C311 Lab #3
Representation Independence:
Representation Independent Interpreters

Will Byrd
webyrd@indiana.edu

February 5, 2005
(based on Professor Friedman's lecture on January 29, 2004)

1 Introduction

On January 29, 2004, Professor Friedman gave a lecture showing how to make an environment-passing interpreter representation independent (at least with respect to environments). Professor Friedman believes that this lecture is the most important of the semester, since the remainder of C311 builds on the ideas of interpreters, environments, and representation independence. A student's success in C311 is therefore strongly related to how well he or she understands the material presented in this lecture.

This document is based on the notes I took during Professor Friedman's lecture. I hope you will take advantage of this document in order to thoroughly master this important material.

2 How to Read this Document

When you first read this document, do not overly concern yourself with how the interpreter works, or even what the interpreter does. Concentrate instead on how a representation dependent Scheme procedure that uses environments can be made representation independent.

As you master the concepts of representation independence and environments, I strongly encourage you to re-read this document, this time with the intent of understanding how the interpreter itself works.

3 A Simple Programming Language

The interpreter we are going to write will interpret expressions in a simple subset of Scheme that includes the following five expression types:

numbers such as `0` and `137`, which evaluate to themselves.

variables represented as symbols such as `'a` and `'x`, which evaluate to whatever values they are bound to within the current environment.

λ-expressions such as $(\lambda\ (x)\ x)$, which create and return new procedures (the identity procedure, in this case).

procedure applications of the form (*proc arg*), which apply the procedure *proc* to the argument *arg*. For example, $((\lambda\ (x)\ x)\ 9)$ applies the identity procedure to the number `9`, returning the number `9`.

if-expressions such as (**if** 5 6 7), which evaluates to 7[1].

Professor Friedman pointed out that only variables, λ-expressions, and procedure application are fundamental expression types[2], as we can use these three expression types to represent numbers and **if**-expressions.

4 A Simple Interpreter

Here is a simple interpreter for our little language:

```
(define eval-exp
  (λ (exp env)
    (kmatch exp
      [(n) ,n (guard (integer? n)) n]
      [(var) ,var (guard (symbol? var)) (env var)]
      [(id body) (λ (,id) ,body)
       (λ (arg)
         (eval-exp body (λ (var)
                          (if (eq? id var)
                              arg
                              (env var)))))]
      [(test conseq alt) (if ,test ,conseq ,alt)
       (if (eval-exp test env)
           (eval-exp conseq env)
           (eval-exp alt env))]
      [(rator rand) (,rator ,rand)
       ((eval-exp rator env) (eval-exp rand env))])))
```

The *eval-exp* procedure takes an expression[3] written in our simple language, along with an environment containing name/value bindings, and returns the result of evaluating the expression within the context of that environment.

For example, let's evaluate the expression ((λ (x) x) 3) in the context of the empty environment. The empty environment contains no name/value bindings, and therefore signals an error whenever it is applied to a variable.

```
(eval-exp '((λ (x) x) 3)
  (λ (var)
    (error 'empty-env "unbound variable ~s" var)))
```

The result of this call to *eval-exp* is 3.

The call to *error* makes this code somewhat cumbersome. Following the convention Professor Friedman used in lecture, from now on we will represent the call to *error* as the symbol ⊥ (pronounced "bottom"). Here are three test programs written using this new notation, along with the values they return[4]:

(*eval-exp* '((λ (x) x) 3) (λ (*var*) ⊥)) ⇒ 3

(*eval-exp* '((λ (x) y) 3) (λ (*var*) ⊥)) signals the error: Error in empty-env: unbound variable y.

(*eval-exp* '((λ (x) y) 3) (λ (*var*) (**if** (*eq? var* 'y) 7 ((λ (*var*) ⊥) *var*)))) ⇒ 7

[1] Actually, **if**-expressions are not very useful in this language, since we have no way of representing false values. We can easily remedy this problem by extending our language to include the boolean expressions #t and #f, which evaluate to themselves.

[2] You may recognize these three expression types, as they form the simple lambda calculus.

[3] More correctly, *eval-exp* takes a *datum*, such as a quoted list, that represents the expression.

[4] Of course, a much more comprehensive test suite would be required to sufficiently test even our simple interpreter.

5 Representation Independence

The interpreter presented in the last section is *representation dependent*: we can tell by looking at the code for *eval-exp* that the environment is represented by procedures.

