Title

Conditions

Authors

Richard Kelsey and Michael Sperber

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-35@srfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Draft: 2002-07-24--2002-10-20
- Revised: 2002-09-20
- Final: 2002-12-01

Abstract

The SRFI defines constructs for creating and inspecting *condition* types and values. A condition value encapsulates information about an exceptional situation, or exception. This SRFI also defines a few basic condition types.

Rationale

Conditions are values that communicate information about exceptional situations between parts of a program. Code that detects an exception may be in a different part of the program than the code that handles it. In fact, the former may have been written independently from the latter. Consequently, to facilitate effective handling of exceptions, conditions must communicate as much information as possible as accurately as possible, and still allow effective handling by code that did not precisely anticipate the nature of the exception that occurred.

This SRFI provides two mechanisms to enable this kind of communication

- subtyping among condition types allows handling code to determine the *general* nature of an exception even though it does not anticipate its *exact* nature,
- compound conditions allow an exceptional situation to be described in multiple ways.

Specification

Conditions are records with named fields. Each condition belongs to one or more *condition types*. Each condition type specifies a set of field names. A condition belonging to a condition type includes a value for each of the type's field names. These values can be extracted from the condition by using the appropriate field name.

There is a tree of condition types with the distinguished `&condition` as its root. All other condition types have a parent condition type.

A condition belonging to several condition types with a common supertype may have distinct values for the supertype's fields for each type. The type used to access a field determines which of the values is returned. The program can extract each of these field values separately.

Procedures

`(make-condition-type` *id parent field-names*`)`

> `Make-condition-type` returns a new condition type. *Id* must be a symbol that serves as a symbolic name for the condition type. *Parent* must itself be a condition type. *Field-names* must be a list of symbols. It identifies the fields of the conditions associated with the condition type.
>
> *Field-names* must be disjoint from the field names of *parent* and its ancestors.

`(condition-type?` *thing*`)`

> `Condition-type?` is a predicate for condition types: it returns `#t` if *thing* is a condition type, and `#f` otherwise

`(make-condition` *type field-name value ...*`)`

> `Make-condition` creates a condition value belonging condition type *type*. The following arguments must be, in turn, a field name and an arbitrary value. There must be such a pair for each field of *type* and its direct and indirect supertypes. `Make-condition` returns the condition value, with the argument values associated with their respective fields.

`(condition?` *thing*`)`

> `Condition?` is a predicate for conditions: it returns `#t` if *thing* is a condition type, and `#f` otherwise

`(condition-has-type?` *condition condition-type*`)`

> `Condition-has-type?` tests if condition *condition* belongs to condition type *condition-type*. It returns `#t` if any of *condition* 's types includes *condition-type* eitherdirectlyorasanancestorand `#f` otherwise.
>
> It is an error if *condition* is not a condition, or if *condition-type* is not a condition type.

`(condition-ref` *condition field-name*`)`

Condition must be a condition, and *field-name* a symbol. Moreover, *condition* must belong to a condition type which has a field name called *field-name*, or one of its (direct or indirect) supertypes must have the field. `Condition-ref` returns the value associated with *field-name*.

It is an error to refer to a field the condition does not have.

`(make-compound-condition` *condition*$_0$ *condition*$_1$ *...*`)`

`Make-compound-condition` returns a compound condition belonging to all condition types that the *condition*$_i$ belong to.

`Condition-ref`, when applied to a compound condition will return the value from the first of the *condition*$_i$ that has such a field.

`(extract-condition` *condition condition-type*`)`

Condition must be a condition belonging to *condition-type*. `Extract-condition` returns a condition of condition type *condition-type* with the field values specified by *condition*.

If *condition* is a compound condition, `extract-condition` extracts the field values from the subcondition belonging to *condition-type* that appeared first in the call to `make-compound-condition` that created the the condition. The returned condition may be newly created; it is possible for

```
(let* ((&c (make-condition-type 'c &condition '()))
       (c0 (make-condition &c))
       (c1 (make-compound-condition c0)))
  (eq? c0 (extract-condition c1 &c)))
```

to return false.

Macros

`(define-condition-type` <condition-type> <supertype> <predicate> <field-spec> ...`)`

This defines a new condition type. <Condition-type>, <supertypes>, and <predicate> must all be identifiers. `Define-condition-type` defines an identifier <condition-type> to some value describing the condition type. <supertype> must be the name of a previously defined condition type.

`Define-condition-type` also defines <predicate> to a predicate that identifies conditions associated with that type, or with any of its subtypes.

