The Scheme of Things

Pavel Curtis

Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Pavel@Xerox.Com

I decided to begin my tenure on this column by describing two recent proposals for additions to Scheme. The first is a facility for creating new, programmer-defined data types; the second makes it possible for procedures to return more than one value. These proposals will be considered by the Scheme authors for inclusion in the *Revised*^{5} *Report on Scheme* whenever that document comes up for discussion.

For future columns, I am considering discussing other possible directions for Scheme language development, including macros, dynamic binding, exception handling, modules, and concurrency. I may also do a column or two on Scheme compilation and interpretation techniques. If you've got an idea for some other topic you'd like me to discuss here, please feel free to write me at the addresses above; your suggestions are always welcome.

One of the most glaring holes in the Scheme language is the lack of a facility for defining new data types in a modular way. Many portable Scheme programs have little prologues that look something like this:

```
(define (make-frob a b)
  (vector a b))
(define (frob-a frob)
  (vector-ref frob 0))
(define (frob-b frob)
  (vector-ref frob 1))

(define (frob-set-a! frob obj)
  (vector-set! frob 0 obj))
(define (frob-set-b! frob obj)
  (vector-set! frob 1 obj))
```

This sort of programming manages to do the job well enough for many purposes. For larger programs, though, there are several problems with this approach to defining new types of data:

- There is no way to test whether or not a given value is a "frob".
- Every frob looks like a vector.
- For each new field in such a structure, there are three places to be maintained consistently.
- Defining new data types this way is tedious.

The first problem could be addressed by making every frob vector one element longer and putting some unique, distinguished value in that slot. We might modify the code above to do this as follows:

```
(define frob-marker (list "Frob"))
```

```
(define (make-frob a b)
  (vector frob-marker a b))
```

(We must, of course, also add one to each of the vector indices in the rest of the code.) Given this, we could define **frob?** as follows:

```
(define (frob? obj)
  (and (vector? obj)
       (> (vector-length obj) 0)
       (eqv? frob-marker
             (vector-ref obj 0))))
```

The second problem, that every frob looks like a vector, has several consequences.

All code that might manipulate frobs must be written carefully so as not to mistake them for vectors. For example, suppose that foo is a procedure that takes either frobs or vectors as inputs. It must be written like this:

```
(define (foo obj)
  (if (frob? obj)
    (do-frob-stuff obj)
    (do-vector-stuff obj)))
```

and not like this:

```
(define (foo obj)
  (if (vector? obj)
    (do-vector-stuff obj)
    (do-frob-stuff obj)))
```

In essence, the programmer must always keep in mind that frobs are implemented as vectors, a serious breach of modularity. If there is a bug of this variety, it may well be very difficult to notice, let alone find.

Since frobs cannot be distinguished from vectors, they will be printed as such by the standard procedures **write** and **display**. This is almost certainly not the most readable external representation one could choose.

The other two problems I listed above (that this approach is tedious and error-prone) could presumably be solved by means of some higher-level syntax or procedures. The "records" proposal, originally formulated by Jonathan Rees based upon a suggestion by Norman Adams, is primarily aimed at solving the second problem: modularity.

Under the proposal, five new standard procedures will be added to Scheme:

```
make-record-type
record-constructor
record-predicate
record-accessor
record-updater
```

Before I describe the new procedures, let me show you how the "frob" example above would be rewritten using them:

```
(define frob-type
  (make-record-type "Frob"
                    '(a b)))

(define make-frob
  (record-constructor frob-type
                      '(a b)))

(define frob?
  (record-predicate frob-type))

(define frob-a
  (record-accessor frob-type 'a))
(define frob-b
  (record-accessor frob-type 'b))

(define frob-set-a!
  (record-updater frob-type 'a))
(define frob-set-b!
  (record-updater frob-type 'b))
```

The procedure **make-record-type** takes two arguments, a string giving a name for the new type (for use in printing the values and other such debugging purposes) and a list of symbols naming the fields to appear in each value of the new type. **Make-record-type** returns a *record-type descriptor* (abbreviated "rtd"), a value that can be passed to the other four procedures to identify the new type.

It is guaranteed that each rtd returned by **make-record-type** really identifies a brand new type, disjoint from all others. Values of the new type are not vectors, pairs, strings, etc.

