12

Generalized Variables

Chapter 8 mentioned that one of the advantages of macros is their ability to transform their arguments. One macro of this sort is `setf`. This chapter looks at the implications of `setf`, and then shows some examples of macros which can be built upon it.

Writing correct macros on `setf` is surprisingly difficult. To introduce the topic, the first section will provide a simple example which is slightly incorrect. The next section will explain what's wrong with this macro, and show how to fix it. The third and fourth sections present examples of utilities built on `setf`, and the final section explains how to define your own `setf` inversions.

12.1 The Concept

The built-in macro `setf` is a generalization of `setq`. The first argument to `setf` can be a call instead of just a variable:

```
> (setq lst '(a b c))
(A B C)
> (setf (car lst) 480)
480
> lst
(480 B C)
```

In general (`setf` *x y*) can be understood as saying "see to it that *x* evaluates to *y*." As a macro, `setf` can look inside its arguments to see what needs to be done to make such a statement true. If the first argument (after macroexpansion) is a

symbol, the `setf` just expands into a `setq`. But if the first argument is a query, the `setf` expands into the corresponding assertion. Since the second argument is a constant, the preceding example could expand into:

```
(progn (rplaca lst 480) 480)
```

This transformation from query to assertion is called *inversion*. All the most frequently used Common Lisp access functions have predefined inversions, including `car`, `cdr`, `nth`, `aref`, `get`, `gethash`, and the access functions created by `defstruct`. (The full list is in CLTL2, p. 125.)

An expression which can serve as the first argument to `setf` is called a *generalized variable*. Generalized variables have turned out to be a powerful abstraction. A macro call resembles a generalized variable in that any macro call which expands into an invertible reference will itself be invertible.

When we also write our own macros on top of `setf`, the combination leads to noticeably cleaner programs. One of the macros we can define on top of `setf` is `toggle`,[1]

```
(defmacro toggle (obj)                                  ; wrong
  `(setf ,obj (not ,obj)))
```

which toggles the value of a generalized variable:

```
> (let ((lst '(a b c)))
    (toggle (car lst))
    lst)
(NIL B C)
```

Now consider the following sample application. Suppose someone—a soap-opera writer, energetic busybody, or party official—wants to maintain a database of all the relations between the inhabitants of a small town. Among the tables required is one which records people's friends:

```
(defvar *friends* (make-hash-table))
```

The entries in this hash-table are themselves hash-tables, in which names of potential friends are mapped to `t` or `nil`:

```
(setf (gethash 'mary *friends*) (make-hash-table))
```

To make John the friend of Mary, we would say:

```
(setf (gethash 'john (gethash 'mary *friends*)) t)
```

[1] This definition is not correct, as the following section will explain.

The town is divided between two factions. As factions are wont to do, each says "anyone who is not with us is against us," so everyone in town has been compelled to join one side or the other. Thus when someone switches sides, all his friends become enemies and all his enemies become friends.

To toggle whether `x` is the friend of `y` using only built-in operators, we have to say:

```
(setf (gethash x (gethash y *friends*))
      (not (gethash x (gethash y *friends*))))
```

which is a rather complicated expression, though much simpler than it would have been without `setf`. If we had defined an access macro upon the database as follows:

```
(defmacro friend-of (p q)
  `(gethash ,p (gethash ,q *friends*)))
```

then between this macro and `toggle`, we would have been better equipped to deal with changes to the database. The previous update could have been expressed as simply:

```
(toggle (friend-of x y))
```

Generalized variables are like a health food that tastes good. They yield programs which are virtuously modular, and yet beautifully elegant. If you provide access to your data structures through macros or invertible functions, other modules can use `setf` to modify your data structures without having to know the details of their representation.

