The Scheme of Things: User-Defined Data Types[1]

Jonathan Rees
Cornell University
jar@cs.cornell.edu

Most modern programming languages provide some way to define new types. Scheme doesn't. What gives?

The record facility previously described in this column[2] was proposed at the June 1992 meeting of the Scheme report authors, but didn't meet with broad agreement. The fact that a need for user-defined types is strongly felt but no consensus has been reached hints that something interesting is going on. In this column I hope to motivate the need for user-defined types and to articulate the structure of the large and murky space of designs for possible Scheme type-definition facilities.

The original rationale for leaving data type definitions out of Scheme was minimalist: No such facility is needed. If you want a new type, define it yourself on top of the types provided. For example, if I need a two-element record in order to implement a FIFO queue data type, I can use pairs:

```
(define (make-queue) (cons '() '()))

(define (enqueue! q obj)
  (set-car! q (cons obj (car q))))

(define (dequeue! q)
  (if (null? (cdr q))
      (begin (set-cdr! q (reverse (car q)))
             (set-car! q '())))
  (let ((head (car (cdr q))))
    (set-cdr! q (cdr (cdr q)))
    head))
```

(The `queue-empty?` predicate and a check for empty queue in `dequeue!` have been omitted for the sake of brevity.)

[1] This article appeared in *Lisp Pointers* V(3), pages 39–46, July–September 1992.

The first change to be made is to try to make queues distinguishable from other objects. There are several different reasons for this:

1. Debugging: When I accidentally pass an ordinary list to `enqueue!` or `dequeue!`, I would like to see a meaningful diagnostic message.

2. Debugging: When I accidentally pass a queue to an operator like `length` that expects a list, I would like to see a meaningful diagnostic message.

3. Disjointness: One would like to be able write case analyses that discriminate between queues and members of other Scheme types.

The first of these is easily addressed: Change the representation of queues so that they are marked as being queues. For example, queues could be represented as three-element vectors, where one element (the first, say) is a unique token:

```
(define queue-unique-token (list 'queue))

(define (make-queue) (vector queue-unique-token '() '()))

(define (enqueue! q obj)
  (if (queue? q)
      (vector-set! q 1 (cons obj (vector-ref q 1)))
      (error "expected a queue but found this instead"
             q)))

(define (dequeue! q) ...)

(define (queue? obj)
  (and (vector? obj)
       (= (vector-length obj) 3)
       (eq? (vector-ref obj 0) queue-unique-token)))
```

This particular choice of unique token even makes queues easily identifiable when displayed by `write`.

Our only hope of addressing goal number 2 is to represent queues as procedures. Pairs as the representation are subject to accidental `append`s, `length`s, and so forth, and vectors are prone to accidental `vector-fill!`s; a process of elimination and dim recollection of Scheme programming folklore lead us to try procedures, which only support a single operation, application. Using the classic implementation technique, we get something like this:

```
(define (make-queue)
  (let ((incoming '()) (outgoing '()))
    (lambda (operation)
      (case operation
        ((enqueue!)
         (lambda (obj)
           (set! incoming (cons obj incoming))))
        ((dequeue!)
         (if (null? outgoing)
             (begin (set! outgoing (reverse incoming))
                    (set! incoming '())))
         (let ((head (car outgoing)))
           (set! outgoing (cdr outgoing))
           head))
        (else (error "unrecognized queue operation"
                     operation))))))

(define (enqueue! q obj) ((q 'enqueue!) obj))
(define (dequeue! q) (q 'dequeue!))
```

The implementation has even become easier to read because all the vector operations (or `car/cdr`s) have been replaced by accesses and assignments to variables with mnemonic names. (We could have obtained most of this benefit earlier by introducing auxiliary access and modification functions `queue-incoming`, `set-queue-incoming!`, `queue-outgoing`, and `set-queue-outgoing!`.)