Record-constructor takes a record-type descriptor, *rtd*, and a list of symbols, *fields*, all of which must name fields in the type identified by *rtd*. It returns a procedure for constructing values of the new type. This *constructor* takes as many arguments as there were symbols in *fields*; it returns a new value whose fields are initialized to the given arguments. Thus, for example, with `make-frob` defined as shown above, `(make-frob 1 2)` returns a new frob object whose `a` and `b` fields are initialized to `1` and `2`, respectively.

The *fields* argument to `record-constructor` is optional; it defaults to the list of field names given when *rtd* was created. Thus, the definition of `make-frob` above could have been simplified to

```
(define make-frob
  (record-constructor frob-type))
```

`Record-predicate` takes just one argument, a record-type descriptor *rtd*; it returns a membership-testing procedure for the type identified by *rtd*. The returned procedure takes one argument and returns either `#t`, if the argument is a member of the type identified by *rtd*, or `#f`, otherwise.

The procedure `record-accessor` takes two arguments, a record-type descriptor, *rtd*, and a symbol, *field*, naming one of the fields in each value of the type identified by *rtd*. It returns a procedure that takes one argument, which must be a member of the type identified by *rtd*. The returned procedure itself returns the value of the field named *field* in the argument.

Finally, `record-updater` takes the same arguments as `record-accessor`, but returns a different kind of procedure. This procedure takes two arguments, a member of the type identified by *rtd*, call it *record*, and a new value for the field named *field* in *record*; it stores the new value in the field and returns an unspecified value.

Given this proposal, it would be legitimate for a Scheme system to implement the basic operations on pairs as follows:

```
(define pair-type
  (make-record-type "Cons cells"
                    '(car cdr)))

(define cons
  (record-constructor pair-type))

(define pair?
  (record-predicate pair-type))

(define car
  (record-accessor pair-type 'car))
(define set-car!
  (record-updater pair-type 'car))

(define cdr
  (record-accessor pair-type 'cdr))
(define set-cdr!
  (record-updater pair-type 'cdr))
```

Several suggestions have been made for extensions to the records proposal. The most controversial of these is the addition of what Jonathan Rees has called an "abstraction-breaking" procedure, `record-type-descriptor`, that maps any record to an rtd identifying its type. Thus, for example, the expression

```
(let ((a-frob (make-frob 1 2)))
  (record-type-descriptor a-frob))
```

would return an rtd equivalent to `frob-type`, above; that is, the new rtd could be passed to `record-predicate`, for example, to obtain a procedure with precisely the same behavior as `frob?`.

Given this, along with the procedure `record-type-field-names` mapping rtds to their associated list of field names, it is possible to read or write the values of any field in any record, regardless of modularity concerns.

The presence of such procedures in the language, some people claim, would remove the very advantage that the record proposal was meant to provide: modularity. It is likely that this issue will be actively discussed at the next meeting of the Scheme authors.

Under another suggested extension. record types would allow a simple kind of subtyping, not unlike the inheritance mechanisms in most object-oriented languages. This idea is to have `make-record-type` take an optional third argument, a "parent" rtd:

```
(define foo-type
  (make-record-type "Foo"
                    '(c)
                    frob-type))

(define make-foo
  (record-constructor foo-type
                      '(a b c)))

(define foo?
  (record-predicate foo-type))

(define foo-c
  (record-accessor frob-type 'c))
```

Any foo would also be a frob; thus, for example, `(frob? (make-foo 1 2 3))` would yield `#t` and `frob-a` would accept a foo and return its a field.

There has been a cautious reaction to the subtyping suggestion; some folks are afraid that the behavior specified might not be compatible with whatever is eventually decided about object-oriented programming in Scheme.

A final suggestion asked that `make-record-type` take *another* optional argument, a procedure to be used for printing values of the new type. The reaction to this idea has been like the previous case: this is like specifying a method in object-oriented languages, so perhaps we should avoid committing to such an interface before we've decided what we want to do about the more general issue.

The records proposal solves three of the four problems found with the *ad hoc* implementation of frobs at the beginning of the column: there is a way to test whether or not a given object is a frob, frobs do not look like vectors (or anything else but frobs), and the addition of a new field does not require that any subtle consistencies be maintained.

It would be more difficult to argue that the proposal alleviates the fourth problem, that such record-type definitions are tedious to write and read. Given a macro-definition facility, though, one could attack this problem by designing some more convenient syntax for type definitions that expands into uses of the procedures described above. For example, an implementation might introduce a form similar to the `defstruct` macro in Common Lisp.

