The Scheme Environment: Dynamic Variables

William Clinger
Tektronix, Inc.

Scheme variables have *static scope* and *indefinite extent*. That is, the visibility (or scope) of a Scheme variable can be determined statically by looking at the lexical structure of a program, and the lifetime (or extent) of the variable is unbounded. The reason variables have indefinite extent in Scheme is that they may be remembered by procedures created as the result of evaluating a lambda expression.

Pascal variables have static scope and *dynamic extent*. Dynamic extent means that a Pascal variable's lifetime is limited to the execution of the block in which it is declared. Dynamic extent suffices for Pascal because procedures are not first class objects. In particular, Pascal procedures cannot be stored in global variables or data structures or returned as the result of a procedure call.

Traditional Lisp variables have had *indefinite scope* and dynamic extent. Indefinite scope means that you can't tell which variable references refer to which bindings by looking at the program. The only way to tell is to run the program and observe for each variable reference which of the still-active bindings for a variable was performed most recently. This combination of indefinite scope and dynamic extent is known as *dynamic binding*, and dynamically bound variables are variously known as dynamic, fluid, or special variables.

For example, if Scheme variables were dynamically bound and the `map` procedure were defined by

```
(define (map f l)
  (if (null? l)
      '()
      (cons (f (car l)) (map f (cdr l)))))
```

then the expression

```
(let ((l '(a b c)))
  (map (lambda (x) (cons x l))
       '(1 2 3 4)))
```

would yield ((1 1 2 3 4) (2 2 3 4) (3 3 4) (4 4)) as its result instead of the more intuitive result ((1 a b c) (2 a b c) (3 a b c) (4 a b c)). In traditional Lisp the only ways a programmer could prevent such an unfortunate coincidence of names were to use packages, obscure names, or a compiler that deviated from the interpreted semantics by treating variables as statically scoped. Such problems were apparently one of the historical motivations for the development of package systems.

Despite their problems, dynamic variables are occasionally useful because their value can be changed temporarily through binding rather than through an assignment that might be left undone through programmer oversight or a throw out of the continuation intended to undo the assignment. This use of dynamic variables can be implemented in Scheme by defining a macro, say `dynamic-let`, so that

```
    (dynamic-let ((I E))
      E0)
```

expands into

```
    (let ((savedI I)
          (before (lambda () (set! I E)))
          (body (lambda () E0)))
      (dynamic-wind before
                    body
                    (lambda () (set! I savedI))))
```

where dynamic-wind is the procedure described in my last article about continuations. The action of `dynamic-let` is to save the value of the variable `I`, set `I` to the value of `E`, execute the body `E0`, restore `I` to its original value, and return the value of the body `E0`. If there is a throw out of the body, then the `dynamic-wind` will make sure that the original value of `I` is still restored. This implementation of `dynamic-let` therefore provides the same kind of controlled assignment that dynamic variables provide, while improving upon dynamic variables by allowing the programmer to limit the visibility of the assignment using Scheme's static scope rules.

What happens if there is a throw into the body from outside the `dynamic-wind`? This can't happen in many dialects of Lisp, but it certainly can in Scheme. As written above, the `dynamic-wind` will set `I` to the result of re-evaluating the expression `E`. If we had wanted to use the result of the original evaluation of `E`, we could have written

```
    (let ((savedI I)
          (savedE E)
          (body (lambda () E0)))
      (dynamic-wind (lambda () (set! I savedE))
                    body
                    (lambda () (set! I savedI))))
```

Another property of both these implementations is that they always restore `I` to the value it had when the `dynamic-let` first began. If we throw out of the body, assign a new value to `I`, and then throw back in, might it make more sense to restore that new value on the way out than the old value? If we think so, then we can write

```
    (let ((savedI I)
          (savedE E)
          (body (lambda () E0)))
      (dynamic-wind (lambda () (set! savedI I) (set! I savedE))
                    body
                    (lambda () (set! I savedI))))
```

A fourth possibility is to mix this third implementation with the first. The four implementations offer four subtly different semantics for dynamic variables. All four regard dynamic variables as controlled assignments to a standard, statically bound variable and use only a single lexical environment. None of the four possible semantics is clearly better than the others.

