Title

Defining Record Types

Author

Richard Kelsey

Status

This SRFI is currently in *final* status. Here is an explanation of each status that a SRFI can hold. To provide input on this SRFI, please send email to `srfi-9@srfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Received: 1999-07-01
- Revised: 1999-08-25
- Draft: 1999-07-07--1999-09-06
- Final: 1999-09-09

Abstract

This SRFI describes syntax for creating new data types, called record types. A predicate, constructor, and field accessors and modifiers are defined for each record type. Each new record type is distinct from all existing types, including other record types and Scheme's predefined types.

Rationale

Many Scheme implementations provide means for creating new types, usually called either records or structures. The `DEFINE-RECORD-TYPE` syntax described here is a slight simplification of one written for Scheme 48 by Jonathan Rees. Unlike many record-defining macros or special forms, it does not create any new identifiers. Instead, the names of the record type, predicate, constructor, and so on are all listed explicitly in the source. This has the following advantages:

- It can be defined using a simple `SYNTAX-RULES` macro in Scheme implementations that provide a procedural interface for creating record types.
- It does not restrict users to a particular naming convention.
- Tools like `grep` and GNU Emacs's tag facility will see the defining occurrence of each identifier.

Specification

The syntax of a record-type definition is:

```
<command or definition>
  -> <record type definition>          ; addition to 8.1.6 in R5RS
```

```
<record type definition>
  -> (define-record-type <type name>
       (<constructor name> <field tag> ...)
       <predicate name>
       <field spec> ...)

<field spec> -> (<field tag> <accessor name>)
             -> (<field tag> <accessor name> <modifier name>)

<field tag> -> <identifier>
<... name>  -> <identifier>
```

`DEFINE-RECORD-TYPE` is generative: each use creates a new record type that is distinct from all existing types, including other record types and Scheme's predefined types. Record-type definitions may only occur at top-level (there are two possible semantics for `internal' record-type definitions, generative and nongenerative, and no consensus as to which is better).

An instance of `DEFINE-RECORD-TYPE` is equivalent to the following definitions:

- `<type name>` is bound to a representation of the record type itself. Operations on record types, such as defining print methods, reflection, etc. are left to other SRFIs.
- `<constructor name>` is bound to a procedure that takes as many arguments as there are `<field tag>`s in the `(<constructor name> ...)` subform and returns a new `<type name>` record. Fields whose tags are listed with `<constructor name>` have the corresponding argument as their initial value. The initial values of all other fields are unspecified.
- `<predicate name>` is a predicate that returns #T when given a value returned by `<constructor name>` and #F for everything else.
- Each `<accessor name>` is a procedure that takes a record of type `<type name>` and returns the current value of the corresponding field. It is an error to pass an accessor a value which is not a record of the appropriate type.
- Each `<modifier name>` is a procedure that takes a record of type `<type name>` and a value which becomes the new value of the corresponding field; an unspecified value is returned. It is an error to pass a modifier a first argument which is not a record of the appropriate type.

Records are disjoint from the types listed in Section 4.2 of R5RS.

`Set!`ing the value of any of these identifiers has no effect on the behavior of any of their original values.

The following

```
(define-record-type :pare
  (kons x y)
  pare?
  (x kar set-kar!)
  (y kdr))
```

defines `KONS` to be a constructor, `KAR` and `KDR` to be accessors, `SET-KAR!` to be a modifier, and `PARE?` to be a predicate for `:PARE`s.

```
(pare? (kons 1 2))        --> #t
(pare? (cons 1 2))        --> #f
(kar (kons 1 2))          --> 1
(kdr (kons 1 2))          --> 2
(let ((k (kons 1 2)))
  (set-kar! k 3)
  (kar k))                --> 3
```

Implementation

This code is divided into three layers. In top-down order these are:

1. Syntax definitions for `DEFINE-RECORD-TYPE` and an auxillary macro.
2. An implementation of record types with a procedural interface. Some Scheme implementations already have something close to this.
3. Vector-like records implemented in R5RS. This redefines some standard Scheme procedures and therefor must be loaded before any other code, including part 2 above. Note that these procedures can be used to break the record-type abstraction (for example, `RECORD-SET!` can be used to modify the type of a record). Access to these procedures should be restricted.

Syntax definitions

```
; Definition of DEFINE-RECORD-TYPE

(define-syntax define-record-type
  (syntax-rules ()
    ((define-record-type type
       (constructor constructor-tag ...)
       predicate
       (field-tag accessor . more) ...)
     (begin
       (define type
         (make-record-type 'type '(field-tag ...)))
       (define constructor
         (record-constructor type '(constructor-tag ...)))
       (define predicate
         (record-predicate type))
       (define-record-field type field-tag accessor . more)
       ...))))

; An auxilliary macro for define field accessors and modifiers.
; This is needed only because modifiers are optional.

(define-syntax define-record-field
  (syntax-rules ()
    ((define-record-field type field-tag accessor)
     (define accessor (record-accessor type 'field-tag)))
    ((define-record-field type field-tag accessor modifier)
     (begin
```

```
        (define accessor (record-accessor type 'field-tag))
        (define modifier (record-modifier type 'field-tag))))))
```

