Records

Kent Dybvig
May 2005

Outline

Goals/Issues

Syntactic interface

Procedural interface

Open questions

Goals

Creation of distinct, structured types

Convenient high-level syntax

Portable readers, printers, inspectors, interpreters

Issues

Inheritence

Generativity

Naming conventions

Control over mutability

Control over printed representation

Read/write syntax for record instances

Security

Two interfaces

High-level syntactic interface

- `define-record` definition
- produces constructor, predicate, accessors, mutators

Low-level procedural interface

- `make-record-type` procedure
- returns a new record-type descriptor (RTD) . . .
- . . . from which can create constructor, predicate, etc.

Reflection: obtaining RTD from name or record instance

Syntax: Syntactic Interface

New production for *definition*:

$$
\begin{array}{lcl}
\mathit{definition} & \longrightarrow & \texttt{(define-record}\ \mathit{name}\ \texttt{(}\mathit{fld}_1{}^{*}\texttt{)} \\
 & & \quad \texttt{((}\mathit{fld}_2\ \mathit{init}\texttt{)}^{*}\texttt{)} \\
 & & \quad \texttt{(}\mathit{opt}^{*}\texttt{))} \\
 & | & \texttt{(define-record}\ \mathit{name}\ \mathit{parent\text{-}name}\ \texttt{(}\mathit{fld}_1{}^{*}\texttt{)} \\
 & & \quad \texttt{((}\mathit{fld}_2\ \mathit{init}\texttt{)}^{*}\texttt{)} \\
 & & \quad \texttt{(}\mathit{opt}^{*}\texttt{))}
\end{array}
$$

Notes:

1. *name* and *parent-name* are identifiers
2. *parent-name* must be a record name
3. $\mathit{fld}_1{}^{*}$ are initialized by constructor arguments
4. $\mathit{fld}_2{}^{*}$ are initialized by *init* expressions
5. $\texttt{((}\mathit{fld}_2\ \mathit{init}\texttt{)}^{*}\texttt{)}$ may be omitted
6. $\texttt{(}\mathit{opt}^{*}\texttt{)}$ may be omitted

Syntax: Fields

$$\begin{array}{lcl} \mathit{fld} & \longrightarrow & \mathit{field\text{-}name} \\ & \mid & (\,\mathit{class}\ \ \mathit{field\text{-}name}\,) \\ \mathit{field\text{-}name} & \longrightarrow & \mathit{identifier} \\ \mathit{class} & \longrightarrow & \texttt{mutable} \mid \texttt{immutable} \end{array}$$

Notes:

1. fields are mutable by default

Syntax: Options

$$
\begin{array}{lcl}
\mathit{opt} & \longrightarrow & \texttt{(constructor } \mathit{identifier}\texttt{)} \\
 & | & \texttt{(predicate } \mathit{id}\texttt{)} \\
 & | & \texttt{(prefix } \mathit{string}\texttt{)}
\end{array}
$$

Notes:

1. prefix is is used for accessors, mutators
2. could extend to allow finer control

Products

Definition of record named R with fields F `...` produces:

```
(begin
  (define-expand-time-binding R unspecified)
  (define make-R constructor-expr)
  (define R? predicate-expr)
  (define R-F accessor-expr)
  ...
  (define R-F-set! mutator-expr)
  ...)
```

Notes:

1. `make-`R replaced with $constructor$ if specified
2. R? replaced with $predicate$ if specified
3. R replaced by $prefix$ in accessors/mutators if specified
4. accessors/mutators not produced for parent fields

Reflection

New production for $expression$:

$$expression \quad \longrightarrow \quad \texttt{(type-descriptor } record\text{-}name\texttt{)}$$

Notes:

1. evaluates to a record-type descriptor (rtd)

Generativity

Created at run-time by default

Non-generativity if unique identifier specified in syntax

(define-record #{foo |a5nY+Q+YH$A?\\%|} (*field* ...))