Each <field-spec> must be of the form `(` <field> <accessor>`)` where both <field> and <accessor> must be identifiers. `Define-condition-type` defines each <accessor> to a procedure which extracts the value of the named field from a condition associated with this condition type.

`(condition` <type-field-binding> ...`)`

This creates a condition value. Each <type-field-binding> must be of the form `(` <condition-type> <field-binding> ...`)` Each <field-binding> must be of the form `(` <field> <exp>`)` where <field> is a field identifier from the definition of <condition-type>.

The <exp> are evaluated in some unspecified order; their values can later be extracted from the condition object via the accessors of the associated condition types or their supertypes.

The condition returned by `condition` is created by a call of form

```
(make-compound-condition
  (make-condition <condition-type> '<field-name> <value>...)
  ...)
```

with the condition types retaining their order from the`condition` form. The field names and values are duplicated as necessary as described below.

Each <type-field-binding> must contain field bindings for *all* fields of <condition-type> without duplicates. There is an exception to this rule: if a field binding is missing, and the field belongs to a supertype shared with one of the other <type-field-binding> subforms, then the value defaults to that of the first such binding in the `condition` form.

Standard Conditions

`&condition`

This is the root of the entire condition type hierarchy. It has a no fields.

`&message`

This condition type could be defined by

```
(define-condition-type &message &condition
  message-condition?
  (message condition-message))
```

It carries a message further describing the nature of the condition to humans.

`&serious`

This condition type could be defined by

```
(define-condition-type &serious &condition
  serious-condition?)
```

This type describes conditions serious enough that they cannot safely be ignored. This condition type is primarily intended as a supertype of other condition types.

`&error`

This condition type could be defined by

```
(define-condition-type &error &serious
  error?)
```

This condition describes errors, typically caused by something that has gone wrong in the interaction of the program with the external world or the user.

Examples

```
(define-condition-type &c &condition
  c?
  (x c-x))

(define-condition-type &c1 &c
  c1?
  (a c1-a))

(define-condition-type &c2 &c
  c2?
  (b c2-b))
(define v1 (make-condition &c1 'x "V1" 'a "a1"))

(c? v1)         => #t
(c1? v1)        => #t
(c2? v1)        => #f
(c-x v1)        => "V1"
(c1-a v1)       => "a1"

(define v2 (condition (&c2
                        (x "V2")
                        (b "b2"))))

(c? v2)         => #t
(c1? v2)        => #f
(c2? v2)        => #t
(c-x v2)        => "V2"
(c2-b v2)       => "b2"

(define v3 (condition (&c1
                       (x "V3/1")
                       (a "a3"))
                      (&c2
                       (b "b3"))))

(c? v3)         => #t
(c1? v3)        => #t
(c2? v3)        => #t
(c-x v3)        => "V3/1"
(c1-a v3)       => "a3"
(c2-b v3)       => "b3"
```

```
(define v4 (make-compound-condition v1 v2))

(c? v4)         => #t
(c1? v4)        => #t
(c2? v4)        => #t
(c-x v4)        => "V1"
(c1-a v4)       => "a1"
(c2-b v4)       => "b2"

(define v5 (make-compound-condition v2 v3))

(c? v5)         => #t
(c1? v5)        => #t
(c2? v5)        => #t
(c-x v5)        => "V2"
(c1-a v5)       => "a3"
(c2-b v5)       => "b2"
```