Since application is the only operation available on procedures, the only way we can get into trouble on that account is by accidentally calling a queue as if it were a procedure, passing it a single argument that is one of the two symbols `'enqueue!` or `'dequeue!`. This is highly unlikely, but still possible. Happily, if we happen to be paranoid about this, we can make it quite impossible by lexically closing the queue module over unique tokens accessible only to it:

```
(define make-queue #f)
(define enqueue! #f)
(define dequeue! #f)
(let ((queue-module
       (let ()
         (define enqueue!-token (list 'enqueue!))
         (define dequeue!-token (list 'dequeue!))
         (define (make-queue)
           (let ((incoming '())
                 (outgoing '()))
             (lambda (operation)
               (cond ((eq? operation enqueue!-token)
                      (lambda (obj) ...))
                     ((eq? operation dequeue!-token) ...)
                     (else
                      (error "unrecognized queue operation"
                             operation))))))
         (define (enqueue! q obj) ((q enqueue!-token) obj))
         (define (dequeue! q) (q dequeue!-token))
         (list make-queue enqueue! dequeue!))))
  (set! make-queue (car queue-module))
  (set! enqueue! (cadr queue-module))
  (set! dequeue! (caddr queue-module)))
```

(The encapsulation idiom used here — defining the exported variables to be `#f`, creating a new scope with `(let () ...)`, and extracting all the exports from some single object containing them — can easily be captured by a module building macro. This is left as an exercise.)

But now we have failed to meet goal number 1. Sure, we can't accidentally apply a list or vector operation to a queue, or apply a queue to any unexpected arguments, but we can apply queue operations to procedures. We would like the following to be errors, but instead they quietly return useless values:

```
(dequeue! list)
(enqueue! (lambda (ignore) list))
```

This is a result of the same aspect of procedures that made them attractive in the first place: The only operation on procedures is application to arguments, and applying an unknown procedure to any set of arguments could have disastrous consequences (e.g. suppose it is the `launch-missile` procedure). Scheme doesn't give us any way to safely distinguish the particular procedures that are queues from those that aren't.

But even worse, all the representations described so far fail to meet goal 3. No matter what we do, queues will always return true to some built-in Scheme predicate — `pair?`, `vector?`, or `procedure?` for the above implementations, or some other predicate for the less plausible representations (numbers, ports, etc.). Suppose, for example, that we wanted to write a general object printing utility, and that this utility was intended to do something reasonable with queues as well as with other Scheme objects like vectors. Somewhere there would be a case analysis on the type of the argument. Such a case analysis would plausibly look something like the following:

```
(cond ...
      ((vector? obj) (print-vector obj))
      ((queue? obj) (print-queue obj))
      ...)
```

If queues are implemented as vectors, then the `queue?` case will never be reached. We could just stipulate that such case analyses must try applying `queue?` before `vector?` (and `pair?`), but this gratuitously exposes an implementation detail to users of the queue abstraction. If queues are implemented as procedures, then there is no hope of implementing a `queue?` predicate at all.

This is the line of reasoning that leads even minimalists to urge the Scheme report authors to add data type definitions to the language. Which leads to a discussion of the language design space.

* * *

The earnest minimalist might propose the addition to Scheme of just one new type, with a constructor, a single accessor, and a predicate. Choosing the bland term *entity* for these new objects, the required primitives are:

(`make-entity` *object*) $\longrightarrow$ *entity*
(`entity-value` *entity*) $\longrightarrow$ *object*
(`entity? obj`) $\longrightarrow$ *boolean*

This seems to solve the problem of disjointness of a user-defined type from other Scheme types. I can implement queues as entities whose value components are pairs, as follows:

```
(define (make-queue) (make-entity (cons '() '())))
(define (enqueue! q obj) ... (entity-value q) ...)
...
(define queue? entity?)
```

But now I have commandeered the entity type for my own purposes, leaving it unavailable for others. If I had multiple types to represent, I could install markers in my own entities to distinguish their various types, but if there's no agreement from people implementing other types on which I might depend, then chaos will ensue. The representation value must generally be a pair or vector, and all the problems that arose earlier appear again. It would be better to canonize a mechanism for generating new types. Given a single type, as above, it is possible to implement such an institution (for example, the record proposal). But for the purpose of sharing program *modules* instead of just complete programs, it is desirable for all Scheme programmers to use the *same* such institution, at which point we might as well add it to the language definition.

