1 The `flexichain` protocol

In this section, we describe the `flexichain` protocol, allowing client code to dynamically add elements to, and delete elements from a sequence (or chain) of such elements.

1.1 The concept of a position

A flexichain uses the concept of a "position", which has two different meanings in different contexts.

The first meaning is the position at which an element is located in the chain. In this context, the position must have a value between 0 and $l - 1$, where l is the length of the chain. This meaning is used when an element is to be deleted or when an element is accessed or replaced.

The second meaning is the position *between two elements* (or at one of the extreme ends of the chain). In this context, the position may have a value between 0 and the length of the chain inclusive. The position 0 means the beginning of the chain (before the first element, if any), and the position equal to the length means the end of the chain (after the last element if any). This meaning is used when an element is to be inserted.

1.2 Performance

Accessing and replacing an element are constant-time operations.

We guarantee linear average complexity of a sequence of insert and delete operations provided that the position of two successive operations in the sequence is bounded.

More specifically, the average complexity of an operation is proportional to the difference between the position of the operation and the position of the previous operation.

Here we actually consider distance modulo the length of the chain so that the distance between the last position and the first is 1. In particular, the longest possible distance between to operations is half of the number of elements in the chain.

The implementation will allocate more space than is necessary. Whenever space runs out, we allocate a bigger chunk of memory to hold the elements. The new chunk size will be lk, were l again is the length of the chain and k is a *constant factor*. We multiply rather than add, because we want to guarantee the linear average complexity of a sequence of insert and delete operations. Typically, k is somewhere between 1.5 and 2. The default value is $k = 1.5$.

We shrink the space whenever the length of the chain (the number of elements) is significantly smaller than the available space. By default, the definition of "significantly" is that the ratio of length to size must be less than $1/k^2$ in order for the space to be shrunk. Again, the new chunk size will be lk.

Using the default value of k, this means that the amount of wasted space can be as large as the length of the chain in the worst case, but for a sequence of insert operations, the average wasted space is only 25%.

Applications that store a number of elements that does not vary much, can choose a small value for the k to waste less space. Such applications will have to resize the space relatively rarely so performance will not be affected by small values of k.

The space will not shrink below a minimum size (default 5 elements). The reason for this is to avoid to many resize operations for small chains. It is probably not reasonable to use a value below around 5, since the bookkeeping information takes up at least this much space. This value is also used for the initial size of the chain. Applications that will typically store a large number of elements can choose a greater value for the minimum size. Doing so also improves performance since fewer resize operations will have to be executed.

1.3 Protocol classes and functions

Many names of operations in this section have a terminatin "`*`" which is meant to suggest a *spread* version of the operation. Later (in the `flexicursor` section) we give *nospread* versions of the operations.

⇒ `flexichain` [*Protocol Class*]

The protocol class for flexichains.

⇒ `:initial-contents` [*Initarg*]
⇒ `:element-type` [*Initarg*]
⇒ `:fill-element` [*Initarg*]
⇒ `:expand-factor` [*Initarg*]
⇒ `:min-size` [*Initarg*]

All instantiable subclasses of `flexichain` accept these initargs.

The `:initial-contents` initarg is a sequence (list, vector, string) of objects to be stored in the `flexichain` from the start.

The `:element-type` initarg determines the type of the elements of the `flexichain` (default is `t`).

The `:fill-element` initarg should be an object that is compatible with the `:element-type` initarg and will be used to fill unoccupied space in the chain (to help the garbage collector). The default value for this initarg will be supplied by the implementation according to the element-type given. The implementation will test `nil`, `0`, and `#\a`. If none of these values will work, the client must supply a value that is compatible with `:element-type`.

The `:expand-factor` initarg is used to determine the factor by which the available space will be multiplied whenever the space for the chain is full. Default value is 1.5.

The `:min-size` initarg determines the smallest space allocated to hold elements of the chain. Default value is 5. It is not reasonable to supply values smaller than 5.

The instance created by make-instance will have a length which is that of the sequence given by `:initial-contents` or 0 if no `:initial-contents` was given.

⇒ `standard-flexichain` [*Class*]

The standard instantiable subclass of `flexichain`.

⇒ `nb-elements` *chain* [*Generic Function*]

Return the number of elements in the flexichain *chain*.

⇒ `flexi-error` [*Error Condition*]

The base condition for all conditions that may be signaled by the operations on flexichains.

⇒ `flexi-position` [*Error Condition*]

This condition will be signaled by operations that require a position argument whenever that argument is out of range.