Although our interpreter works, we can rewrite the interpreter to hide how environments are represented; we can make our interpreter *representation independent*. That way, if we ever wish to change the way in which we represent environments, we do not need to modify the code for *eval-exp*. As our programs become more complex, the benefit of hiding the representation of key datatypes increases.

Why would we ever wish to change the underlying representation of a datatype, such as the environments used by our interpreter? Efficiency is one possible reason. By hiding how environments are implemented from the code that uses environments, we are free to switch to a more efficient implementation of environments in the future.

Making datatypes representation independent also makes it easier to make more dramatic changes to our programs, such as porting a program written in Scheme to C. Unlike Scheme, the C programming language does not support higher-order procedures[5]. By making all of our datatypes representation independent, we can convert datatypes represented by higher-order procedures to representations that use data structures supported by C.

6 A Representation Independent Interpreter

There are three places in our interpreter where it is obvious that we are using procedures to represent environments. The variable and λ-expression clauses within *eval-exp* both expose this implementation detail. When calling *eval-exp* we must pass in a procedure representing a new environment, which similarly makes it clear that we are representing environments as procedures.

In the process of making our interpreter representation independent, we will create three new *interface procedures*, one for each place in our code that exposes our representation of environments. Let's begin with the code that actually calls the *eval-exp* procedure:

```
(eval-exp '((λ (x) x) 3) (λ (var) ⊥))
```

In this call to *eval-exp* we are passing in a new, empty environment created by the expression (λ (*var*) ⊥)[6]. We want to replace this expression with a call to a new procedure, which we shall call *empty-env*:

```
(define empty-env
  (λ (some-args)
    some-body))
```

We do not yet know which arguments *empty-env* should take, nor what expression should be the body of this new procedure. Fortunately, the body of *empty-env* can be copied directly from the call to *eval-exp*[7]:

```
(define empty-env
  (λ (some-args)
    (λ (var) ⊥)))
```

The free variables within the body we just copied now become *empty-env*'s arguments. The expression (λ (*var*) ⊥) does not contain any free variables, so *empty-env* does not take any arguments:

[5] A *higher-order* procedure is a procedure that creates and returns a new procedure at runtime.

[6] Remember that ⊥ represents the code: (*error* 'empty-env "unbound variable ~s" *var*).

[7] The Emacs command '`kill-sexp`', which by default is bound to the key sequence Escape-Control-k, can be used to delete an entire Scheme expression from a file, copying the expression to the Emacs kill-ring. The '`yank`' command, usually bound to Control-y, can then be used to paste the Scheme code to another location in the file.

```
(define empty-env
  (λ ()
    (λ (var) ⊥)))
```

We have defined the first of our environment interface procedures, and can now use *empty-env* within the call to *eval-exp*. In particular,

```
(eval-exp '((λ (x) x) 3) (λ (var) ⊥))
```

now becomes

```
(eval-exp '((λ (x) x) 3) (empty-env))
```

The call to *eval-exp* is now representation independent, as it is impossible to determine from the call alone how environments are represented.

Let's move on and make the *eval-exp* procedure itself representation independent. Here, once again, is the code for *eval-exp*:

```
(define eval-exp
  (λ (exp env)
    (kmatch exp
      [(n) ,n (guard (integer? n)) n]
      [(var) ,var (guard (symbol? var)) (env var)]
      [(id body) (λ (,id) ,body)
       (λ (arg)
         (eval-exp body (λ (var)
                          (if (eq? id var)
                              arg
                              (env var)))))]
      [(test conseq alt) (if ,test ,conseq ,alt)
       (if (eval-exp test env)
           (eval-exp conseq env)
           (eval-exp alt env))]
      [(rator rand) (,rator ,rand)
       ((eval-exp rator env) (eval-exp rand env))])))
```

Both the variable clause and the λ-expression clause of **kmatch** expose that we are using a procedural representation of environments. First we will concentrate on the variable clause:

```
(define eval-exp
  (λ (exp env)
    (kmatch exp
      ---
      [(var) ,var (guard (symbol? var)) (env var)]
      ---)))
```

According to this code, when the expression to be evaluated is a symbol representing a variable, the *eval-exp* procedure applies the current environment to that variable and returns the result. Since the code uses a procedure application to apply the environment to the variable, environments obviously must be represented by procedures.

