User-extensible sequences in Common Lisp

Christophe Rhodes

Goldsmiths, University of London

Wednesday 4th April
1 Introduction
   Motivation
   Sequences

2 Design
   Utility
   Incompatibility
   Implementability

3 Future Work
• Ever seen `foo-position-if` in code?
  • flexichain: `nb-elements`, `element*`
  • climacs: `size`, `buffer-position-if`
  • trees: `size`, `reduce`, `position`
  • rucksack: `p-length`, `p-replace`, `p-delete-if`
  • cxml: `dom:length`, `dom:item`

• Identify simple building blocks of sequence functionality, to make it easy to have full range of functions available.

• Validate the “programmable programming language” claim.
Motivation

- Ever seen `foo-position-if` in code?
  - `flexichain`: `nb-elements`, `element*`
  - `climacs`: `size`, `buffer-position-if`
  - `trees`: `size`, `reduce`, `position`
  - `rucksack`: `p-length`, `p-replace`, `p-delete-if`
  - `cxml`: `dom:length`, `dom:item`

- Identify simple building blocks of sequence functionality, to make it easy to have full range of functions available.
- Validate the “programmable programming language” claim.
Motivation

- Ever seen `foo-position-if` in code?
  - flexichain: `nb-elements`, `element*`
  - climacs: `size`, `buffer-position-if`
  - trees: `size`, `reduce`, `position`
  - rucksack: `p-length`, `p-replace`, `p-delete-if`
  - cxml: `dom:length`, `dom:item`

- Identify simple building blocks of sequence functionality, to make it easy to have full range of functions available.

- Validate the “programmable programming language” claim.
Experiment: can we get Common Lispers to agree on anything?
Data type: a finite ordered collection of elements. Sequence has a size (length) and elements are addressable by single-integer position.

Examples:

- linked list, vector
- doubly-linked-list, queue, gap buffer
- DOM node
- compiler basic blocks
- ...
Data type: a finite ordered collection of elements. Sequence has a size (length) and elements are addressable by single-integer position. Examples:

- linked list, vector
- doubly-linked-list, queue, gap buffer
- DOM node
- compiler basic blocks
- ...
Fundamentals I:

- length
- elt, (setf elt)
Sequences

Introduction
Motivation
Design
Utility
Incompatibility
Implementability
Future Work
Summary

Operations:

- count, count-if, count-if-not
- find{-if{-not}}, position{-if{-not}}
- sort, fill, map-into ...
- remove{-if{-not}}, delete{-if{-not}}
- remove-duplicates, delete-duplicates
- map, merge, coerce, make-sequence, concatenate
Operations:

- count, count-if, count-if-not
- find{,-if{,-not}}}, position{,-if{,-not}}}
- sort, fill, map-into ...
- remove{,-if{,-not}}}, delete{,-if{,-not}}}
- remove-duplicates, delete-duplicates
- map, merge, coerce, make-sequence, concatenate
Sequences

Operations:

- count, count-if, count-if-not
- find{,-if{,-not}},{, position{,-if{,-not}}}
- sort, fill, map-into ...
- remove{,-if{,-not}},{, delete{,-if{,-not}}}
- remove-duplicates, delete-duplicates
- map, merge, coerce, make-sequence, concatenate
Fundamentals I:
- length
- elt, (setf elt)

Fundamentals II:
- make-sequence-like (creation of new sequence)
- adjust-sequence (adjusting of existing sequence if possible)
Desiderata

- Usefulness
- Convenience
- Minimize incompatibility with existing standards
- Implementability
• Users may define subclasses of `cl:sequence`. To do so, they must also write methods on
  • `sequence:length`, `sequence:elt`, `(setf sequence:elt)`
  • `sequence:make-sequence-like`,
    `sequence:adjust-sequence`

That’s it! No more is *necessary*. Can then call standard Common Lisp functions.