12.2 The Multiple Evaluation Problem

The previous section warned that our initial definition of `toggle` was incorrect:

```
(defmacro toggle (obj)                                  ; wrong
  `(setf ,obj (not ,obj)))
```

It is subject to the problem described in Section 10.1, multiple evaluation. Trouble arises when its argument has side-effects. For example, if `lst` is a list of objects, and we write:

```
(toggle (nth (incf i) lst))
```

then we would expect to be toggling the (`i`+1)th element. However, with the current definition of `toggle` this call will expand into:

```
(setf (nth (incf i) lst)
      (not (nth (incf i) lst)))
```

This increments `i` twice, and sets the (`i`+1)th element to the opposite of the (`i`+2)th element. So in this example

```
> (let ((lst '(t nil t))
        (i -1))
    (toggle (nth (incf i) lst))
    lst)
(T NIL T)
```

the call to `toggle` seems to have no effect.

It is not enough just to take the expression given as an argument to `toggle` and insert it as the first argument to `setf`. We have to look inside the expression to see what it does: if it contains subforms, we have to break them apart and evaluate them separately, in case they have side effects. In general, this is a complicated business.

To make it easier, Common Lisp provides a macro which automatically defines a limited class of macros on `setf`. This macro is called `define-modify-macro`, and it takes three arguments: the name of the macro, its additional parameters (after the generalized variable), and the name of the function[2] which yields the new value for the generalized variable.

Using `define-modify-macro`, we could define `toggle` as follows:

```
(define-modify-macro toggle () not)
```

Paraphrased, this says "to evaluate an expression of the form (`toggle` *place*), find the location specified by *place*, and if the value stored there is *val*, replace it with the value of (`not` *val*)." Here is the new macro used in the same example:

```
> (let ((lst '(t nil t))
        (i -1))
    (toggle (nth (incf i) lst))
    lst)
(NIL NIL T)
```

This version gives the correct result, but it could be made more general. Since `setf` and `setq` can take an arbitrary number of arguments, so should `toggle`. We can add this capability by defining another macro on top of the modify-macro, as in Figure 12.1.

[2] A function name in the general sense: either `1+` or `(lambda (x) (+ x 1))`.

```
(defmacro allf (val &rest args)
  (with-gensyms (gval)
    `(let ((,gval ,val))
       (setf ,@(mapcan #'(lambda (a) (list a gval))
                       args)))))

(defmacro nilf (&rest args) `(allf nil ,@args))

(defmacro tf (&rest args) `(allf t ,@args))

(defmacro toggle (&rest args)
  `(progn
     ,@(mapcar #'(lambda (a) `(toggle2 ,a))
               args)))

(define-modify-macro toggle2 () not)
```

Figure 12.1: Macros which operate on generalized variables.

12.3 New Utilities

This section gives some examples of new utilities which operate on generalized variables. They must be macros in order to pass their arguments intact to `setf`.

Figure 12.1 shows four new macros built upon `setf`. The first, `allf`, is for setting a number of generalized variables to the same value. Upon it are built `nilf` and `tf`, which set their arguments to `nil` and `t`, respectively. These macros are simple, but they make a difference.

Like `setq`, `setf` can take multiple arguments—alternating variables and values:

```
(setf x 1 y 2)
```

So can these new utilities, but you can skip giving half the arguments. If you want to initialize a number of variables to `nil`, instead of

```
(setf x nil y nil z nil)
```

you can say just

```
(nilf x y z)
```

```
(define-modify-macro concf (obj) nconc)

(define-modify-macro conc1f (obj)
  (lambda (place obj)
    (nconc place (list obj))))

(define-modify-macro concnew (obj &rest args)
  (lambda (place obj &rest args)
    (unless (apply #'member obj place args)
      (nconc place (list obj)))))
```

Figure 12.2: List operations on generalized variables.

The last macro, `toggle`, was described in the previous section: it is like `nilf`, but gives each of its arguments the opposite truth value.

These four macros illustrate an important point about operators for assignment. Even if we only intend to use an operator on ordinary variables, it's worth writing it to expand into a `setf` instead of a `setq`. If the first argument is a symbol, the `setf` will expand into a `setq` anyway. Since we can have the generality of `setf` at no extra cost, it is rarely desirable to use `setq` in a macroexpansion.