Dynamic variables in Common Lisp are quite different. They live in a completely separate environment, the dynamic value environment, and the value of a variable in the dynamic value environment is always accessible through the `symbol-value` procedure even if the variable has a completely different binding in the lexical value environment. The dynamic value environment amounts to a global table, indexed by symbols, that is carried along with the current continuation and is adjusted whenever there is a throw out of the current continuation. Since it turns out in practice that the main use for dynamic variables in Scheme is in porting code written for dialects such as Common Lisp, we need to consider a semantics for dynamic variables that more closely mimics this idea of a global table indexed by symbols.

Since our implementations of dynamic variables as controlled assignments illustrated the implementation technique known as *shallow binding*, we will use the implementation technique known as *deep binding* for this next example. Deep binding is more efficient in a multi-tasking environment because the dynamic environment can be changed for a task switch (or throw) in constant time rather than time proportional to the number of dynamic variables.

We will represent the global table by an association list stored in a global variable. In a serious implementation we would of course make this variable local to the procedures that need it.

```
(define **dynamic-environment** '())
```

To bind a dynamic variable, we cons a pair consisting of its name and value onto the association list.

```
(define %internal-fluid-bind
  (lambda (symbol value)
    (set! **dynamic-environment**
          (cons (cons symbol value)
                **dynamic-environment**))))
```

Looking up the value of a dynamic variable is trivial. The lookup procedure shown here returns the pair so the value can be extracted using `cdr` or assigned using `set-cdr!`.

```
(define %internal-fluid-lookup
  (lambda (symbol)
    (let ((temp (assq symbol **dynamic-environment**)))
      (if temp
          temp
          (cerror "Unbound fluid variable" symbol)))))
```

The interesting part is what happens when we do a throw. The continuation we are throwing to will have to restore the dynamic environment to the value it had when the continuation was created. We can arrange that by redefining `call-with-current-continuation`:

```
(let ((original-call/cc call-with-current-continuation))
  (set! call-with-current-continuation
        (lambda (f)
          (original-call/cc
           (lambda (k)
             (letrec ((saved-dynamic-env **dynamic-environment**)
                      (new-k (lambda (v)
                               (set! **dynamic-environment**
                                     saved-dynamic-env)
                               (k v))))
               (f new-k)))))))
```

This new definition works just like the old except that the procedure it creates remembers the dynamic environment in effect when it was created and restores that dynamic environment when it is called.

The technique of redefining `call-with-current-continuation` is worth studying, since it is the same technique needed to implement `dynamic-wind`. Note that we are assuming that the only way to do a throw is to call a procedure created by `call-with-current-continuation` and that the system we are loading this code into contains no such procedures. If these assumptions are not true, then we simply have more cases to take care of.

We have now defined the low-level table maintenance procedures. It is a simple matter to design more palatable macro syntaxes for using them. We can for example define `fluid-let` so that

```
(fluid-let ((I1 E1) ...) E ...)
```

expands into something like

```
(call-with-current-continuation
 (lambda (k)
  (%internal-fluid-bind 'I1 E1)
  ...
  (k (begin E ...))))
```

except that we must arrange for all the names to be resolved without clashes.

When porting code, we must insert the special syntaxes wherever dynamic variables are used. This poses a problem: Local inspection cannot determine that a Common Lisp variable is static rather than dynamic. Despite the suggested coding practice that all dynamic variables be flagged by asterisks in their names, the only way to be sure is to scan the entire Common Lisp program for `defvar` and `special` proclamations. That this nuisance is common to all Common Lisp program maintenance is but small comfort to the person who has to do it.

In traditional Lisp with dynamic variables, hardly any procedures contain tail-recursive calls because a new continuation needs to be created to remove the procedure's arguments from the dynamic environment. It is possible, however, to imagine a semantics for dynamic variables such that procedure calls in tail-recursive positions are evaluated as follows: The operator and operand positions are evaluated, any necessary adjustments to the dynamic environment are performed, and only then is the call performed. This would make calls that appear to be tail-recursive actually be tail-recursive, at the cost of a still different, not necessarily better or worse, semantics. The people I've talked to about this seem to think it pretty strange, though.

I hope that the six alternative semantics for dynamic variables I have described here give you some feeling for the complexity of programming language design. By no means have we considered all the possibilities.

* * *

I wish to thank Norman Adams for his help with this article.