• May also customize
  • Iteration: a set of coupled generic functions to specialize.
  • Existing CL sequence functions: generic function analogue in sequence package.
Example: Class definition

Implement a kons type, which is like a cons except

- only \texttt{kons} or \texttt{nil} in the \texttt{kdr}: no dotted pairs.
- a \texttt{kons} knows its length.

(defclass kons (sequence standard-object)
  ((length :reader sequence:length :initarg :length)
   (kar :accessor kar :initarg :kar)
   (kdr :accessor kdr :initarg :kdr :type (or kons null))))

(defun kons (kar kdr)
  (make-instance 'kons :kar kar :kdr kdr
    :length (1+ (length kdr))))
Implement a `kons` type, which is like a `cons` except

- only `kons` or `nil` in the `kdr`: no dotted pairs.
- a `kons` knows its length.

```
(defclass kons (sequence standard-object)
  ((length :reader sequence:length :initarg :length)
   (kar :accessor kar :initarg :kar)
   (kdr :accessor kdr :initarg :kdr :type (or kons null))))
```

```
(defun kons (kar kdr)
  (make-instance 'kons :kar kar :kdr kdr :length (1+ (length kdr))))
```

```
(defun (setf kdr) :after (new-value (k kons))
  (setf (slot-value k 'length) (1+ (length new-value))))
```

```
(defmethod (setf kdr) :after (new-value (k kons))
  (setf (slot-value k 'length) (1+ (length new-value))))
```
Example: Class definition

Implement a kons type, which is like a cons except

- only kons or nil in the kdr: no dotted pairs.
- a kons knows its length.

(defclass kons (sequence standard-object)
  ((length :reader sequence:length :initarg :length)
   (kar :accessor kar :initarg :kar)
   (kdr :accessor kdr :initarg :kdr :type (or kons null)))
)

(defmethod (setf kdr) :after (new-value (k kons))
  (setf (slot-value k 'length) (1+ (length new-value))))

(defun kons (kar kdr)
  (make-instance 'kons :kar kar :kdr kdr
      :length (1+ (length kdr))))
Implement a `kons` type, which is like a `cons` except

- only `kons` or `nil` in the `kdr`: no dotted pairs.
- a `kons` knows its length.

```
(defclass kons (sequence standard-object)
  ((length :reader sequence:length :initarg :length)
   (kar :accessor kar :initarg :kar)
   (kdr :accessor kdr :initarg :kdr :type (or kons null)))))
```

```
(defun kons (kar kdr)
  (make-instance 'kons :kar kar :kdr kdr
    :length (1+ (length kdr))))
```
With that class definition, `cl:length` (but nothing else) works. Get `cl:elt` and `(setf cl:elt)` working with

```
(defmethod sequence:elt ((k kons) n)
  (if (= n 0) (kar k) (elt (kdr k) (1- n))))
(defmethod (setf sequence:elt) (nv (k kons) n)
  (if (= n 0)
      (setf (kar k) nv)
      (setf (elt (kdr k) (1- n)) nv)))
```

This is enough to support iteration without changing the sequence structure: `fill`, `sort`, `every`, `nsubstitute`, `count`, `find`, `position`, a `loop` path...
With that class definition, `(cl:length)` (but nothing else) works. Get `(cl:elt)` and `(setf cl:elt)` working with

```lisp
(defmethod sequence:elt ((k kons) n)
  (if (= n 0) (kar k) (elt (kdr k) (1- n))))
(defmethod (setf sequence:elt) (nv (k kons) n)
  (if (= n 0)
      (setf (kar k) nv)
      (setf (elt (kdr k) (1- n)) nv)))
```

This is enough to support iteration without changing the sequence structure: `fill`, `sort`, `every`, `nsubstitute`, `count`, `find`, `position`, a `loop` path...
With that class definition, `cl:length` (but nothing else) works. Get `cl:elt` and `(setf cl:elt)` working with

```lisp
(defmethod sequence:elt ((k kons) n)
  (if (= n 0) (kar k) (elt (kdr k) (1- n))))
(defmethod (setf sequence:elt) (nv (k kons) n)
  (if (= n 0)
      (setf (kar k) nv)
      (setf (elt (kdr k) (1- n)) nv)))
```

This is enough to support iteration without changing the sequence structure: `fill`, `sort`, `every`, `nsubstitute`, `count`, `find`, `position`, a `loop` path...
Two distinct missing pieces:

- **make new sequences** (substitute, subseq, coerce...)
  