Figure 12.2 contains three macros for destructively modifying the ends of lists. Section 3.1 mentioned that it is unsafe to rely on

```
(nconc x y)
```

for side-effects, and that one must write instead

```
(setq x (nconc x y))
```

This idiom is embodied in `concf`. The more specialized `conc1f` and `concnew` are like `push` and `pushnew` for the other end of the list: `conc1f` adds one element to the end of a list, and `concnew` does the same, but only if the element is not already a member.

Section 2.2 mentioned that the name of a function can be a lambda-expression as well as a symbol. Thus it is fine to give a whole lambda-expression as the third argument to `define-modify-macro`, as in the definition of `conc1f`. Using `conc1` from page 45, this macro could also have been written:

```
(define-modify-macro conc1f (obj) conc1)
```

The macros in Figure 12.2 should be used with one reservation. If you're planning to build a list by adding elements to the end, it may be preferable to use

push, and then nreverse the list. It is cheaper to do something to the front of a list than to the end, because to do something to the end you have to get there first. It is probably to encourage efficient programming that Common Lisp has many operators for the former and few for the latter.

12.4 More Complex Utilities

Not all macros on setf can be defined with define-modify-macro. Suppose, for example, that we want to define a macro _f for applying a function destructively to a generalized variable. The built-in macro incf is an abbreviation for setf of +. Instead of

```
(setf x (+ x y))
```

we say just

```
(incf x y)
```

The new _f is to be a generalization of this idea: while incf expands into a call to +, _f will expand into a call to the operator given as the first argument. For example, in the definition of scale-objs on page 115, we had to write

```
(setf (obj-dx o) (* (obj-dx o) factor))
```

With _f this will become

```
(_f * (obj-dx o) factor)
```

The incorrect way to write _f would be:

```
(defmacro _f (op place &rest args)                    ; wrong
  `(setf ,place (,op ,place ,@args)))
```

Unfortunately, we can't define a correct _f with define-modify-macro, because the operator to be applied to the generalized variable is given as an argument.

More complex macros like this one have to be written by hand. To make such macros easier to write, Common Lisp provides the function get-setf-method, which takes a generalized variable and returns all the information necessary to retrieve or set its value. We will see how to use this information by hand-generating an expansion for:

```
(incf (aref a (incf i)))
```

When we call get-setf-method on the generalized variable, we get five values intended for use as ingredients in the macroexpansion:

```
> (get-setf-method '(aref a (incf i)))
(#:G4 #:G5)
(A (INCF I))
(#:G6)
(SYSTEM:SET-AREF #:G6 #:G4 #:G5)
(AREF #:G4 #:G5)
```

The first two values are lists of temporary variables and the values that should be assigned to them. So we can begin the expansion with:

```
(let* ((#:g4 a)
       (#:g5 (incf i)))
  ...)
```

These bindings should be created in a `let*` because in the general case the value forms can refer to earlier variables. The third[3] and fifth values are another temporary variable and the form that will return the original value of the generalized variable. Since we want to add `1` to this value, we wrap the latter in a call to `1+`:

```
(let* ((#:g4 a)
       (#:g5 (incf i))
       (#:g6 (1+ (aref #:g4 #:g5))))
  ...)
```

Finally, the fourth value returned by `get-setf-method` is the assignment that must be made within the scope of the new bindings:

```
(let* ((#:g4 a)
       (#:g5 (incf i))
       (#:g6 (1+ (aref #:g4 #:g5))))
  (system:set-aref #:g6 #:g4 #:g5))
```

More often than not, this form will refer to internal functions which are not part of Common Lisp. Usually `setf` masks the presence of these functions, but they have to exist somewhere. Everything about them is implementation-dependent, so portable code should use forms returned by `get-setf-method`, rather than referring directly to functions like `system:set-aref`.

