Reference Manual for the

Elk Record Extension

Oliver Laumann

1. Introduction

The record extension to Elk allows Scheme applications to define record data types (similar to, although less powerful than, Common Lisp *structures*).

A record type can be instantiated to obtain a new record (a member of the given record type). Each record is a collection of named fields that can hold arbitrary Scheme objects. Records are first-class Scheme objects; they are members of the *record* data type that is disjoint from all other Scheme types. Record types are first-class objects as well; each record type is a member of the *record-type* Scheme data type.

The record extension provides facilities to define new record types, create instances of existing record types, define accessor and modifier functions to read and write the fields of records of a given type, and type predicates for the *record* and *record-type* data types.

In addition, the extension provides macros that simplify the definition of constructor, accessor and modifier functions for a newly defined record type.

2. Using the Record Extension

The record extension is loaded by evaluating

```
(require 'record)
```

in the interactive toplevel or in a Scheme program.

This causes the files *record.scm* and *record.o* to be loaded into the interpreter (*record.o* has to be linked with the interpreter on platforms that do not support dynamic loading of object files).

Loading the record extension causes the features *record* and *record.o* to be provided.

3. Record Types

(make-record-type *type-name fields*) **procedure**

make-record-type creates a new record type. *type-name* (a string or a symbol) is the name of the record type; it is used in the printed representation of the record type and of records belonging to the new type. If *type-name* is a symbol, it is converted into a string first. *fields* is a list of symbols naming the fields of a record of the new type. An error is signaled of the list contains duplicate names.

make-record-type returns the new record type (an object of the Scheme type *record-type*).

Example:

```
(define time-record
  (make-record-type 'time '(hours minutes seconds)))
```

(record-type? *obj***)** **procedure**

This type predicate returns #t if *obj* is a *record-type* object (i. e. the return value of a call to *make-record-type*), #f otherwise.

(record-type-name *rt***)** **procedure**

This procedure returns the type name (a string) of the record type *rt*, i. e. the *type-name* argument that was supplied to the call to *make-record-type* that created the record type *rt*.

(record-type-field-names *rt***)** **procedure**

record-type-field-names returns the list of field names associated with the record type *rt* (a list of symbols), i. e. the *fields* argument that was given in the call to *make-record-type* that created *rt*.

(record-constructor *rt fields***)** **procedure**

(record-constructor *rt***)** **procedure**

record-constructor returns a procedure for creating instances of the record type *rt*.

The returned procedure accepts as many arguments as there are symbols in the list *fields*; these arguments are used as the initial values of those fields in the newly created record instance. The values of any fields for which no initial value is specified (i. e. that are not present in *fields*) are undefined. If the *fields* argument is omitted, the field names that were given as an argument in the call to *make-record-type* that created the record type *rt* are used instead.

Example:

```
(define make-time
   (record-constructor time-record))
(define noon (make-time 12 0 0))

(define make-uninitialized-time
   (record-constructor time-record '()))
```

(record-predicate *rt***)** **procedure**

record-predicate returns a procedure for testing membership in the record type *rt*. The returned procedure accepts one argument and returns #t if the argument is a member of the record type *rt* (i. e. if it has been created by invoking a constructor returned by calling *record-constructor* with *rt* as an argument), #f otherwise.

(record-accessor *rt field***)** **procedure**

record-accessor returns a procedure for reading the value of the field named by *field* of a member of the record type *rt*. The returned procedure accepts one argument, which must be a record of the record type *rt*; it returns the current value of the specified field in that record. *field* must be a member of the list of field names that was supplied to the call to *make-record-type* that created *rt*.

Example:

```
(define time-hours
  (record-accessor time-record 'hours))

(define noon ((record-constructor time-record) 12 0 0))
(time-hours noon)   → 12
```

(record-modifier *rt field***)** **procedure**

record-modifier returns a procedure for writing the value of the field named by *field* of a member of the record type *rt*. The returned procedure accepts two arguments: a record of the record type *rt* and an arbitrary object; it stores the given object into the specified field in that record and returns the previous value of the field. *field* must be a member of the list of field names that was supplied to the call to *make-record-type* that created *rt*.

Example

```
(define set-time-hours!
  (record-modifier time-record 'hours))
```

(describe-record-type *rt***)** **procedure**

This procedure prints the names of the fields associated with the record type *rt*; it is automatically invoked by the standard *describe* procedure of Elk if *describe* is invoked with a record type.

4. Records

(record? *obj***)** **procedure**

This type predicate returns #t if *obj* is an object of type *record* (i. e. the return value of a call to a record constructor of any record type), #f otherwise.

(record-type-descriptor *record***)** **procedure**

This procedure returns the record type representing the type of the given record. The returned record type object is equal (in the sense of *eq?*) to the record type argument that was passed to *record-constructor* in the call that created the constructor procedure that created *record*.

Example: evaluating the expression

```
((record-predicate (record-type-descriptor r)) r)
```

always yields #t for any given record *r*.

(record-values *record***)** **procedure**

record-values returns the current contents of the fields of *record* as a vector. The *n*th element of the vector corresponds to the field with the name given as the *n*th element of the *fields* argument in the call to *make-record-type* that created the type to which *record* belongs.

The returned vector is not a copy of the actual fields; i. e. modifying the contents of the vector directly writes the corresponding fields of the record.

(describe-record *record***)** **procedure**

This procedure prints the names and current values of the fields of the given record; it is automatically invoked by the standard *describe* procedure of Elk if *describe* is invoked with a record.

5. Convenience Macros

The macros described in this section are loaded by evaluating

```
(require 'recordutil)
```

after having loaded the record extension. This causes the file *recordutil.scm* to be loaded and defines the feature *recordutil*.

(define-record-type *name fields***)** **syntax**

This macro defines a variable *<name>-record*, invokes the procedure *make-record-type* with the given *name* and *fields*, and assigns the result to this variable. In addition, *define-record-type* defines a type predicate for the new record type as *<name>−record?* and a constructor function as *make−<name>−record*. The constructor function accepts no arguments and returns an uninitialized record of the newly defined record type.

Example:

```
(require 'record)
(require 'recordutil)

(define-record-type
  time (hours minutes seconds))

(record-type? time-record)   → #t

(define t (make-time-record))

(time-record? t)   → #t
```

(define-record-accessors *rt***)** **syntax**

(define-record-modifiers *rt***)** **syntax**

The macro *define-record-accessors* (*define-record-modifiers*) defines accessor (modifier) functions for the fields of the record type *rt*. For each field named *field*, *define-record-accessors* (*define-record-modifiers*) defines a function *<name>−<field>* (*set−<name>−<field>!*), where *name* is the type name of the given record type. Each of the functions is the result of a call to *record-accessor* (*record-modifier*) as described above, with the arguments *rt* and the name of the field.

Example:

```
(define-record-type time (hours minutes seconds))
(define-record-modifiers time-record)

(define noon (make-time-record))
(set-time-hours! noon 12)
(set-time-minutes! noon 0)
(set-time-seconds! noon 0)

(define-record-accessors time-record)

(time-hours noon)   → 12
```

Index

Table of Contents