CS251 Programming Languages **Handout # 24**
Prof. Lyn Turbak **March 6, 2002**
Wellesley College

Call-by-Value Recursion

This handout discusses a number of subtle issues surrounding the meaning and implementation of a recursive binding construct like `bindrec` in a call-by-value language.

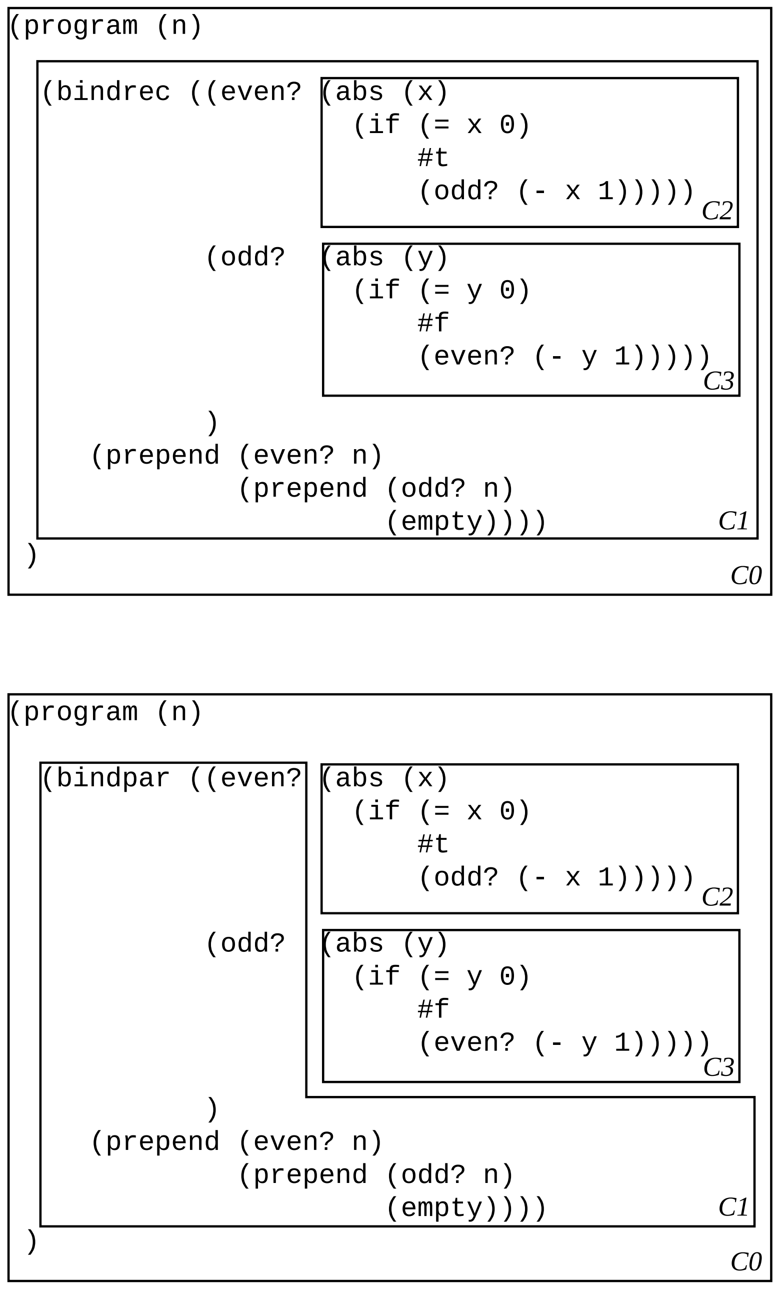
1 The Scope of `bindrec`

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

```
(program (n)
  (bindrec ((even? (abs (x)
                     (if (= x 0)
                         true
                         (odd? (- x 1)))))
            (odd? (abs (y)
                     (if (= y 0)
                         false
                         (even? (- y 1)))))
            )
     (prepend (even? n)
              (prepend (odd? n)
                       (empty)))))
```

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. 1. 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.


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

```
(program (n)
  (bindpar ((even? (abs (x)
                     (if (= x 0)
                         #t
                         (odd? (- x 1)))))
            (odd?  (abs (y)
                     (if (= y 0)
                         #f
                         (even? (- y 1)))))
            )
    (prepend (even? n)
             (prepend (odd? n)
                      (empty))))
 )
```

Figure 1: Contour diagrams illustrating the scoping of `bindrec` and `bindpar`.