Now to implement `_f` we write a macro which does almost exactly what we did when expanding `incf` by hand. The only difference is that, instead of wrapping the last form in the `let*` in a call to `1+`, we wrap it in an expression made from the arguments to `_f`. The definition of `_f` is shown in Figure 12.3.

[3] The third value is currently always a list of one element. It is returned as a list to provide the (so far unconsummated) potential to store multiple values in generalized variables.

```
(defmacro _f (op place &rest args)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-method place)
    `(let* (,@(mapcar #'list vars forms)
            (,(car var) (,op ,access ,@args)))
       ,set)))

(defmacro pull (obj place &rest args)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-method place)
    (let ((g (gensym)))
      `(let* ((,g ,obj)
              ,@(mapcar #'list vars forms)
              (,(car var) (delete ,g ,access ,@args)))
         ,set))))

(defmacro pull-if (test place &rest args)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-method place)
    (let ((g (gensym)))
      `(let* ((,g ,test)
              ,@(mapcar #'list vars forms)
              (,(car var) (delete-if ,g ,access ,@args)))
         ,set))))

(defmacro popn (n place)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-method place)
    (with-gensyms (gn glst)
      `(let* ((,gn ,n)
              ,@(mapcar #'list vars forms)
              (,glst ,access)
              (,(car var) (nthcdr ,gn ,glst)))
         (prog1 (subseq ,glst 0 ,gn)
                ,set)))))
```

Figure 12.3: More complex macros on `setf`.

This utility is quite a useful one. Now that we have it, for example, we can easily replace any named function with a memoized (Section 5.3) equivalent. [4] To memoize `foo` we would say:

```
(_f memoize (symbol-function 'foo))
```

Having `_f` also makes it easy to define other macros on `setf`. For example, we could now define `conc1f` (Figure 12.2) as:

```
(defmacro conc1f (lst obj)
  `(_f nconc ,lst (list ,obj)))
```

Figure 12.3 contains some other useful macros on `setf`. The next, `pull`, is intended as a complement to the built-in `pushnew`. The pair are like more discerning versions of `push` and `pop`; `pushnew` pushes a new element onto a list if it is not already a member, and `pull` destructively removes selected elements from a list. The `&rest` parameter in `pull`'s definition makes `pull` able to accept all the same keyword parameters as `delete`:

```
> (setq x '(1 2 (a b) 3))
(1 2 (A B) 3)
> (pull 2 x)
(1 (A B) 3)
> (pull '(a b) x :test #'equal)
(1 3)
> x
(1 3)
```

You could almost think of this macro as if it were defined:

```
(defmacro pull (obj seq &rest args)                    ; wrong
  `(setf ,seq (delete ,obj ,seq ,@args)))
```

though if it really were defined that way, it would be subject to problems with both order and number of evaluations. We could define a version of `pull` as a simple modify-macro:

```
(define-modify-macro pull (obj &rest args)
  (lambda (seq obj &rest args)
    (apply #'delete obj seq args)))
```

[4] Built-in functions should not be memoized in this way, though. Common Lisp forbids the redefinition of built-in functions.

but since modify-macros must take the generalized variable as their first argument, we would have to give the first two arguments in reverse order, which would be less intuitive.

The more general `pull-if` takes an initial function argument, and expands into a `delete-if` instead of a `delete`:

```
> (let ((lst '(1 2 3 4 5 6)))
    (pull-if #'oddp lst)
    lst)
(2 4 6)
```

These two macros illustrate another general point. If the underlying function takes optional arguments, so should the macro built upon it. Both `pull` and `pull-if` pass optional arguments on to their `delete`s.