⇒ `insert*` *chain position object* [*Generic Function*]

Insert an object at *position* of the flexichain. If *position* is out of range (less than 0 or greater than the length of *chain*), the `flexi-position` condition will be signaled.

⇒ `delete*` *chain position* [*Generic Function*]

Delete an element at *position* of the flexichain. If *position* is out of range (less than 0 or greater than or equal to the length of *chain*), the `flexi-position` condition will be signaled.

⇒ `delete-elements*` *chain position n* [*Generic Function*]

Delete N elements at *position* of the flexichain. If *position* + N is out of range (less than 0 or greater than or equal to the length of *chain*, the `flexi-position-error` condition will be signaled, and nothing will be deleted. n can be negative, in which case elements will be deleted before *position*.

⇒ `element*` *chain position* [*Generic Function*]

Return the element at *position* of the *chain*. If *position* is out of range (less than 0 or greater than or equal to the length of *chain*), the `flexi-position` condition will be signaled.

⇒ `(setf element*)` *object chain position* [*Generic Function*]

Replace the element at *position* of *chain* by *object*. If *position* is out of range (less than 0 or greater than or equal to the length of *chain*), the `flexi-position` condition will be signaled.

1.4 Stack and queue operations

A `flexichain` can be used as a stack or as a queue with very good performance. In this section, we suggest a set of operations to facilitate such use.

⇒ `push-start` *chain object* [*Generic Function*]

Insert an object at the beginning of the chain.

⇒ `push-end` *chain object* [*Generic Function*]

Insert an object at the end of the chain.

⇒ `pop-start` *chain* [*Generic Function*]

Pop and return the element at the beginning of the `chain`

⇒ `pop-end` *chain* [*Generic Function*]

Pop and return the element at the end of the `chain`

⇒ `rotate` *chain* `&optional` *(n 1)* [*Generic Function*]

Rotate the elements of the `chain` so that the element that used to be at position n now is at position 0. With a negative value of n rotate the elements so that the element that used to be at position 0 now is at position n. When the magnitude of n is greater than the length of the `chain`, the operation wraps around so that it becomes equivalent to the same operation with a value of n modulo the length. When the length of the `chain` is less than 2, this function does nothing.

2 Implementation of the `flexichain` protocol

2.1 Representation

We keep elements in a vector treated as a circular gap buffer with two sentinel elements, one before the first element of the chain (with a position of -1), and one after the last element of the chain (with a position equal to the length of the chain). We use the word *position* to refer to the abstract position of an element in a flexichain, and the word *index* when we talk about indexes of the gap buffer used in the implementation. We say that an index i is *valid* if $0 \leq i < l$ where l is the size of the vector (the vector is never of size 0, so it is always the case that $l > 0$).

We use the term *extended element* to mean a user element or a sentinel.

We say that the vector is *full* when it contains as many extended elements as its length (i.e., the gap has a size of 0), and *empty* when it contains no user elements (and thus only the sentinels) (i.e., the gap is the size as the

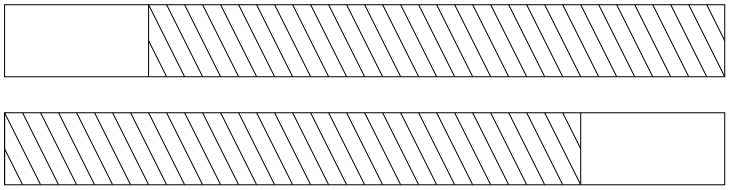

Figure 1: Gap and data are both contiguous

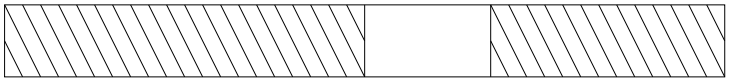

Figure 2: Data is not contiguous

vector minus 2).

There are three different possible configurations of the gap with respect to the data. Figure 1 shows the case where both the gap and the data are contiguous. Figure 2 shows the case where the data is not contiguous. Finally, figure 3 shows the case where the gap is not contiguous.

The implementation of a flexichain allows for the first element (i.e., the first sentinel with a position of -1) to be located at any valid index of the vector. For that reason, we need an index (called `data-start`) which always indicates the index of the first sentinel i.e.. `data-start` is always a valid index

A *positional index* is an index in the vector that corresponds to a position in the flexichain, and so is an index either of a user object or the index of the last sentinel.