Record types

```
; We define the following procedures:
;
; (make-record-type <type-name <field-names>)    -> <record-type>
; (record-constructor <record-type<field-names>) -> <constructor>
; (record-predicate <record-type>)               -> <predicate>
; (record-accessor <record-type <field-name>)    -> <accessor>
; (record-modifier <record-type <field-name>)    -> <modifier>
;   where
; (<constructor> <initial-value> ...)         -> <record>
; (<predicate> <value>)                       -> <boolean>
; (<accessor> <record>)                       -> <value>
; (<modifier> <record> <value>)         -> <unspecific>

; Record types are implemented using vector-like records.  The first
; slot of each record contains the record's type, which is itself a
; record.

(define (record-type record)
  (record-ref record 0))

;----------------
; Record types are themselves records, so we first define the type for
; them.  Except for problems with circularities, this could be defined as:
;  (define-record-type :record-type
;    (make-record-type name field-tags)
;    record-type?
;    (name record-type-name)
;    (field-tags record-type-field-tags))
; As it is, we need to define everything by hand.

(define :record-type (make-record 3))
(record-set! :record-type 0 :record-type)       ; Its type is itself.
(record-set! :record-type 1 ':record-type)
(record-set! :record-type 2 '(name field-tags))

; Now that :record-type exists we can define a procedure for making more
; record types.

(define (make-record-type name field-tags)
  (let ((new (make-record 3)))
    (record-set! new 0 :record-type)
    (record-set! new 1 name)
    (record-set! new 2 field-tags)
    new))

; Accessors for record types.
```

```
(define (record-type-name record-type)
  (record-ref record-type 1))

(define (record-type-field-tags record-type)
  (record-ref record-type 2))

;----------------
; A utility for getting the offset of a field within a record.

(define (field-index type tag)
  (let loop ((i 1) (tags (record-type-field-tags type)))
    (cond ((null? tags)
           (error "record type has no such field" type tag))
          ((eq? tag (car tags))
           i)
          (else
           (loop (+ i 1) (cdr tags))))))

;----------------
; Now we are ready to define RECORD-CONSTRUCTOR and the rest of the
; procedures used by the macro expansion of DEFINE-RECORD-TYPE.

(define (record-constructor type tags)
  (let ((size (length (record-type-field-tags type)))
        (arg-count (length tags))
        (indexes (map (lambda (tag)
                        (field-index type tag))
                      tags)))
    (lambda args
      (if (= (length args)
             arg-count)
          (let ((new (make-record (+ size 1))))
            (record-set! new 0 type)
            (for-each (lambda (arg i)
                        (record-set! new i arg))
                      args
                      indexes)
            new)
          (error "wrong number of arguments to constructor" type args)))))

(define (record-predicate type)
  (lambda (thing)
    (and (record? thing)
         (eq? (record-type thing)
              type))))

(define (record-accessor type tag)
  (let ((index (field-index type tag)))
    (lambda (thing)
      (if (and (record? thing)
               (eq? (record-type thing)
                    type))
          (record-ref thing index)
```

```
          (error "accessor applied to bad value" type tag thing)))))

(define (record-modifier type tag)
  (let ((index (field-index type tag)))
    (lambda (thing value)
      (if (and (record? thing)
               (eq? (record-type thing)
                    type))
          (record-set! thing index value)
          (error "modifier applied to bad value" type tag thing)))))
```

Records

```
; This implements a record abstraction that is identical to vectors,
; except that they are not vectors (VECTOR? returns false when given a
; record and RECORD? returns false when given a vector).  The following
; procedures are provided:
;   (record? <value>)                -> <boolean>
;   (make-record <size>)             -> <record>
;   (record-ref <record> <index>)    -> <value>
;   (record-set! <record> <index> <value>) -> <unspecific>
;
; These can implemented in R5RS Scheme as vectors with a distinguishing
; value at index zero, providing VECTOR? is redefined to be a procedure
; that returns false if its argument contains the distinguishing record
; value.  EVAL is also redefined to use the new value of VECTOR?.

; Define the marker and redefine VECTOR? and EVAL.

(define record-marker (list 'record-marker))

(define real-vector? vector?)

(define (vector? x)
  (and (real-vector? x)
       (or (= 0 (vector-length x))
           (not (eq? (vector-ref x 0)
                     record-marker)))))

; This won't work if ENV is the interaction environment and someone has
; redefined LAMBDA there.

(define eval
  (let ((real-eval eval))
    (lambda (exp env)
      ((real-eval `(lambda (vector?) ,exp))
       vector?))))

; Definitions of the record procedures.

(define (record? x)
  (and (real-vector? x)
```

```
       (< 0 (vector-length x))
       (eq? (vector-ref x 0)
            record-marker)))

(define (make-record size)
  (let ((new (make-vector (+ size 1))))
    (vector-set! new 0 record-marker)
    new))

(define (record-ref record index)
  (vector-ref record (+ index 1)))

(define (record-set! record index value)
  (vector-set! record (+ index 1) value))
```

Copyright

Editor: Mike Sperber