The final macro in Figure 12.3, `popn`, is a generalization of `pop`. Instead of popping just one element of a list, it pops and returns a subsequence of arbitrary length:

```
> (setq x '(a b c d e f))
(A B C D E F)
> (popn 3 x)
(A B C)
> x
(D E F)
```

Figure 12.4 contains a macro which sorts its arguments. If `x` and `y` are variables and we want to ensure that `x` does not have the lower of the two values, we can write:

```
(if (> y x) (rotatef x y))
```

But if we want to do this for three or more variables, the code required grows rapidly. Instead of writing it by hand, we can have `sortf` write it for us. This macro takes a comparison operator plus any number of generalized variables, and swaps their values until they are in the order dictated by the operator. In the simplest case, the arguments could be ordinary variables:

```
> (setq x 1 y 2 z 3)
3
> (sortf > x y z)
3
> (list x y z)
(3 2 1)
```

```
(defmacro sortf (op &rest places)
  (let* ((meths (mapcar #'(lambda (p)
                            (multiple-value-list
                              (get-setf-method p)))
                        places))
         (temps (apply #'append (mapcar #'third meths))))
    `(let* ,(mapcar #'list
                    (mapcan #'(lambda (m)
                                (append (first m)
                                        (third m)))
                            meths)
                    (mapcan #'(lambda (m)
                                (append (second m)
                                        (list (fifth m))))
                            meths))
       ,@(mapcon #'(lambda (rest)
                     (mapcar
                       #'(lambda (arg)
                           `(unless (,op ,(car rest) ,arg)
                              (rotatef ,(car rest) ,arg)))
                       (cdr rest)))
                 temps)
       ,@(mapcar #'fourth meths))))
```

Figure 12.4: A macro which sorts its arguments.

In general, they could be any invertible expressions. Suppose `cake` is an invertible function which returns someone's piece of cake, and `bigger` is a comparison function defined on pieces of cake. If we want to enforce the rule that the `cake` of `moe` is no less than the `cake` of `larry`, which is no less than that of `curly`, we write:

```
(sortf bigger (cake 'moe) (cake 'larry) (cake 'curly))
```

The definition of `sortf` is similar in outline to that of `_f`. It begins with a `let*` in which the temporary variables returned by `get-setf-method` are bound to the initial values of the generalized variables. The core of `sortf` is the central `mapcon` expression, which generates code to sort these temporary variables. The code generated by this portion of the macro grows exponentially with the number of arguments. After sorting, the generalized variables are reassigned using the

```
(sortf > x (aref ar (incf i)) (car lst))
```

expands (in one possible implementation) into:

```
(let* ((#:g1 x)
       (#:g4 ar)
       (#:g3 (incf i))
       (#:g2 (aref #:g4 #:g3))
       (#:g6 lst)
       (#:g5 (car #:g6)))
  (unless (> #:g1 #:g2)
    (rotatef #:g1 #:g2))
  (unless (> #:g1 #:g5)
    (rotatef #:g1 #:g5))
  (unless (> #:g2 #:g5)
    (rotatef #:g2 #:g5))
  (setq x #:g1)
  (system:set-aref #:g2 #:g4 #:g3)
  (system:set-car #:g6 #:g5))
```

Figure 12.5: Expansion of a call to `sortf`.

forms returned by `get-setf-method`. The algorithm used is the $O(n^2)$ bubble-sort, but this macro is not intended to be called with huge numbers of arguments.

Figure 12.5 shows the expansion of a call to `sortf`. In the initial `let*`, the arguments and their subforms are carefully evaluated in left-to-right order. Then appear three expressions which compare and possibly swap the values of the temporary variables: the first is compared to the second, then the first to the third, then the second to the third. Finally the the generalized variables are reassigned left-to-right. Although the issue rarely arises, macro arguments should usually be assigned left-to-right, as well as being evaluated in this order.