2 Evaluating `bindrec`

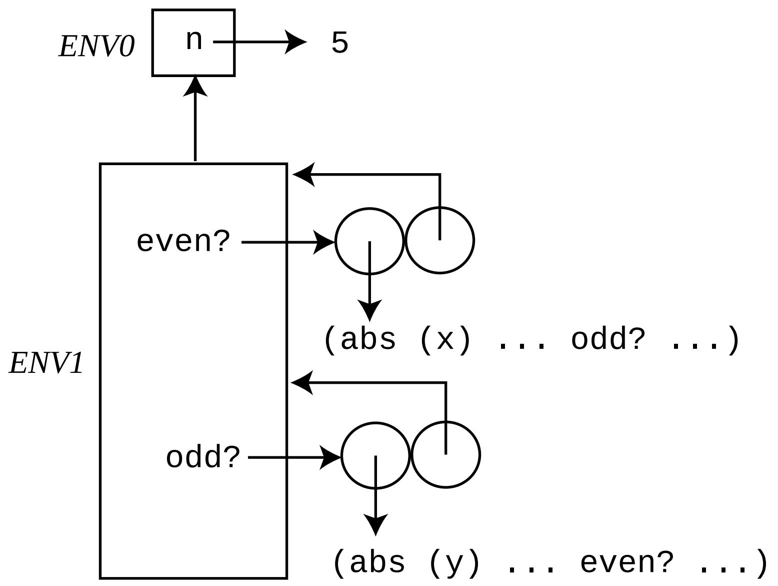
How is `bindrec` handled in the environment model? We do it in three stages:

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 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 was called on the input 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`, we're asking to use the value `x` before we've determined it.

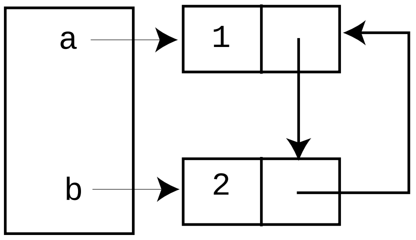
```
(bindrec ((x (+ x 1)))
  (* x 2))
```

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 (prepend 1 b))
          (b (prepend 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, Scheme, ML, 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 `(prepend 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**).

```
(program (n)
  (bindrec ((one-two (pair 1 (abs () two-one)))
            (two-one (pair 2 (abs () one-two)))
            (prefix (abs (num stream)
                      (if (= num 0)
                          (empty)
                          (prepend (fst stream)
                                   (prefix (- num 1)
                                           ((snd stream)))))))
            )
    (prefix n one-two)))
```

Any attempt to use the delayed variables requires applying the thunks to zero arguments (as in the expression `((snd stream))` within the `prefix` function). When the above program is applied to the input 5, the result is the list `(1 2 1 2 1)`.

3 Implementing `bindrec`

Implementing the “knot-tying” aspect of the recursive bindings of `bindrec` within the `env-eval` function of the statically scoped HOFL interpreter is 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? exp)
 (env-eval
   (bindrec-body exp)
   (env-extend
     (bindrec-names exp)
     (map (lambda (defn)
            (env-eval defn ???))
          (bindrec-defns exp))
     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 Scheme’s `let` to name the result of `env-extend`:

```
;; Broken Attempt 2
((bindrec? exp)
 (env-eval
   (bindrec-body exp)
   (let ((new-env (env-extend
                    (bindrec-names exp)
                    (map (lambda (defn)
                           (env-eval defn new-env))
                         (bindrec-defns exp))
                    env)))
     new-env)))
```

This attempt fails because, by the scoping rules of `let`, `new-env` is an unbound variable in `(env-extend ...)`.

