CS450: Structure of Higher Level Languages

Fall 2020 Assignment 7, Part 1

Due: Sat. December 5 on Gradescope

Taken from assignments by Profs. Carl Offner and Ethan Bolker

Modifying The Metacircular Evaluator

In this assignment we will add some features to `s450` in order to learn about different methods of argument passing and get some simple experience with continuations.

There are two parts to this assignment. Part 1 is much longer than Part 2. However, they are completely independent, so you can do them in any order. You can even do Part 2 first.

I will collect two files: `s450.scm` will contain your Scheme code for this assignment. Another file will contain your design notes (what changes did you have to make to `s450.scm` to support this feature), alternatives you considered, problems you encountered, some indication of how you tested your code (but not complete test scripts!), and so on. Please do not leave your notes for the end. It is an important part of this assignment.

Guidelines

Please do not read the section of Chapter 4 that talks about delayed arguments. You will just get confused and end up writing code that is incorrect. What the book means by delayed evaluation in that section is not the same as what we mean in this assignment. You will not learn anything useful by reading that part of the book. In fact, I would suggest not reading the book at all for this assignment. I'm explaining everything you need to know in class. If you have a question, it's much better to ask me than to try to dig it out from the book.

Various Methods of Argument Passing

In Scheme, and in s450, all arguments are passed by value. Other languages use other semantics:

- Algol 60 introduced the concept of call-by-name. There is still a lot of interest in call-by-name in the functional programming world.
- APL and the traditional versions of lisp (including Emacs lisp) all use dynamic rather than static scoping. Perl uses static scoping for formal parameters. However, for all other declared variables, Perl uses both dynamic scoping (for local variables) and static scoping (for my variables).
- Pascal and Modula-2 have var parameters that implement call-by-reference. C does not have call-by-reference, but the programmer can get the same functionality by passing (by value) a pointer to a variable. (In spite of what you often hear people say, however, this is not call-by-reference; it is a way that the programmer can simulate call-by-reference.) C++ does have call-by-reference as an option. Java uses call-by-reference for objects (though not for primitive types). Fortran and Perl both use call-by-reference for everything.

We want to build all of these into s450, in an extensible way that allows for other argument semantics not yet imagined. To do so we extend the special form lambda, giving each formal argument an optional tag, so we get the following kind of syntax:

```
(lambda (x
         (delayed y)
         (dynamic z)
         (reference w))
  <body>)
```

The `<body>` refers to the formal arguments `x, y, z, w` in the usual way, with no special syntax required to deal with the different methods of argument passing. (That is, if you just looked at the code in the body, you would have no way of knowing that `w` was a reference argument. That information is only contained in the formal argument list.) The tags (delayed, dynamic, and reference) in the formal argument list are there simply to indicate how the corresponding actual arguments are handled at run-time at a procedure call site (that is, at a point in the program where a procedure is called).

- The formal argument `x` represents a call-by-value argument, using what you might think of as applicative-order evaluation for that argument. That is, the actual argument (which may of course be any Scheme expression) is evaluated, and its value is passed to the procedure. The procedure body is then executed. This is the argument-passing mechanism that is already implemented in s450. We don't need to make any changes to support this.
- The delayed formal argument `y` is a way of implementing call-by-name (or call-by-need). It could also be thought of as a form of call-by-value, except that it uses what you might think of as normal-order evaluation for that argument. That is, the actual argument is not evaluated at the call site, but is packaged (together with the calling environment) into a thunk that is passed to the body of the procedure. The thunk is forced only when its value is definitely needed in the course of the execution of the procedure body. This is discussed in more detail below.
- The dynamic formal argument `z` specifies dynamic scoping, as we discussed in class. It is yet another version of call-by-value, again with applicative-order evaluation. That is, the actual argument is evaluated before the procedure body is executed. However, in this case, the actual argument is evaluated in the dynamic environment of the program at run-time, rather than in the (static, lexical) environment associated with the procedure object.
- The reference formal argument `w` represents a reference to the actual argument. There is an important difference between reference arguments and the other kinds of arguments: When a lambda expression is applied, the actual arguments for the other kind of arguments (call-by-value, delayed, and dynamic) can be arbitrary expressions, but (in our implementation) an actual reference argument must be a symbol with a value in the environment in which the lambda expression is evaluated. (The new special form defined? may come in handy.)