  (defmethod sequence:make-sequence-like
    ((k kons) length &key initial-contents initial-element)
    (unless initial-contents
      (setq initial-contents
        (make-list length :initial-element initial-element)))
    (reduce #'kons initial-contents :from-end t :initial-value nil))

- **alter existing sequences** (delete, delete-duplicates)
  
  (defmethod sequence:adjust-sequence
    ((k kons) length &key &allow-other-keys)
    (cond
      ((= length 0) nil)
      ((= length 1)
       (setf (slot-value k 'length) 1 (kdr k) nil) k)
      ((< length (length k))
       (setf (slot-value k 'length) length)
       (sequence:adjust-sequence (kdr k) (1- length))
       k)))

Now all sequence functionality works! (inefficiently. Iteration implemented by default as index-based, which will be $O(N^2)$ for `cons-like data structures.) Iteration protocol in paper can be customized to recover efficiency for particular data structures. Also allow for customization of individual sequence functions.
Now all sequence functionality works! (inefficiently. Iteration implemented by default as index-based, which will be $O(N^2)$ for `kons`-like data structures.)

Iteration protocol in paper can be customized to recover efficiency for particular data structures. Also allow for customization of individual sequence functions.
Now all sequence functionality works!
(inefficiently. Iteration implemented by default as index-based, which will be $O(N^2)$ for `cons`-like data structures.) Iteration protocol in paper can be customized to recover efficiency for particular data structures. Also allow for customization of individual sequence functions.
Only known incompatibility of the whole proposal with ANS is in `make-sequence`: see CDR 3 for gory details. The type `sequence` not specified as `(or list vector)`.

Some potential issues with user code:

```lisp
(defun foo (sequence)
  (etypecase sequence
    (list  ...)
    (vector  ...)))
```

but that code will continue to work on lists and vectors; it will just not work with arbitrary sequences.
Only known incompatibility of the whole proposal with ANS is in `make-sequence`: see CDR 3 for gory details. The type `sequence` not specified as `(or list vector)`. Some potential issues with user code:

```lisp
(defun foo (sequence)
  (etypecase sequence
    (list ...)
    (vector ...)))
```

but that code will continue to work on lists and vectors; it will just not work with arbitrary sequences.
SBCL implementation features:

- optimizing for unchanged performance of existing code
- trampoline strategy
- `cl:length` distinct from `sequence:length`
- defined MIT loop path
- minor modification to CLOS implementation
- more invasive modifications to type system knowledge

Why two packages?
Other possible implementations:

- **simple**: `cl:length eql` to `sequence:length`. Potentially pays cost of generic function dispatch (but this can be a small cost, and compiler macros can make this cost go away for arguments whose type is known at compile-time).

- **defadvice**: calls to `cl:length` wrapped by advice function, calling `sequence:length` if arg is extended sequence, otherwise calling original function. Potential problem with interfering compiler macros.

- **new CL package**: `new-cl:find`. OK but likely to run into trouble in corner cases, particularly in compiler macros or the type system; lack of interoperability with even generically-written third-party code.
Future Work

- Get proposal used (and implemented for other CL implementations)
- Sort out some issues: what to do about sequences with invariants that are potentially violated by `(setf elt)`?
- Collections (hash-tables): convenient to have unified framework, but don’t have established names to work with
- Work out other user-subclassable things. `function` and `stream` well served. `number`, `real`?
Resources:

- SBCL home page: http://www.sbcl.org/
- CDR 3: http://cdr.eurolisp.org/document/3

Extensible sequences: dragging CL into the 1990s.