Suppose then that there is to be an unbounded supply of triples ⟨*constructor, accessor, predicate*⟩, each manipulating a type disjoint from all others. A single such triple is sufficient to establish a new type. The different predicates are true for disjoint sets of objects, and it is an error to apply one type's accessor to members of another type. A type definition facility might then take the form of a new kind of definition:

(`define-type` *constructor accessor predicate*)

For example,

```
(define-type rep->queue queue->rep queue?)
```

would simultaneously define `rep->queue`, `queue->rep`, and `queue?`. `rep` is short for "representation." Then with pairs as the representation for queues, we could define

```
(define (make-queue)
  (rep->queue (cons '() '())))
(define (enqueue! q obj)
  ... (queue->rep q) ...)
```

and so forth. A facility like this has something of the flavor of Common Lisp `defstruct`.

Alternatively, type generation might take a more dynamic form:

> `(make-new-type)` $\longrightarrow$ *type-descriptor*
> `(type-constructor` *type-descriptor*`)` $\longrightarrow$ *procedure*
> `(type-accessor` *type-descriptor*`)` $\longrightarrow$ *procedure*
> `(type-predicate` *type-descriptor*`)` $\longrightarrow$ *procedure*

with

```
(define queue-type (make-new-type))
(define (make-queue)
  ((type-constructor queue-type) (cons '() '())))
```

and so forth.

These two forms are equipotent. Clearly `define-type` can be defined as a macro that generates a call to `make-new-type` and definitions for the constructor, accessor, and predicate extracted from the new type descriptor. Unless `define-type` is to be a primitive special form, something like `make-new-type` must exist in order for `define-type` to be implementable as a macro.

Less obviously, `make-new-type` can be defined in terms of `define-type`, assuming that `define-type` forms are permitted where internal definitions are:

```
(define (make-new-type)
  (define-type new-type make access has-type?)
  (lambda (operation)
    (case operation
      ((constructor) make)
      ((accessor) access)
      ((predicate) has-type?)
      (else (error ...)))))

(define (type-constructor type) (type 'constructor))
```

and so on.

Allowing type definitions in such non-top-level contexts is necessary in order to be true to the spirit of block structure.

Given this equivalence, together with Scheme's historical aversion to adding new primitive syntactic forms, the `make-new-type` primitives seem preferable to `define-type`.

The `make-new-type` primitive is non-applicative because each call to it must generate a different type descriptor. It is odd that an almost-functional language seems to require a non-applicative primitive in order to be able to define abstract data types. Section 3.2 of [3] discusses two applicative models for abstract data types; one model relies on static typing, and is open to some forms of abstraction violation, while the other relies on the use of passwords. These do not seem applicable to Scheme, although it would be interesting to attempt to adapt them somehow.

Some variants on `make-new-type` present themselves.

- For those averse to adding a type-descriptor type to the language, the `make-new-type` primitive could instead return the constructor, accessor, and predicate as three values using Scheme's newly-approved multiple value return mechanism (see the description of `call-with-values` and `values` in [2]).

- Another way to avoid a type of type descriptors would be to have the constructor, accessor, and predicate each take two arguments:

  (`rep->datum` *object type-identifier*) $\longrightarrow$ *datum*
  (`datum->rep` *datum type-identifier*) $\longrightarrow$ *object*
  (`has-type?` *object type-identifier*) $\longrightarrow$ *boolean*

  These are just un-curried version of `type-constructor`, `type-accessor`, and `type-predicate`. But now a separate type descriptor type is unnecessary, since the *type-identifier* argument to these procedures could be an arbitrary user-supplied object, such as a unique token.

- Stepping a bit away from minimalism now: Members of user-defined types could directly contain multiple fields. Fields could be indexed, as are vector components, or named, as in the record proposal. One argument for providing this functionality would be efficiency. An extra indirection might be avoided, since a member of a multi-component user-defined type could be represented directly as a single object in memory rather than as two. A reason not to do this is orthogonality: It is somewhat inelegant to bundle the user-defined type mechanism with a new way to make compound objects (product types), given that Scheme has two kinds of compound data already.