We introduce two different indexes (always valid as well), `gap-start` and `gap-end`. The `gap-start` index is the first index of the gap. When the

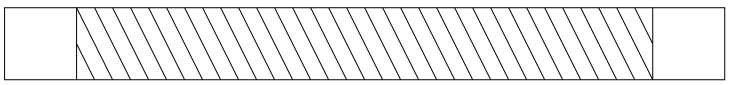

Figure 3: Gap is not contiguous

vector is not full, the `gap-start` is always an index containing no extended element, such that the previous index does contain an extended element. The `gap-end` index is the first index beyond the gap and is always the index of an extended element. When the vector is not full, the previous index does not contain an extended element. Notice that for certain configurations `gap-start` is smaller than `gap-end` and for certain other configurations, the reverse is true.

When the vector is full, `gap-start` and `gap-end` are always equal.

2.2 Computing and index from a position

Step one in inserting or deleting an element is to determine an index corresponding to the position. Here is how it is done: we compute a value s which is equal to `gap-start` if `gap-start` is greater than `data-start`. Otherwise s is equal to `gap-start` plus the size of the vector. The position is added to `data-start`, giving the value i. If i is greater than or equal to s, the size of the gap is added to i. Finally, if i is greater than or equal to the length of the vector, the length of the vector is subtracted from i (prove that the result is always a positional index). Call this final value of i the *hot spot*.

2.3 Moving the gap to the right place

After determining the index from a position, we need to determine whether the gap is in the right place. This is the case if and only if the hot spot is equal to `gap-end`. If that is not the case, we need to move the gap.

There is a case when it is particularly simple to move the gap, namely when the vector is full. In that case, we can just assign both `gap-start` and `gap-end` to the value of the hot spot.

There are two ways of getting `gap-end` to be equal to the hot spot either move everything to the left of the hot spot even further left, or everything to the right of the hot spot (including the hot spot itself) even further right. We always do the one that requires the fewest elements to be moved. One solution will require fewer than half the elements to be moved and the other one at least half.

Moving to the right will require that a number of elements equal to the

difference between `gap-start` and the hot spot to be moved, provided that `gap-start` is greater than the hot spot. If `gap-start` is smaller than the hot spot, it is that difference plus the size of the vector. We check whether that value is smaller than half of `nb-elements`.

Moving the elements requires one, two, or three calls to `replace`.

2.3.1 Moving elements to the left

Let us first consider the case of moving elements to the left.

Case 1: If the entire contiguous gap is to the left of the hot spot (as in the upper half of figure 1 or as in figure 2 with the hot spot to the right of the gap), a single call is required.

Case 2:A single call is also required if the highest valid index is inside the gap (as in the lower part of figure 1 and in figure 3) provided that the number of elements to be moved is no greater than the part of the gap that is flush right in the vector.

Case 3: Two calls are needed if the highest valid index is inside the gap (as in the lower part of figure 1 and in figure 3), but the number of elements to be moved is greater than the part of the gap that is flush right in the vector. The first call will fill the part of the gap that is flush right in the vector, giving the situation of the upper half of figure 1. The second call will be as in case 1 above.

Case 4: Two calls are also needed if the data is not contiguous (as in figure 2) and the entire contiguous gap is to the right of the hot spot, but the number of elements to the left of the hot spot (i.e., the index of the hot spot before the move) is no greater than the size of the gap. The first call will move everything to the right of the gap so that the gap will be flush right as in case 2 above. The second call will move the remaining elements.

Case 5: Three calls are needed if the data is not contiguous (as in figure 2) and the entire contiguous gap is to the right of the hot spot, but the number of elements to the left of the hot spot (i.e., the index of the hot spot before the move) is greater than the size of the gap. The first call will move the gap flush right, creating the situation of case 3 above (which then requires another two calls).

2.3.2 Moving elements to the right

Let us now consider moving elements to the right.

Case 1: If the entire contiguous gap is to the right of the hot spot (as in the lower half of figure 1 or as in figure 2 with the hot spot to the left of the gap), a single call is required.

Case 2:A single call is also required if the index 0 is inside the gap (as in the higher part of figure 1 and in figure 3) provided that the number of elements to be moved is no greater than the part of the gap that is flush left in the vector.

Case 3: Two calls are needed if index 0 is inside the gap (as in the lower part of figure 1 and in figure 3), but the number of elements to be moved is greater than the part of the gap that is flush left in the vector. The first call will fill the part of the gap that is flush left in the vector, giving the situation of the lower half of figure 1. The second call will be as in case 1 above.

Case 4: Two calls are also needed if the data is not contiguous (as in figure 2) and the entire contiguous gap is to the left of the hot spot, but the number of elements to the right of the hot spot (i.e., the index of the hot spot before the move) is no greater than the size of the gap. The first call will move everything to the left of the gap so that the gap will be flush left as in case 2 above. The second call will move the remaining elements.