We want to replace the expression (*env var*) with a call to a new procedure, *apply-env*:

```
(define apply-env
  (λ (some-args)
    some-body))
```

Once again, we do not yet know which arguments *apply-env* should take, nor what expression should be the body for this new procedure. However, we can use the same trick that we used for *empty-env*, and copy the body of *apply-env* directly from the right hand side of the variable clause within *eval-exp*:

```
(define apply-env
  (λ (some-args)
    (env var)))
```

And, once again, the arguments of *apply-env* are just the free variables within the new body of *apply-env*. In this case the body has two free variables, *env* and *var*. Therefore, *apply-env* will take two arguments, *env* and *var*:

```
(define apply-env
  (λ (env var)
    (env var)))
```

We have finished writing *apply-env*, and can now rewrite the variable clause of *eval-exp* using *apply-env*. The original code,

```
(define eval-exp
  (λ (exp env)
    (kmatch exp
      ---
      [(var) ,var (guard (symbol? var)) (env var)]
      ---)))
```

becomes

```
(define eval-exp
  (λ (exp env)
    (kmatch exp
      ---
      [(var) ,var (guard (symbol? var)) (apply-env env var)]
      ---)))
```

Someone looking at the variable clause of *eval-exp* can no longer tell that we are representing environments as procedures. Someone could gain this information, however, by looking at the λ-expression clause of *eval-exp*:

```
(define eval-exp
  (λ (exp env)
    (kmatch exp
      ---
      [(id body) (λ (,id) ,body)
       (λ (arg)
         (eval-exp body (λ (var)
                          (if (eq? id var)
                              arg
                              (env var)))))]
      ---)))
```

Once again, we wish to replace the representation dependent code with a call to a new interface procedure. This time we will name our interface procedure *extend-env*:

```
(define extend-env
  (λ (some-args)
    some-body))
```

The representation dependent code in the λ-expression clause is the following expression that returns a new extended environment:

```
(λ (var)
  (if (eq? id var)
      arg
      (env var)))
```

As before, we copy this expression into the body of our interface procedure:

```
(define extend-env
  (λ (some-args)
    (λ (var)
      (if (eq? id var)
          arg
          (apply-env env var)))))
```

And, as before, the free variables in the expression we just copied become the arguments of our interface procedure:

```
(define extend-env
  (λ (id arg env)
    (λ (var)
      (if (eq? id var)
          arg
          (apply-env env var)))))
```

Now that we have defined our final interface procedure, we can rewrite the λ-expression clause of *eval-exp*. The original code,

```
(define eval-exp
  (λ (exp env)
    (kmatch exp
      ---
      [(id body) (λ (,id) ,body)
       (λ (arg)
         (eval-exp body (λ (var)
                          (if (eq? id var)
                              arg
                              (env var)))))]
      ---)))
```

becomes

```
(define eval-exp
  (λ (exp env)
    (kmatch exp
      ---
      [(id body) (λ (,id) ,body)
       (λ (arg)
         (eval-exp body (extend-env id arg env)))]
      ---)))
```

Here is the new version of our interpreter:

```
(define empty-env
  (λ ()
    (λ (var) ⊥)))

(define extend-env
  (λ (id arg env)
    (λ (var)
      (if (eq? id var)
          arg
          (apply-env env var)))))

(define apply-env
  (λ (env var)
    (env var)))

(define eval-exp
  (λ (exp env)
    (kmatch exp
      [(n) ,n (guard (integer? n)) n]
      [(var) ,var (guard (symbol? var)) (apply-env env var)]
      [(id body) (λ (,id) ,body)
       (λ (arg)
          (eval-exp body (extend-env id arg env)))]
      [(test conseq alt) (if ,test ,conseq ,alt)
       (if (eval-exp test env)
           (eval-exp conseq env)
           (eval-exp alt env))]
      [(rator rand) (,rator ,rand)
       ((eval-exp rator env) (eval-exp rand env))])))
```

A programmer looking at our new definition of *eval-exp* has no way of knowing how environments are represented; our new interpreter is therefore representation independent with respect to environments.

We still have one task left before we are finished with our new version of *eval-exp*. We changed our interpreter, so we had better check that our test programs still work properly:

(*eval-exp* '((λ (x) x) 3) (*empty-env*)) ⇒ 3.

(*eval-exp* '((λ (x) y) 3) (*empty-env*)) signals the error: Error in empty-env: unbound variable y.

(*eval-exp* '((λ (x) y) 3) (*extend-env* 'y 7 (*empty-env*))) ⇒ 7.

All of our test programs work as before, as expected.