Here's an example illustrating the difference between call-by-value and reference arguments:

```
s450==> (define f (lambda (x (reference y))
                (display (cons x y))(newline)
                         (set! x 10)
                        (set! y 20)
                (cons x y)))
f

s450==> (define a 1)
a

s450==> (define b 2)
b

s450==> (f a b)
(1 . 2)         ; from the display statement in f
(10 . 20)       ; pair returned by f, displayed at end of r-e-p loop

s450==>a
```

```
1               ; actual argument a (passed by value) is unchanged

s450==>b
20              ; actual argument b (passed by reference) has new value

s450==> (f a 2)
error: lambda expression -
       actual argument for reference formal argument y must
       be a defined symbol
reset and all that stuff ...

==>
```

I suggest that you not try to implement all these capabilities at once. Rather, implement them one at a time. Here is one way to go about it – it's the way I used myself:

1. First, make a fresh copy of your `s450.scm`. Make sure this file is clean, well-organized, and well-commented. Feel free to rearrange it in any way you like. Try hard to make it as easy to understand as possible. This will of course help me when I read it, but – trust me on this – it will also pay off greatly for you in that it will make your job in this assignment much easier.

2. Get in the practice of backing up your work. You can of course use a tool such as CVS to help you with this. But since you are in effect a one-person project, it's just as simple to do it by hand, like this:

   - Before you make any changes to s450.scm, create a backup copy: copy s450.scm to s450_1.scm. This, of course, is so that if you make some sort of terrible mistake that you can't correct, you can always get back to where you started. In general, whenever you get to a relatively stable version of the interpreter, back it up in this fashion, using a new name each time. (Note, however, that I will not look at any of your backup copies. Make sure that s450.scm contains your latest and greatest version when I collect your files.)
   - Implement `delayed` arguments in s450.scm. When you have done this, make a backup copy and call it s450_2.scm.
   - Implement `dynamic` arguments in s450.scm. When you have done this, make a backup copy and call it s450_3.scm.
   - Implement `reference` arguments in s450.scm. When you have done this, make a backup copy and call it s450_4.scm.
   - Finally, when you are done with all this (or at some point), implement Part 2 of this assignment, also in s450.scm.

Of course, you can implement these various features in any order you find convenient.

Here are the specific things you should do, and some hints for your design:

- `delayed` arguments: The key idea is that when you process a procedure call, instead of evaluating each argument and placing that value in the new frame, you package up the unevaluated argument (which is an expression) together with the current environment (i.e., the environment in which that argument needs to be evaluated) into an object called a thunk. The thunk becomes the value that is placed in the frame (corresponding to the formal argument of the procedure). When (and only when) the value of the formal argument is actually needed when evaluating the body of the procedure, then the thunk is "forced". That is, the expression together with the environment (both stored in the thunk) are passed to `xeval` to get the value that is required at that point.

  One important question is this: how do you know, when you are processing a procedure call (in `xapply`) whether or not you need to create a thunk or just evaluate the actual argument? The answer is simple: that information is contained in the corresponding formal argument. The formal argument list is part of the procedure object, and so is available to you when you need it in setting up for the procedure call. If the formal argument is tagged as delayed, you need to create a thunk. Otherwise, you just evaluate the actual argument in the usual fashion.

Note, however, that if the formal argument is represented in the lambda expression as `(delayed x)`, for instance, then when you make the new frame for a procedure call, the "delayed" keyword is not part of that frame. The frame contains x and the corresponding thunk.

Once you have implemented thunks, you can support stream processing in s450. Do it like this:

1. Implement a special form `cons-stream` in such a way that

   ```
   (cons-stream elmt strm)
   ```

   is equivalent to (i.e., is processed as) a cons pair whose `car` is what `elmt` evaluates to, and whose `cdr` is a thunk holding `strm`.

2. Implement the primitive procedures

   - `stream-car`
   - `stream-cdr`
   - `stream-null?`

   as well as the object

   - `the-empty-stream`

   `Be careful:` These stream operations do not use the `delayed` arguments you have implemented. (At least, they don't the way I did this. You may do it differently.) But they do use the machinery you created in order to support the delayed arguments – what was referred to as "thunk".

3. Test these stream operations to make sure they work as they should, and report on what you find in your notes.

- `dynamic arguments`: The only difference between these and ordinary call-by-value arguments is that the environment that is passed to `xeval` when they are evaluated is different. At any one time there is only one active dynamic environment. Therefore, I suggest that you keep a global variable, perhaps `the-dynamic-environment`. This environment is a list of frames, just like the static environment we have been using up to now. However, you need to manage this variable like a stack. Each time a function is invoked, you push a frame onto `the-dynamic-environment` (using `cons`).

  The frame that you push onto `the-dynamic-environment` is really the same frame that is added to the static environment. The difference is that you are not adding to the procedure's environment (i.e., the environment that the procedure object was originally defined in), but instead you are adding to the dynamic environment. (That's why it's a stack.)

  Each time a function terminates, you have to restore `the-dynamic-environment` to its previous state. Thus, after the function call is complete, you pop that frame off of `the-dynamic-environment`.

