

Recall that HOFL’s fun construct desugars into nested instances of abs and that top-level HOFL declarations of the form

\[
\text{(def } (I_{name} I_{formal_1}, \ldots, I_{formal_n}) E_{body})\]

desugar into a bindrec of funs.

As a simple example of block structure, consider the following HOFL function declaration:
(def (index-bs x xs)
  (bindrec ((loop (fun (i ys)
        (if (empty? ys)
            -1
            (if (= x (head ys))
              i
              (loop (+ i 1) (tail ys))))))
  (loop 1 xs)))

Note how the local \texttt{loop} function can refer to the parameter \texttt{x} of the enclosing function declaration even though it is not passed as an explicit parameter.

The above program can be expressed without block structure by passing \texttt{x} as an explicit parameter to the \texttt{index-loop} function:

(def (index-no-bs x xs)
  (index-loop 1 x xs))

(def (loop i ys x)
  (if (empty? ys)
    -1
    (if (= x (head ys))
      i
      (loop (+ i 1) (tail ys) x))))

As another example of block structure, consider a block-structured version of a function calculating cartesian products:

(def (cartesian-product-bs xs ys)
  (bindrec ((prod (fun (zs)
        (if (empty? zs)
          #e
          (bind x (head zs)
            (bindrec ((map-duple (fun (ws)
              (if (empty? ws)
                #e
                (prep (list x (head ws))
                  (map-duple (tail ws))))))
              (append (map-duple ys) ; Assume APPEND defined elsewhere
                (prod (tail zs))))))))
  (prod xs)))

The same program can be expressed without block structure by passing an extra list argument \texttt{ys} to the \texttt{prod} function and an extra \texttt{x} argument to the \texttt{map-duple} function:
The ability to refer to names in enclosing scopes without passing them explicitly as parameters is a key advantage of block structure. This advantage may not seem so important in the context of simple examples like those above. A much more convincing example of the importance of block structure is the local definitions of the functions `eval` and `apply` within the `run` function in Fig. 2. In the block-structured version, `eval` takes only a single argument `venv`. Without block structure, it would be necessary for `cdeval` to take three arguments: `venv`, `fenv`, and `genv`. Changing every invocation of `eval` to pass two extra arguments would make it significantly more complex and harder to understand without changing its meaning.

Another advantage of block structure is that it helps to indicate which functions are used where in a program. A locally defined function that is not used in a first-class way can only be used in the region of the program delimited by scope in which it is created.

### 2.2 The Syntax of Fobs

Fobs adds block structure to FOFL via local recursive function declarations that have the following form:

```
(funrec Ebody FD_1 ... FD_k)
```

As in FOFL, each Fobs function declaration `FD` has the form

```
(def (F name I_{formal_1} ... I_{formal_n}) E_{body})
```

As in FOFL, Fobs functions are second-class, and function names are in a different namespace from values. The function declarations in a `funrec` are mutually recursive; any function in the `funrec` may call any other function in the `funrec` in its body. The result of a `funrec` expression is the result of evaluating the final body expression `E_{body}` in a context where all the functions declared in the `funrec` are in scope.

The grammar of Fobs is exactly the same as the grammar of FOFL except for the addition of the `funrec` expression. Unlike FOFL, Fobs does not need to handle top-level function declarations specially, since these can be desugared into `funrec`.

Here is a version of the cartesian product example expressed in Fobs:
(fobs (a b)
  (funrec (bindpar ((xs (range 1 a))
                   (ys (range 1 b)))
               (funrec (prod xs)
                      (def (prod zs)
                           (if (empty? zs)
                               #e
                               (bind x (head zs)
                                    (append (funrec (map-duple ys)
                                                   (def (map-duple ws)
                                                        (if (empty? ws)
                                                            #e
                                                            (prep (list x (head ws))
                                                               (map-duple (tail ws))))))))
                           (prod (tail zs))))))))
(append xs ys)
  (if (empty? xs)
      ys
      (prep (head xs)
             (append (tail xs) ys)))))
  (def (range lo hi)
      (if (> lo hi)
          #e
          (prep lo (range (+ lo 1) hi))))))

2.3 Evaluation and Scope in Fobs

An environment model evaluator for fobs is presented in Fig. 3. The env-run function is parameterized over two scoping mechanisms: one for variable names (vscope) and one for function names (fscope). Each of these two scopes can independently be chosen to be static or dynamic via the following functions:

let static = fun senv denv -> senv
let dynamic = fun senv denv -> denv

We can also model “weird” scoping mechanisms if we so desire:

let empty = fun senv denv -> Env.empty
let merged1 = fun senv denv -> Env.merge senv denv
let merged2 = fun senv denv -> Env.merge denv senv

Similar to HOFL, static scoping in Fobs requires that a function be represented via a closure that associates a function with the environment in which it was created. A Fobs closure must be closed over both the variable environment and function environment in which it was created.