7 Is this Interpreter Really Representation Independent?

Our current version of the interpreter is representation independent, but only with respect to environments. Our little language contains both λ-expressions and procedure application, which create and apply procedures, respectively. It is obvious from looking at the code for *eval-exp* that our interpreter uses Scheme procedures to represent the procedures of our little language. Our interpreter is representation independent with respect to environments, but not with respect to procedures.

8 Interface Procedures

We have created three new *interface procedures* for creating and manipulating environments:

(*empty-env*) returns the empty environment, which contains no name/value bindings.

(*extend-env id arg old-env*) returns a new environment containing all of the name/value bindings of *old-env*, along with a new binding of *id* to *arg*.

(*apply-env env var*) returns the value of *var* within the environment *env*.

These interface procedures hide our implementation of environments from the *eval-exp* procedure. Our new interpreter uses only the three interface procedures to manipulate environments. Whenever we wish to change the implementation of environments, we change only the bodies of these interface procedures. We we never again touch any of the code within *eval-exp*, nor will we change the number or order of the arguments to our interface procedures.

9 Converting to a Tagged List Representation of Environments

Previously we defined three environment interface procedures that use a procedural representation of environments. We can now redefine these operators using a tagged list representation of environments, taking advantage of the powerful **kmatch** macro.

There are two variants of environments: the empty environment and extended environments. We created two interface procedures, *empty-env* and *extend-env*, that construct and return procedures representing these two types of environments. Such procedures are known as *constructors*.

The first thing we must do is rewrite our constructor procedures to construct and return tagged list representations of the two types of environments. The tagged list version of each constructor simply returns a list whose *car* is a symbol called the *tag*, and whose *cdr* is the list of arguments passed into the constructor procedure:

```
(define empty-env
  (λ ()
    '(empty-env)))

(define extend-env
  (λ (id arg env)
    '(extend-env ,id ,arg ,env)))
```

Note that the *empty-env* constructor does not take any arguments. Therefore, the list returned by *empty-env* contains only the tag symbol, *empty-env*[8].

We will use the tag symbols to distinguish between the two types of environments within the body of *apply-env*.

Now we only need to modify the *apply-env* interface procedure to use the new tagged list representation of environments. We can use our old friend, the **kmatch** macro, to help us:

```
(define apply-env
  (λ (env var)
    (kmatch env
      clause-1
      ---
      clause-N)))
```

[8]The first version of this document used *make-empty-env* and *make-extended-env* as the names of the constructor procedures, and *empty-env* and *extended-env* as the names of the tags associated with those procedures. Note that in the current version the tag names are indentical to the constructor names.

We will create exactly two **kmatch** clauses, corresponding to the tagged list representations of the two variants of environments:

```
(define apply-env
  (λ (env var)
    (kmatch env
      [() (empty-env) do-something]
      [(id arg env) (extend-env ,id ,arg ,env) do-something-else])))
```

What value should *apply-env* return when *env* matches against the tagged list (*empty-env*)? The value returned by the nested λ-expression in the original version of the *empty-env* procedure. Therefore, we can simply copy the body of the nested λ-expression in the original *empty-env*:

```
(define empty-env
  (λ ()
    (λ (var) ⊥)))
```

The body of the nested lambda expression is simply ⊥, which we copy into the right hand side of the *empty-env* clause of *apply-env*:

```
(define apply-env
  (λ (env var)
    (kmatch env
      [() (empty-env) ⊥]
      [(id arg env) (extend-env ,id ,arg ,env) do-something-else])))
```

We use the same technique with the (*extend-env* ,*id* ,*arg* ,*env*) clause. First we examine the original implementation of *extend-env*:

```
(define extend-env
  (λ (id arg env)
    (λ (var)
      (if (eq? id var)
          arg
          (apply-env env var)))))
```

The body of the nested λ-expression is the expression:

```
(if (eq? id var)
    arg
    (apply-env env var))
```

We copy this expression to the right hand side of the *extend-env* clause:

```
(define apply-env
  (λ (env var)
    (kmatch env
      [() (empty-env) ⊥]
      [(id arg env) (extend-env ,id ,arg ,env)
       (if (eq? id var)
           arg
           (apply-env env var))])))
```

Here, then, are our three new environment operators:

```
(define empty-env
  (λ ()
    `(empty-env)))

(define extend-env
  (λ (id arg env)
    `(extend-env ,id ,arg ,env)))

(define apply-env
  (λ (env var)
    (kmatch env
      [() (empty-env) ⊥]
      [(id arg env) (extend-env ,id ,arg ,env)
       (if (eq? id var)
           arg
           (apply-env env var))])))
```

Notice that we did not need to modify the code for *eval-exp* when we switched to a tagged list implementation of environments.

