Homework 7 Notes

Robert Steele

Final Implementation:

My final implementation relied heavily on two new expressions I wrote:

resolve-params: takes a list of procedure parameters and evaluates each parameter, returning a list of symbols. For each parameter it checks if the parameter is a delayed, dynamic, or reference parameter and if it is then the symbol is extracted. Otherwise, the parameter remains unchanged.

and:

resolve-args: which takes the same list of procedure parameters as resolve-params as well as the list of arguments and the environment the procedure is being called in. It first checks to make sure the number of parameters match the number of arguments and then performs a similar process to resolve-params. For each parameter/argument pair it checks if the parameter is one of those special cases (delayed, dynamic, reference) and if it is then it performs a special operation before recursing to the next pair. The special operations are as follows:

Delayed: If the parameter is delayed then we create a thunk with the argument and the environment and recurse to the next pair.

Dynamic: If the parameter is dynamic then we evaluate the argument but instead of passing the environment that was passed to resolve-args we pass the-dynamic-environment. Then we recurse to the next pair.

Reference: If the parameter is a reference then we check to make sure the argument is a symbol and if it is, we create a reference with the argument and the environment and recurse to the next pair.

Default: If the parameter is none of the previous cases then we xeval the argument in the environment and recurse to the next pair.

With these two expressions we can handle every single type of formal argument. All that is left to do is handle the thunk and reference objects whenever their values are required and manage the dynamic environment.

Handling Thunks: When a variable is looked up and a thunk is found the thunk is forced (the argument of the thunk is evaluated in the environment of the thunk) and the value is returned. I implemented this in lookup-variable-value.

Handling the Dynamic Environment: I changed xtend-environment so that on every call a single frame was created which was added to both the base-frame and the-dynamic-environment. After eval-sequence finishes in xapply I reset the dynamic environment to its previous state (cdr the-dynamic-environment). This creates a push pop effect where every time xtend-environment is called in xeval a new frame with all relevant variables and values is pushed onto the dynamic stack and once the procedure had been applied (and all relevant dynamic values are processed) the first frame of the dynamic stack is popped. To ensure the dynamic stack is restored in the event of an exception I also call:

```
set! the-dynamic-environment the-global-environment
```

when entering the main loop in s450.

Handling References: When a variable is looked up and a reference is found the reference is evaluated (the argument of the reference is evaluated in the environment of the reference) and the value is returned. As with thunks I implemented this in lookup-variable-value. But there is one other case to be addressed with reference values. When assigning to a variable that has a reference as a value, we do not want to mutate that variable but rather the variable it references. To do this we check for a reference value in set-variable-value!. If we find a reference then instead of mutating that variable, we recursively call set-variable-value! with the reference value, new value, and reference environment as arguments. This will recursively search until an argument has been found that does not have a reference value at which point the value will be mutated to the new value.

Note: We do not need to worry about checking for reference values when defining. This is because a definition only operates within the first frame of the environment it is called in and should override whatever previous values are defined. We can think of it this way. In assignment we look to change the value of a variable. But in the case of a variable that references another variable the value of the original variable is the value of the variable that it references. For this reason, when we attempt to change the value of the first variable, we must instead mutate the referenced variables value. With a definition there is no consideration given to the value of something that is previously defined. We simply mutate the binding of the variable to the value given in the definition or create a new binding if none exist.

Changes to xeval: I changed the call to xapply in xeval. Instead of calling list-of-values on the operands of the procedure I simply pass the operands directly to xapply. I also passed the current static environment to xapply. I also added checks for eof-object? and exit-code?. If those objects are encountered I jump to call/cc with the expression as the argument and perform checks there to see what should be done.

Changes to xapply: As I said before instead of calling list-of-values on the operands of the expression before calling xapply, xeval instead passes the operands directly in the value arguments. I also changed xapply to take the environment from xeval. This allowed me to do two things. In the case of a primitive procedure, I could call:

```
 (list-of-values arguments environment)
```

to evaluate all the arguments in the environment. And because list-of-values was not called before xapply none of the arguments have yet been evaluated so in the case of a user-defined-procedure we can resolve the arguments as needed before passing them to xtend-environment. This is where I called my procedures resolve-params and resolve-args. So instead of calling:

```
(xtend-environment (procedure-parameters procedure) arguments (procedure-environment procedure)
```

I call:

```
(xtend-environment (resolve-params (procedure-parameters procedure))
                   (resolve-args (procedure-parameters procedure) arguments environment)
                   environment)
```

This has the effect of extending the current static environment (and the dynamic environment) with a list of parameters that have been properly resolved and a list of arguments that have been thunkified, dynamically evaluated, referenced, or evaluated as the need may be. This extended environment is passed to eval-sequence which evaluates the procedure with the given arguments and returns a value which is then returned after popping the dynamic stack.

s450error and call/cc: I implemented exit, eof-exit, and exceptions with call/cc in the main loop of the s450 prompt with save_continuation.scm as my model. I simply defined a target (interrupt in this case) which is overwritten in the call to call/cc. Then I wrote a method s450error which takes multiple args and calls interrupt with a list tagged with s450error that is evaluated in the s450 loop. The s450 loop evaluates the return value of call/cc to see if it has received an error, exit, or eof object. If it has received one it handles it accordingly and if not simply continues evaluation as per usual.

Other Notes (Problems, Alternatives, etc…):

It took me way too long to realize that "delayed parameters" had nothing to do with the traditional delay and force. I implemented that first and was confused when Gradescope said everything was wrong. Finally, I read the directions carefully and realized I was thinking about this in the wrong way.

I built something with some similarities to resolve-params and resolve-args but could not figure out why things were being evaluated before reaching xapply. That is until I found list-of-values. After that it took me a while to decide on the best way to bypass list-of-values. Now it seems obvious what I should have done but before I was not sure exactly where the best place to intercept them was. In the end I obviously settled on delaying evaluation of the arguments until xapply when I could decide which variables needed to be evaluated and which needed to be delayed/dynamically evaluated/referenced.

One thing I considered was checking if the procedure was a primitive procedure in xeval and resolving the arguments there before passing them to xapply. I spent a while doing that but then I realized it was just clunky to do it that way and it would be far simpler to add a parameter to xapply and pass it the environment so it could evaluate the arguments if the procedure was a primitive procedure.

Initially I evaluated thunks in xeval assuming that would be the most obvious place to force a thunk. But I kept failing the second delay test on Gradescope and finally I decided to try forcing thunks in lookup-variable-value which let me pass the tests. I thought that, "the thunk is forced only when its value is definitely needed in the course of the execution of the procedure body" implied that we should only evaluate it if we absolutely needed to. Like if we passed it to xeval for example. Otherwise, it would be returned as a thunk. Evidently that is not what was expected. That took me a while to figure out.