Operators like `_f` and `sortf` bear a certain resemblance to functions that take functional arguments. It should be understood that they are something quite different. A function like `find-if` takes a function and calls it; a macro like `_f` takes a *name*, and makes it the car of an expression. Both `_f` and `sortf` could have been written to take functional arguments. For example, `_f` could have been written:

```
(defmacro _f (op place &rest args)
  (let ((g (gensym)))
    (multiple-value-bind (vars forms var set access)
                         (get-setf-method place)
    `(let* ((,g ,op)
            ,@(mapcar #'list vars forms)
            (,(car var) (funcall ,g ,access ,@args)))
       ,set))))
```

and called `(_f #'+ x 1)`. But the original version of `_f` can do anything this one could, and since it deals in names, it can also take the name of a macro or special form. As well as `+`, you could call, for example, `nif` (page 150):

```
> (let ((x 2))
    (_f nif x 'p 'z 'n)
    x)
P
```

12.5 Defining Inversions

Section 12.1 explained that any macro call which expands into an invertible reference will itself be invertible. You don't have to define operators as macros just to make them invertible, though. By using `defsetf` you can tell Lisp how to invert any function or macro call.

This macro can be used in two ways. In the simplest case, its arguments are two symbols:

```
(defsetf symbol-value set)
```

In the more complicated form, a call to `defsetf` is like a call to `defmacro`, with an additional parameter for the updated value form. For example, this would define a possible inversion for `car`:

```
(defsetf car (lst) (new-car)
  `(progn (rplaca ,lst ,new-car)
          ,new-car))
```

There is one important difference between `defmacro` and `defsetf`: the latter automatically creates gensyms for its arguments. With the definition given above, `(setf (car x) y)` would expand into:

```
(let* ((#:g2 x)
       (#:g1 y))
  (progn (rplaca #:g2 #:g1)
         #:g1))
```

```
(defvar *cache* (make-hash-table))

(defun retrieve (key)
  (multiple-value-bind (x y) (gethash key *cache*)
    (if y
        (values x y)
        (cdr (assoc key *world*)))))

(defsetf retrieve (key) (val)
  `(setf (gethash ,key *cache*) ,val))
```

Figure 12.6: An asymmetric inversion.

Thus we can write `defsetf` expanders without having to worry about variable capture, or number or order of evaluations.

In CLTL2 Common Lisp, it is possible to define `setf` inversions directly with `defun`, so the previous example could also be written:

```
(defun (setf car) (new-car lst)
  (rplaca lst new-car)
  new-car)
```

The updated value should be the first parameter in such a function. It is also conventional to return this value as the value of the function.

The examples so far have suggested that generalized variables are supposed to refer to a place in a data structure. The villain carries his hostage down to the dungeon, and the rescuing hero carries her back up again; they both follow the same path, but in different directions. It's not surprising if people have the impression that `setf` must work this way, because all the predefined inversions seem to be of this form; indeed, `place` is the conventional name for a parameter which is to be inverted.

In principle, `setf` is more general: an access form and its inversion need not even operate on the same data structure. Suppose that in some application we want to cache database updates. This could be necessary, for example, if it were not efficient to do real updates on the fly, or if all the updates had to be verified for consistency before committing to them.

Suppose that `*world*` is the actual database. For simplicity, we will make it an assoc-list whose elements are of the form (*key* . *val*). Figure 12.6 shows a lookup function called `retrieve`. If `*world*` is

```
((a . 2) (b . 16) (c . 50) (d . 20) (f . 12))
```

then

```
> (retrieve 'c)
50
```

Unlike a call to `car`, a call to `retrieve` does not refer to a specific place in a data structure. The return value could come from one of two places. And the inversion of retrieve, also defined in Figure 12.6, only refers to one of them:

```
> (setf (retrieve 'n) 77)
77
> (retrieve 'n)
77
T
```

This lookup returns a second value of `t`, indicating that the answer was found in the cache.

Like macros themselves, generalized variables are an abstraction of remarkable power. There is probably more to be discovered here. Certainly individual users are likely to discover ways in which the use of generalized variables could lead to more elegant or more powerful programs. But it may also be possible to use `setf` inversion in new ways, or to discover other classes of similarly useful transformations. ○