The records proposal does not include such a tedium-saving syntactic extension because it was felt that more experimentation was needed before adding one to the language. If and when the rest of the proposal is accepted, it will be possible for such experiments to be carried out portably.

Suppose that you are designing an implementation of hash-tables in which `table-ref` is the lookup procedure. You must decide what `(table-ref` *table key*`)` is to return. An obvious approach is to return the value associated with *key* in *table*, if there is one, or `#f` otherwise. This simple strategy leads to ambiguity, however. Whenever `table-ref` returns `#f`, one knows that either `#f` is the value associated with *key* or *key* has *no* associated value; there is no way to distinguish the two cases.

Of course, there are several ways to solve this particular problem. For example, one could specify at table creation time some distinguished value to be returned in the "no associated value" case; each client of hash-tables could then pick a value appropriate to their application and check for it whenever `table-ref` was used. Alternatively, `table-ref` could take a procedure to be called when there's no associated value. The possibilities are endless.

A perhaps more general solution would be for `table-ref` somehow to return *two* values, a boolean indicating whether or not *key* has an associated value and the value itself.

I say that this solution is more general because the ability for procedures to return more than one value is useful in a wide variety of situations. For example:

- The computation of some value involves the simultaneous computation of one or more other, useful values. For example, when computing the quotient of two integers, the remainder is also computed; it might be useful for a division procedure to return both quotient and remainder. Many parsing or matching operations also have this property.

- In inductions (that is, loops or recursions) over more than one variable, a part of the computation is encapsulated in a procedure that takes the old values of the induction variables and must return new values for them. For example, consider a tree walk that simultaneously computes maximum tree depth and the number of leaves.

- Some functions inherently compute two or more values. For example, matrix factoring usually involves computing two new matrices from the input matrix.

In all of these cases, one could, of course, return a single value that contains all of the desired values, such as a list. In addition to being inefficient, though, this has conceptual problems. It could be argued that values in programs should represent conceptual wholes; in many cases, the collection of values returned by some procedure lack this coherence.

With this and other considerations in mind, several Scheme authors collaborated to put forward a proposal to allow multiple-result procedures. It calls for the addition of two standard procedures:

```
values
call-with-values
```

The first of these is the basic means for returning multiple values and the second is that for using them.

The procedure **values** takes an arbitrary number of arguments, including none, and returns all of the arguments as its results. For example, the expression (**values** 1 2) returns 1 and 2. To say this in another way, **values** invokes its own continuation on the arguments passed to it. In fact, **values** can be written in standard Scheme:

```
(define (values . arguments)
  (call-with-current-continuation
    (lambda (k)
      (apply k arguments))))
```

Since **values** can be written so easily, you may well wonder why there is any need for a multiple-values proposal at all. The reason is that standard Scheme specifies that the continuations reified by **call-with-current-continuation** accept exactly one argument. Thus, the invocation (**values** 1 2) is currently defined to be in error. What is needed is some way to provide continuations that *do* accept other numbers of arguments.

The procedure **call-with-values** provides this capability. It takes as arguments two procedures, call them *producer* and *consumer*. It invokes *producer* with no arguments and then passes all of the values *producer* returns as arguments to *consumer*. The result(s) of calling *consumer* are returned by **call-with-values** as well.

Thus, this expression returns 5:

```
(call-with-values (lambda ()
                    (values 2 3))
                  +)
```

and this is an overly-complex procedure to return the absolute difference between its two real arguments:

```
(lambda (x y)
  (call-with-values
    (lambda ()
      (if (> x y)
        (values x y)
        (values y x)))
    (lambda (bigger smaller)
      (- bigger smaller))))
```

Getting back to the hash table design from earlier, you would write **table-ref** itself using **values** and any clients of **table-ref** using **call-with-values**.

A few questions remain concerning (a) the effect of returning a different number of values than the continuation expects, and (b) the number of values expected by all of the various kinds of continuations.

In relation to the first question, consider the following expression:

```
(call-with-values (lambda ()
                    (values 1 2 3))
                  cons)
```

In this case, the continuation of the `values` expression is expecting two values, the arguments to `cons`, but three are returned. Under the proposal, this is treated the same as passing the wrong number of arguments in a normal procedure call; that is, it is an error for this to happen.