  The simplest way to add the frame is to do it when you are adding a frame to the static environment, i.e., in `xtend-environment`. But be careful: the return value of `xtend-environment` has to be the static environment, not the dynamic environment.

  NOTE: you might be tempted to call `xtend-environment` twice from within `xapply` – once to extend the static environment and a second time to extend the dynamic environment. This will not work, because you will get two copies of the new frame. The problem with this is that any updates (e.g., using `set!`) to one frame will not be visible in the other frame.

  Where does the dynamic environment get restored to its previous value after the procedure call? One obvious place is in `xapply`. But again you have to be careful: the return value of `xapply` is the return value of the procedure. Make sure you don't throw that value away when you restore the dynamic environment. (You may have another idea of where to restore the dynamic environment. Whatever you do, document it carefully.)

  Here is an example of dynamic scoping that you can use to test your implementation. (We may also have talked about some other examples in class that you could also use.) If you start in s450 with

  ```
  (define f (lambda(x)(lambda(y)(cons (g x) y))))
  (define g (lambda((dynamic z))(cons z 4)))
  (define h (f 2))
  (define x 1)
  ```

  then

```
s450==> (h 5)
((1 . 4) . 5)
```

Can you see why this is true? This would be true even if the (define x 1) was placed before (define h (f 2)). Think of it this way:

1. The global environment (`the-global-environment`) contains a binding of x to the value 1, since `(define x 1)` is evaluated at the top level.
2. When `(f 2)` is evaluated, x is bound to 2 in a frame that is added to the static environment for f. The body of f is then executed, thereby defining h as the procedure object that looks like this:

   ```
   parameter:   y
   body:        (cons (g x) y)
   environment: (((x 2)) the-global-environment)
   ```

3. Now `h` is applied to its argument, which is 5. Note that at the point when h is applied, we are at the outermost dynamic scope. (In effect, we are at the s450 prompt; the dynamic environment is `the-global-environment`.) The binding of the formal argument y of h to 5 is added to both the static and the dynamic environments, as follows:

   To add it to the static environment, we add it to the environment of the procedure h. The new static environment is therefore

   ```
   (((y 5)) ((x 2)) the-global-environment)
   ```

   To add it to the dynamic environment, on the other hand, we add it to the current dynamic environment, which as we noted above, is just `the-global-environment`. The new dynamic environment is therefore

   ```
   (((y 5)) the-global-environment)
   ```

   Now g takes its argument from the dynamic environment. So that is where its actual argument x is looked up, and the value 1 is obtained.

   If z were not declared to be a dynamic argument for g, then the actual argument would be evaluated in the static environment, yielding the value 2, and so (h 5) would evaluate to ((2 . 4) . 5)

You should make sure you understand this, and also that your interpreter understands it.

- `reference arguments:` Actually, much of the implementation of reference arguments follows that of delayed arguments. When a lambda expression is invoked, you need to create a thunk-like object (give it a different name, though, like "reference") that holds the actual argument (which must be a symbol) and the environment in which it is found.

  During evaluation, if you encounter a formal argument that is bound to such a "reference", then you `xeval` the actual argument in the saved environment. (This is the same thing as what you do when you force a thunk.)

  So what is the difference between reference and delayed arguments? The difference is this:

  – When you make a change to a delayed argument, you change what that argument is bound to. Setting the argument equal to 4, for instance, means that it is no longer bound to a thunk.

  – When you make a change to a reference argument, you do not change what the argument is bound to, but you change the value of the symbol it is bound to. That is, you are actually reaching into the environment carried along with the reference argument and changing the value bound to the referenced symbol.

  Thus, the difference between reference arguments and delayed arguments appears in their different behavior under `set!`. For example,

  ```
  ((lambda ((reference x)(delayed y)) (set! x 3)(set! y 4)) a b)
  ```

changes the value of a, but not the value of b.

If you chase through the code to see how `set!` is implemented, you will find that the heart of it is in `set-variable-value!`. If you have changed the environment data structures then this function will have to change too. If the binding value is a reference argument, simply invoke `set-variable-value!` recursively, with the name of the actual argument (in the above case 'a') and the saved environment.

To test, you should confirm that if you start in s450 with

```
(define u 3)
(define x 10)
(define t 2)
(define g (lambda ((reference x))(f x x) t))
(define f (lambda ((reference x)(reference y))
           (set! x 5) y))
```

then

```
s450==> (f u u)
5
S450==> (f x x) ; There is a potential problem, when the formal
5          ; and actual arguments have the same name.
        ; Check that you have avoided it.
S450==>(g t)
5
```

What about `define-variable!`? Do you need to make any changes to that procedure? (Answer this question in your notes, as usual.)