As before, we should run our test programs once again to ensure that we have not broken our interpreter.

10 Environments As Untagged Lists

Here is one final representation of environments, this time using untagged lists and **cond** instead of tagged lists and **kmatch**. In particular, environments are represented as lists of pairs, such as ((*a* . 4) (*b* . 3) (*d* . 9) (*a* . 7)). Such lists are known as *association lists*, *assoc lists*, or *a-lists*.

```
(define empty-env
  (λ ()
    '()))

(define extend-env
  (λ (id arg env)
    (cons (cons id arg) env)))

(define apply-env
  (λ (env var)
    (cond
      [(null? env) ⊥]
      [else
       (if (eq? (car (car env)) var)
           (cdr (car env))
           (apply-env (cdr env) var))])))
```

As always, we should run our test cases against the new version of our interpreter.

11 Rewriting `apply-env` Using `assq`

At the end of Professor Friedman's lecture, he mentioned that this most recent version of *apply-env*, which uses association lists to represent environments, can be rewritten to use the standard Scheme procedure *assq*. Let's try to understand what this means.

The Scheme procedure *assq* takes two arguments: a Scheme object *obj* and an association list *ls*. The *assq* procedure returns the first pair in *ls* whose *car* is equal[9] to *obj*, starting from the left hand side of the list. If *ls* does not contain a pair whose *car* is equal to *obj*, *assq* returns #f. Here are two examples showing the use of *assq* to look up the values associated with variables within an envrionment:

(*assq* 'a '((a . 4) (b . 3) (d . 9) (a . 7))) ⇒ (a . 4)
(*assq* 'f '((a . 4) (b . 3) (d . 9) (a . 7))) ⇒ #f

Note that the value of a variable x that is bound in the environment *env* is just (*cdr* (*assq* x *env*)).

We can use *assq* to simplify our implementation of *apply-env*. Here is our current version of *apply-env*:

```
(define apply-env
  (λ (env var)
    (cond
      [(null? env) ⊥]
      [else
       (if (eq? (car (car env)) var)
           (cdr (car env))
           (apply-env (cdr env) var))])))
```

which we can rewrite as:

```
(define apply-env
  (λ (env var)
    (cond
      [(null? env) ⊥]
      [(eq? (car (car env)) var) (cdr (car env))]
      [else (apply-env (cdr env) var)])))
```

Let's temporarily switch the order of arguments to *apply-env*, in order to make the transformation more obvious:

```
(define apply-env
  (λ (var env)
    (cond
      [(null? env) ⊥]
      [(eq? (car (car env)) var) (cdr (car env))]
      [else (apply-env var (cdr env))])))
```

We can now replace the last **cond** clause with two clauses, one of which uses *assq*:

```
(define apply-env
  (λ (var env)
    (cond
      [(null? env) ⊥]
      [(eq? (car (car env)) var) (cdr (car env))]
      [(assq var env) (cdr (assq var env))]
      [else ⊥])))
```

[9] That is, equal according to Scheme's *eq?* predicate.

We can then remove the first two clauses, which are now redundent, since *assq* performs the work of finding *var* within the association list *env*:

```
(define apply-env
  (λ (var env)
    (cond
      [(assq var env) (cdr (assq var env))]
      [else ⊥])))
```

We can simplify the clause that uses *assq* by using **cond**'s nifty arrow syntax:

```
(define apply-env
  (λ (var env)
    (cond
      [(assq var env) => (λ (p) (cdr p))]
      [else ⊥])))
```

Using a standard trick known as η-reduction, we can simplify the expression to the right of the arrow:

```
(define apply-env
  (λ (var env)
    (cond
      [(assq var env) => cdr]
      [else ⊥])))
```

Now that we are done with our transformation, we can swap back the order of the arguments to *apply-env*:

```
(define apply-env
  (λ (env var)
    (cond
      [(assq var env) => cdr]
      [else ⊥])))
```

Structurally, this final version of *apply-env* is quite different from the version we started with. In particular, our new implementation is no longer recursive.

Of course, being appropriately paranoid, we should once again test our modified interpreter.

We now have a simplified version of *apply-env*, and are finished learning about representation independence (for now).