Error is signaled if two nongenerative records have different characteristics

Printing

Record instances are printed with the following syntax

`#[`uid $field$ `...]`

May override with `record-writer` procedure:

`(record-writer` rtd $proc$`)`

$proc$ must take three arguments:

1. r, the record
2. p, an output port
3. wr, a procedure

Output should be produced to p

wr should be used for recursive `writes`

Example I

```
(define-record point (x y))

(define square (lambda (x) (* x x)))
(define point-disp
  (lambda (p1 p2)
    (sqrt (+ (square (- (point-x p1) (point-x p2)))
             (square (- (point-y p1) (point-y p2)))))))

(define base-disp
  (lambda (p)
    (point-disp (make-point 0 0) p)))

(base-disp (make-point 3 4)))  ⇒  5
```

Example II

```
(module A (point-disp)
  (define-record #{point |%E~s$5D<xO$l%\\%|} (x y))
  (define square (lambda (x) (* x x)))
  (define point-disp
    (lambda (p1 p2)
      (sqrt (+ (square (- (point-x p1) (point-x p2)))
               (square (- (point-y p1) (point-y p2))))))))

(module B (base-disp)
  (define-record #{point |%E~s$5D<xO$l%\\%|} (x y))
  (import A)
  (define base-disp
    (lambda (p)
      (point-disp (make-point 0 0) p))))

(let ()
  (import B)
  (define-record #{point |%E~s$5D<xO$l%\\%|} (x y))
  (base-disp (make-point 3 4)))  ⇒  5
```

Example III

```
> (define-record point (x y))
> (point 3 4)
#[#{point |%E~s$5Q<x5$l%\\%|} 3 4]
> '#[#{point |%E~s$5Q<x5$l%\\%|} 3 4]
#[#{point |%E~s$5Q<x5$l%\\%|} 3 4]
> (point-x '#[#{point |%E~s$5Q<x5$l%\\%|} 3 4])
3
> (record-writer (type-descriptor point)
    (lambda (x p wr)
      (display "<" p)
      (write (point-x x))
      (display "," p)
      (write (point-y x))
      (display ">" p)))
> (point 3 4)
<3,4>
> (point-x '#[#{point |%E~s$5Q<x5$l%\\%|} 3 4])
3
```

Procedural Interface

`(make-record-type` *name fields*`)` $\Rightarrow$ *rtd*
`(make-record-type` *parent-rtd name fields*`)` $\Rightarrow$ *rtd*

`(record-constructor` *rtd*`)` $\Rightarrow$ *procedure*
`(record-predicate` *rtd*`)` $\Rightarrow$ *procedure*
`(record-field-accessor` *rtd field-id*`)` $\Rightarrow$ *procedure*
`(record-field-mutator` *rtd field-id*`)` $\Rightarrow$ *procedure*

$$
\begin{array}{lcl}
\textit{name} & \longrightarrow & \textit{string} \mid \textit{gensym} \\[1em]
\textit{fields} & \longrightarrow & (\,\textit{field}^{*}\,) \\[1em]
\textit{field} & \longrightarrow & \textit{symbol} \\
 & \mid & (\,\textit{class field-name}\,) \\[1em]
\textit{field-id} & \longrightarrow & \textit{symbol} \mid \textit{ordinal}
\end{array}
$$

Example

```
(define point (make-record-type "point" ’(x y)))
(define make-point (record-constructor point))
(define point? (record-predicate point))
(define point-x (record-field-accessor point ’x))
(define point-y (record-field-accessor point ’y))
(define point-x-set! (record-field-mutator point ’x))
(define point-y-set! (record-field-mutator point ’y))
```

Example

```
(define point (make-record-type "point" ’(x y)))
(define point2 (make-record-type point "point" ’(x y)))
(define make-point2 (record-constructor point2))
(define point2? (record-predicate point2))
(define point2-x (record-field-accessor point2 0))
(define point2-y (record-field-accessor point2 1))
(define point2-xx (record-field-accessor point2 2))
(define point2-yy (record-field-accessor point2 3))
```

Reflection

Can obtain rtd from record instance

`(record-type-descriptor` $instance$`)` $\Rightarrow$ rtd

Notes:

1. permits writing of portable printers, inspectors
2. capabilities of this rtd may be limited
   - could prohibit obtaining record constructor
   - could prohibit obtaining record predicate
   - record fields could be inaccessible, immutable (see next slide)

Record predicates

```
(record-type-descriptor? object)
(record? object)
(record? rtd object)
(record-field-accessible? rtd field-id)
(record-field-mutable? rtd field-id)
```

Record predicates

(`record-type-descriptor?` *object*)
(`record?` *object*)
(`record?` *rtd* *object*)
(`record-field-accessible?` *rtd* *field-id*)
(`record-field-mutable?` *rtd* *field-id*)

(`record-constructable?` *rtd*)
(`record-predicable?` *rtd*)

Open Issues

Closed (non-inheritable) record types

Naming individual accessors, mutators

Interface with module system

- `(co-export` id id `...)`

Subset options

No syntactic interface

No procedural interface

No non-generative record definitions

Unspecified read/print syntax

No control over printing