A final language design question is that of the *opaqueness* of user-defined types. A data type is *opaque* (or *abstract*) if clients of the type have no access to its representation beyond advertised interface routines. The implementation of queues in terms of pairs is not abstract because clients can get at the representation using `car` and `cdr`. `make-new-type` as described above provides abstract types, because the type descriptor itself acts as a key or capability. If the data type implementation doesn't make the type descriptor accessible to clients, then clients will have no direct way to access representations. If, however, there is a `get-type` primitive that goes from an object of a user-defined type to its type descriptor, then any client can access any representation:

```
(define (representation object)
  ((type-accessor (get-type object)) object))
```

The language design questions here are:

1. Should the language provide any secure representation-hiding mechanism at all?

2. Should user-defined types be opaque?

The Scheme report answers the first question in the affirmative by specifying procedures to be an opaque type. There are those who believe that even this is evil — they see a closed door and insist on being able to get to the other side. Procedures have *some* representation inside the computer, the reasoning goes, and it's unfriendly of Scheme to prevent access to that representation. Access to the representation would be useful in writing, say, a portable debugger or garbage collector. Arguing against this position, however, is the difficulty of giving any clear semantics to such access primitives that would not greatly inhibit the ability to reason about programs and program transformations.

Given that the language has opaqueness in the form of procedures, the second question amounts to asking whether it should be *convenient* to obtain opaque user-defined types. This could go either way:

- Convenient: As described above, require a key or capability in order to be able to use the representation-exposing primitives. Opaqueness is achieved by making the key inaccessible.

- Inconvenient: Provide `get-type` so that clients can inspect representations. A programmer that desires inaccessible representations can

> encapsulate the representation in a procedure. If the representation is a procedure, then members of the new type have as much opaqueness as procedures do, because it is in general impossible to synthesize the right combination of arguments that will cause the procedure to divulge any information of interest.

* * *

This discussion is just the tip of the iceberg. The study of type systems is a small industry within academic computer science. Most of this work is in the context of static type systems, but as we have seen, even dynamic type systems like Scheme's present plenty of challenges.

I haven't spoken about performance, but it will inevitably come up in a full discussion of data type definition facilities. Performance issues include the number of machine operations (indirections in particular) necessary to select fields from records defined as members of user-defined types, the amount of memory required to represent such records, and optimized representations of components, such as non-boxed floating point fields.

An alternative approach to user-defined types is object orientation. The correspondence between first-class procedures and message-accepting objects, with argument lists playing the role of messages, makes Scheme object-oriented *a priori.* But the fact that procedures are used both as objects and as subroutines would lead us to propose the creation of a new type to separate those procedures that are intended to be message-accepting objects from those such as `length` that aren't. This amounts to something like `make-entity` where the representation is required to be a message handler procedure that adheres to certain protocols. (This is exactly what the pre-flavors MIT Lisp Machine system had.) Further exploration leads to a succession of increasingly structured frameworks, leading perhaps to classes, inheritance, and generic functions.

Many fully object-oriented Scheme dialects have been invented, the latest being Dylan [1]. Object orientation is still not well understood according to the conservative standards of those cantankerous Scheme report authors. But let's hope that a future Scheme report will settle on some structured way to express type definitions.

* * *

I would like to thank Norman Adams, Pavel Curtis, Bruce Donald, and Richard Kelsey for their comments on drafts of this article.

References

[1] Apple Computer Eastern Research and Technology. *Dylan: An Object-Oriented Dynamic Language.* Apple Computer, Inc., 1992.

[2] Pavel Curtis. The Scheme of Things. *Lisp Pointers* IV(1): 61–67, ACM Press, 1991.

[3] B. Lampson and R. Burstall. Pebble, a kernel language for modules and abstract data types. *Information and Computation* 76(2/3): 278-346, 1988.