This is a different approach from that taken in the corresponding part of Common Lisp. In that language, any "extra" returned values are simply ignored and when more values are expected than provided, the lack is made up with the appropriate number of `nils`. The concensus among the proposers was that this behavior was inconsistent (because normal procedure call doesn't work that way in either language) and, in some vague sense, too "unprincipled".

This all begs the question of the number of values expected by those continuations *not* created by `call-with-values`. There are five kinds of such continuations, or evaluation contexts:

- the procedure position in a call,
- an argument position in a call,
- the subexpression in a `set!` expression,
- the test position in an `if` expression, and
- any expression except the last in a `begin` expression.

Every expression in a standard Scheme program is evaluated in one of these contexts.

There was general agreement among the proposers that the first four of these contexts should expect exactly one value, to be used in the obvious way. There was more controversy concerning the last context above, sometimes called *effect* context, since the values of such expressions are not used.

One subset of the proposers felt that, for consistency among the contexts, effect context should also expect exactly one value.

Another side, which included me, believed that effect context should allow any number of values, including none, and ignore them all. It was argued that the restriction to a single value in effect context discriminates against multiple-result procedures in a case where the semantics is reasonably well-defined.

Finally, it was also suggested that perhaps effect context should insist on receiving *zero* values. There was somewhat less support for this position, on the grounds that it would be too inconvenient and would break too much existing code.

This issue, of the number of values expected by effect context, is likely to be debated at some length when the proposal is formally considered by the full group of authors.

The fact that the expression `(values)` returns zero values is considered by some to be very useful, and not just a notational accident. In particular, there are a number of standard procedures in Scheme whose returned value is "unspecified"; `set-car!` and `close-input-port` are two examples. Some have suggested that this specification be weakened to say that such procedures return *an unspecified number* of unspecified values. This would allow some implementations to experiment with the idea of such procedures returning zero values. If this were done in an implementation that signalled an error whenever the wrong number of values were returned to some context, then a certain class of portability errors could be caught at the moment they occur. This suggestion has not received widespread vocal support as yet, but it is likely to come up again when the multiple-values proposal is considered.

While `call-with-values` provides an essential service in a simple and general way, uses of it are usually very verbose and obfuscatory. For example, consider a typical client of **table-ref**:

```
(call-with-values
  (lambda ()
    (table-ref table key))
  (lambda (value found?)
    (if found?
      ...)))
```

This compares rather poorly with clients of a single-valued **table-ref**:

```
(let ((value (table-ref table key)))
  (if value
    ...))
```

The proposal does not include any remedy for this problem, on the theory that some significant experimentation is required before we can settle on standard syntactic sugar.

We have been planning to experiment with this issue in SchemeXerox (our implementation of Scheme here at PARC); I'll close this column with a description of a few of our ideas. None have been implemented yet, so comments and suggestions are especially welcome.

In most uses of `call-with-values`, much of the template above is constant:

```
(call-with-values (lambda () expr)
  (lambda (variables ...)
    body))
```

One could simply package up this idiom in a simple form:

```
(bind-values (variables ...)
             expr
  body)
```

This is very similar in syntax and function to the Common Lisp **multiple-value-bind** construct. A problem with this approach is that one frequently ends up writing code like this:

```
(let* ((a (foo))
       (b (bar a)))
  (bind-values (c d)
               (baz a b)
    (let* ((e (mumble a b c d)))
      (frotz a b c d e))))
```

Because the binding of multiple values uses a different construct than the binding of a single value, visual clutter and apparent complexity result. We are thus considering allowing a list of variables to appear in place of a single one in `let` and `let*` expressions:

```
(let* ((a (foo))
       (b (bar a))
       ((c d) (baz a b))
       (e (mumble a b c d)))
  (frotz a b c d e))
```

Another problem concerns situations where a procedure returns several values but only one is needed. For example, if we wanted to test whether or not a particular key had an associated value in a hash table, we would have to write this code:

```
(let (((value found?)
       (table-ref table key)))
  found?)
```

While the extension to `let` has made this much more palatable than the equivalent code using `call-with-values`, perhaps it is worthwhile adding a `values-ref` expression:

```
(values-ref (table-ref table key) 1)
```

If this sort of idiom is particularly common, it might even make sense to add lexical-level syntax to support it:

```
#1(table-ref table key)
```

As I mentioned above, we have not yet implemented any of these ideas and so are most open to comments and suggestions. Perhaps next issue I'll describe the most interesting ideas I receive. λ