Reference Implementation

The reference implementation makes use of SRFI 1 ("List Library"), SRFI 9 ("Defining Record Types"), and SRFI 23 ("Error reporting mechanism").

```
(define-record-type :condition-type
  (really-make-condition-type name supertype fields all-fields)
  condition-type?
  (name condition-type-name)
  (supertype condition-type-supertype)
  (fields condition-type-fields)
  (all-fields condition-type-all-fields))

(define (make-condition-type name supertype fields)
  (if (not (symbol? name))
      (error "make-condition-type: name is not a symbol"
             name))
  (if (not (condition-type? supertype))
      (error "make-condition-type: supertype is not a condition type"
             supertype))
  (if (not
       (null? (lset-intersection eq?
                                 (condition-type-all-fields supertype)
                                 fields)))
      (error "duplicate field name" ))
  (really-make-condition-type name
                              supertype
                              fields
                              (append (condition-type-all-fields supertype)
                                      fields)))

(define-syntax define-condition-type
  (syntax-rules ()
    ((define-condition-type ?name ?supertype ?predicate
       (?field1 ?accessor1) ...)
     (begin
       (define ?name
         (make-condition-type '?name
                              ?supertype
                              '(?field1 ...)))
       (define (?predicate thing)
         (and (condition? thing)
              (condition-has-type? thing ?name)))
       (define (?accessor1 condition)
         (condition-ref (extract-condition condition ?name)
                        '?field1))
       ...))))

(define (condition-subtype? subtype supertype)
  (let recur ((subtype subtype))
    (cond ((not subtype) #f)
          ((eq? subtype supertype) #t)
          (else
           (recur (condition-type-supertype subtype))))))

(define (condition-type-field-supertype condition-type field)
  (let loop ((condition-type condition-type))
    (cond ((not condition-type) #f)
          ((memq field (condition-type-fields condition-type))
           condition-type)
          (else
           (loop (condition-type-supertype condition-type))))))

; The type-field-alist is of the form
; ((<type> (<field-name> . <value>) ...) ...)
(define-record-type :condition
  (really-make-condition type-field-alist)
  condition?
  (type-field-alist condition-type-field-alist))

(define (make-condition type . field-plist)
  (let ((alist (let label ((plist field-plist))
                 (if (null? plist)
                          '()
                     (cons (cons (car plist)
                                 (cadr plist))
                           (label (cddr plist)))))))
    (if (not (lset= eq?
                    (condition-type-all-fields type)
                    (map car alist)))
        (error "condition fields don't match condition type"))
    (really-make-condition (list (cons type alist)))))

(define (condition-has-type? condition type)
  (any (lambda (has-type)
         (condition-subtype? has-type type))
       (condition-types condition)))

(define (condition-ref condition field)
  (type-field-alist-ref (condition-type-field-alist condition)
                        field))

(define (type-field-alist-ref type-field-alist field)
  (let loop ((type-field-alist type-field-alist))
    (cond ((null? type-field-alist)
           (error "type-field-alist-ref: field not found"
                  type-field-alist field))
          ((assq field (cdr (car type-field-alist)))
           => cdr)
          (else
           (loop (cdr type-field-alist))))))

(define (make-compound-condition condition-1 . conditions)
  (really-make-condition
   (apply append (map condition-type-field-alist
                      (cons condition-1 conditions)))))

(define (extract-condition condition type)
  (let ((entry (find (lambda (entry)
                              (condition-subtype? (car entry) type))
                            (condition-type-field-alist condition))))
    (if (not entry)
        (error "extract-condition: invalid condition type"
                      condition type))
    (really-make-condition
      (list (cons type
                  (map (lambda (field)
                         (assq field (cdr entry)))
                       (condition-type-all-fields type)))))))

(define-syntax condition
  (syntax-rules ()
    ((condition (?type1 (?field1 ?value1) ...) ...)
     (type-field-alist->condition
      (list
       (cons ?type1
             (list (cons '?field1 ?value1) ...))
       ...)))))

(define (type-field-alist->condition type-field-alist)
  (really-make-condition
   (map (lambda (entry)
          (cons (car entry)
                (map (lambda (field)
                       (or (assq field (cdr entry))
                           (cons field
                               (type-field-alist-ref type-field-alist field))))
                     (condition-type-all-fields (car entry)))))
        type-field-alist)))

(define (condition-types condition)
  (map car (condition-type-field-alist condition)))

(define (check-condition-type-field-alist the-type-field-alist)
  (let loop ((type-field-alist the-type-field-alist))
    (if (not (null? type-field-alist))
        (let* ((entry (car type-field-alist))
               (type (car entry))
               (field-alist (cdr entry))
               (fields (map car field-alist))
               (all-fields (condition-type-all-fields type)))
          (for-each (lambda (missing-field)
                      (let ((supertype
                             (condition-type-field-supertype type missing-field)))
                        (if (not
                             (any (lambda (entry)
                                    (let ((type (car entry)))
                                      (condition-subtype? type supertype)))
                                  the-type-field-alist))
                            (error "missing field in condition construction"
                                   type
                                   missing-field))))
                    (lset-difference eq? all-fields fields))
          (loop (cdr type-field-alist))))))

(define &condition (really-make-condition-type '&condition
                                               #f
                                               '()
                                               '()))

(define-condition-type &message &condition
  message-condition?
  (message condition-message))

(define-condition-type &serious &condition
  serious-condition?)

(define-condition-type &error &serious
  error?)
```

References

- SRFI 12: Exception Handling by William Clinger, R. Kent Dybvig, Matthew Flatt, and Marc Feeley
- Richard Kelsey's 1996 proposal
- Kent Pitman's history paper

- The Conditions chapter from the Common Lisp HyperSpec
- The Conditions chapter by Kent M. Pitman in *Common Lisp the Language, 2nd edition* by Guy L. Steele
- The Conditions chapter in the Dylan Reference Manual

Copyright

Editor: Francisco Solsona