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

```
;; Broken Attempt 3
((bindrec? exp)
 (env-eval
   (bindrec-body exp)
   (letrec ((new-env (env-extend
                       (bindrec-names exp)
                       (map (lambda (defn)
                              (env-eval defn new-env))
                            (bindrec-defns exp))
                       env)))
     new-env)))
```

The above clause attempts to use the knot-tying ability of Scheme's own recursive binding construct, `letrec`, to implement HOFL's recursive binding construct. Now the `new-env` within `(env-extend ...)` is indeed correctly scoped. Unfortunately, there is still a problem: because Scheme is a call-by-value language, we come face to face with the same sort of problem encountered in the recursive list example from above. That is, occurrence of `env-eval` within the `map` invocation requires that all its arguments be values before it is invoked. But its `new-env` argument is defined to be the result of a computation that depends on the result returned by this occurrence of `env-eval`. This leads to an irresolvable set of constraints: `env-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:

```
;; Working Attempt 4
((bindrec? exp)
 (env-eval
   (bindrec-body exp)
   (letrec ((new-env (env-extend
                       (bindrec-names exp)
                       (map (lambda (defn)
                              (lambda () ;; Introduce a thunk!
                                (env-eval defn new-env)))
                            (bindrec-defns exp))
                       env)))
      new-env)))
```

Now the `bindrec` clause is sensible, but we are not done. We have changed what names are bound to in the environment! Before, all names were bound to HOFL values. Now, at least some names are bound to thunks that return HOFL values when they are "dethunked" - i.e., applied to zero arguments to retrieve their values.

There are two ways to proceed at this point:

1. *Use thunks everywhere.* We can modify `env-eval` to ensure (1) that *all* entities stored in the environments used by `env-eval` are thunks and (2) that whenever a name lookup is performed, the resulting thunk should be dethunked. This makes sense if you think in terms of types (which will be our next major topic of study in the course). Point (1) says that the type of environments is changing from (variable → value) to (variable → (unit → value)), where unit is the type of one element. Point (2) says that since the result of an environment lookup is now of type (unit → value) , it must be applied to zero arguments in order to get a value.

   A drawback of this approach is that it is a global change that affects many other parts of the evaluator. It requires:

   - changing `env-run` to "thunkify" (i.e., wrap in a thunk) the integer arguments of the program.
   - changing `funapply` to thunkify the results of evaluating the argument expressions.[1]

[1]Thunkifying the evaluation of the argument expressions rather than the results of the evaluation would lead to call-by-name semantics, not call-by-value semantics.

   - changing the `varref` clause of `env-eval` to dethunk the thunk returned by a name lookup.

2. *Use thunks only for* `bindrec`. A more modular way to handle the thunks introduced into the environment by `bindrec` is to allow environments to associate names with *either* (1) a HOFL value or (2) a thunk that produces a HOFL value when dethunked. In this approach, the only additional modification we need to make to `env-eval` is in the `varref` clause:

   ```
   ((varref? exp)
    (let ((probe (env-lookup (varref-name exp) env)))
      (cond ((unbound? probe)
             (throw 'unbound-variable (varref-name exp)))
            ((procedure? probe)
             (probe)) ; *** Dethunk thunks introduced by BINDREC.
            (else probe)))) ; *** No dethunking necessary for HOFL values
   ```

   Here, we use the Scheme predicate `procedure?` distinguish thunks (for which `procedure?` returns true) and HOFL values (for which `procedure?` returns false).

The changes in the second approach fine, but they are inefficient. In particular, the dethunking process ends up re-evaluating a recursive definition expression every time it is looked up in the environment.

It turns out that Scheme has a more efficient mechanism for delaying computations than thunks. The construct `(delay` E`)` delays the expression of E by returning a **promise**; if E_{prom} is an expression denoting a promise, its delayed computation can be forced via the application `(force` E_{prom}`)`. The promise "remembers" the value of its computation, so an attempt to force a promise the second time performs no computation but returns the previously computed value.

By replacing all instance of thunks and dethunking by `delay` and `force` in the code presented above, a more efficient implementation of recursive binding can be achieved. The highlights are shown in Figure 2.

```
(define env-eval
  (lambda (exp env)
    (cond
      ⋮
      ((varref? exp)
       (let ((probe (env-lookup (varref-name exp) env)))
         (cond ((unbound? probe)
                (throw 'unbound-variable (varref-name exp)))
               ((promise? probe)
                (force probe)) ; *** Force promises introduced by BINDREC.
               (else probe)))) ; *** No force necessary on non-promises
      ⋮
      ((bindrec? exp)
       (env-eval
         (bindrec-body exp)
         (letrec ((new-env (env-extend
                             (bindrec-names exp)
                             (map (lambda (defn)
                                    ; *** Evaluation of defns must be delayed
                                    (delay (env-eval defn new-env)))
                                  (bindrec-defns exp))
                             env)))
            new-env)))
       ⋮
       )))
```

Figure 2: Implementing `bindrec` with `delay` and `force`.