As examples of scoping in Fobs, consider the result of running each of the four programs in Fig. 4 on the argument list [2; 1; 4] using each possible combination of static and dynamic scope for vscope and fscope.
type 'a scoping = 'a Env.env (* static *) * 'a Env.env (* dynamic *) -> 'a Env.env

closure = Clo of fcn * valu Env.env * closure Env.env (* function closures *)

let rec run vscope fscope (Pgm(fmls,body)) ints =
  let flen = length fmls and ilen = length ints
  in if flen <> ilen then
      raise (EvalError ("Program expected " ^ (string_of_int flen) ^ " arguments but got " ^ (string_of_int ilen)))
  else
      let rec eval exp venv (* current var. env. *) fenv (* current fun. env. *) =
          match exp with
          Lit v -> v
          | Var name ->
              (match Env.lookup name venv with
              Some(i) -> i
              | None -> raise (EvalError("Unbound variable: " ^ name)))
          | PrimApp(op, rands) -> (primopFunction op) (evalExps rands venv fenv)
          | If(tst, thn, els) ->
              (match eval tst venv fenv with
              Bool true -> eval thn venv fenv
              | Bool false -> eval els venv fenv
              | v -> raise (EvalError ("Non-boolean test value " ^ (valuToString v) ^ " in if expression")))
          | Bind(name, defn, body) ->
              eval body (Env.bind name (eval defn venv fenv) venv) fenv
          | App(fname, rands) -> apply fname (evalExps rands venv fenv) venv fenv
          | Funrec(body, fcns) ->
              eval body venv
              (Env.fix
               (fun fe -> Env.bindAllThunks (map fcnName fcns)
                (map (fun fcn ->
                     (fun () -> Clo(fcn, venv, fe))))
                fcns)
               fenv))
      and evalExps exps venv fenv = map (fun e -> eval e venv fenv) exps
      and apply fname actuals dvenv dfenv =
          match Env.lookup fname dfenv with
          None -> raise (EvalError ("unknown function " ^ fname))
          | Some (Clo(Fcn(name,formals,body), svenv, sfenv)) ->
              let flen = length formals and alen = length actuals
              in if flen <> alen then
                  raise (EvalError ("Function " ^ name ^ " expected " ^ (string_of_int flen) ^ " arguments but got " ^ (string_of_int alen)))
              else eval body (Env.bindAll formals actuals (vscope svenv dvenv))
              (fscope sfenv dfenv)
              in eval body
          (Env.make fmls (map (fun i -> Int i) ints)) (* initial venv *)
          Env.empty (* initial fenv *)

Figure 3: An environment model evaluator for Fobs parameterized over two scoping mechanisms:
one (fscope) for the function namespace and one (vscope) for the value namespace.
Программа 1
(fobs (n lo hi)
    (funrec (sum-loop lo 0)
        (def (multiple? x) (= 0 (rem x)))
        (def (rem y) (\ y n))
        (def (sum-loop i sum)
            (if (> i hi)
                sum
                (sum-loop (+ i 1)
                    (if (multiple? i) (+ i sum) sum))))))

Программа 2 (добавляет вложенную функцию FUNREC)
(fobs (n lo hi)
    (funrec (sum-loop lo 0)
        (def (multiple? x) (= 0 (rem x)))
        (def (rem y) (\ y n))
        (def (sum-loop i sum)
            (funrec (if (> i hi)
                sum
                (sum-loop (+ i 1) (new-sum i)))
                (def (new-sum z) (if (multiple? z) (inc z) sum))
                (def (inc w) (+ sum w))))))

Программа 3 (переименовывает Z в N в Программе 2)
(fobs (n lo hi)
    (funrec (sum-loop lo 0)
        (def (multiple? x) (= 0 (rem x)))
        (def (rem y) (\ y n))
        (def (sum-loop i sum)
            (funrec (if (> i hi)
                sum
                (sum-loop (+ i 1) (new-sum i)))
                (def (new-sum n) (if (multiple? n) (inc n) sum))
                (def (inc w) (+ sum w))))))

Программа 4 (переименовывает INC в REM в Программе 4)
(fobs (n lo hi)
    (funrec (sum-loop lo 0)
        (def (multiple? x) (= 0 (rem x)))
        (def (rem y) (\ y n))
        (def (sum-loop i sum)
            (funrec (if (> i hi)
                sum
                (sum-loop (+ i 1) (new-sum i)))
                (def (new-sum z) (if (multiple? z) (rem z) sum))
                (def (rem w) (+ sum w))))))

Фигура 4: Примеры иллюстрации сокрушения для обеих переменных и функций среды в Fobs.