Case 5: Three calls are needed if the data is not contiguous (as in figure 2) and the entire contiguous gap is to the left of the hot spot, but the number of elements to the right of the hot spot is greater than the size of the gap. The first call will move the gap flush left, creating the situation of case 3 above (which then requires another two calls).

2.4 Increasing the size of the vector

We increase the size of the vector whenever it is full and another element needs to be added.

When this call is made, `gap-start` and `gap-end` have the same value. We must preserve the position of the gap in the new vector.

A new vector with the size of the number of required elements multiplied by

`size-multiplier` is first allocated.

Next, we copy (using a single call to `replace`) all elements before the gap to the start of the new vector. Then we copy (using another single call to `replace`) all elements after the gap to the end of the new vector. The value of `gap-end` is incremented by the difference in size of the two vectors, as is `data-start` if it was greater than or equal to `gap-end`.

2.5 Decreasing the size of the vector

Again, a new vector with the size of the number of required elements multiplied by `size-multiplier` is first allocated.

Next, we copy (using a single call to `replace`) all elements before the gap to the start of the new vector. Then we copy (using another single call to `replace`) all elements after the gap to the end of the new vector. The value of `gap-end` is decremented by the difference in size of the two vectors, as is `data-start` if it was greater than or equal to `gap-end`.

2.6 Inserting an object

The insertion operation is given a position. The semantics of the insertion operation require that all elements having a position greater than or equal to the one given as argument to the insertion operation be "moved to the right" i.e., that they have their positions incremented by one.

After moving the hot spot to the right place, the value of `gap-end` is the index corresponding to the position supplied by the call. It should be noted that the same index will result from a position of 0 and from a position equal to the current length of the chain.

But first, we need to make sure the vector is not full. If it is, we call the function to increase its size.

We place the object to be inserted at the index of `gap-start` and then increment `gap-start`. If this operation gives a `gap-start` equal to the size of the vector, then it is set to 0.

2.7 Deleting an element

After moving the hot spot to the right place, we need to delete the element at `gap-end`. We do this by replacing it by the `fill-element` so as to avoid holding on to it in case it is no longer referenced. Then we increment `gap-end`.

Finally, we check whether the size of the vector should be decreased.

2.8 Stack and queue operations

The stack and queue operations are implemented very efficiently. The `push` and `pop` operations simply call the corresponding `insert` and `delete` operations.

The `rotate` operation deletes from one end of the chain and inserts on the other.

3 The `flexicursor` protocol

A *cursorchain* is like a flexichain, but it also keeps around a bunch of "flexicursors".

3.1 The concept of a flexicursor

A flexicursor is an object that corresponds to a position between two elements of the chain. There are two types of flexicursors, *left-sticky* and *right-sticky*. The difference between the two is the way they behave when an object is inserted at corresponding position. When an object is inserted at the position corresponding to a left-sticky flexicursor, this cursor will be positioned *before* the newly inserted object, i.e., the cursor "sticks" to the element on its left. When an object is inserted at the position corresponding to a right-sticky flexicursor, this cursor will be positioned *after* the newly inserted object, i.e., the cursor "sticks" to the element on its right.

Whenever an object is inserted before the position of a cursor, the position of the cursor will be incremented. Conversely, whenever an element is

deleted from a position below that of a cursor, the position of the cursor is decremented.

3.2 Mixing `flexicursor` and `flexichain` operations

The user can freely mix editing operations from the `flexicursor` and the `flexichain` protocol. When an editing operation from the `flexichain` protocol is used on an `cursorchain` object, the cursors of the `cursorchain` object are updated accordingly.

3.3 Performance

There can be a very large number of cursors in a chain without any negative impact on performance. In particular, a sequence of insert operations is not affected by the number of cursors of the chain. For insert operations, we maintain the complexity proportional to the distance between two consecutive positions.

A delete operation takes time proportional to the number of left-sticky cursors to the right of the element to delete plus the number of right-sticky cursors to the left of it.

The only bad case is thus a delete operation of an element with an unbounded number of cursors sticking to it.

3.4 Protocol classes and functions

⇒ `cursorchain` [*Protocol Class*]

This is a subclass of `flexichain`.

⇒ `standard-cursorchain` [*Class*]

The standard instantiable subclass of `cursorchain`.

⇒ `flexicursor` [*Protocol Class*]

The protocol class for all flexicursors.

⇒ `chain` [*Initarg*]

This initarg determines the cursorchain with which the cursor is associated.

⇒ `standard-flexicursor` [*Class*]

The standard instantiable subclass of `flexicursor`.

⇒ `left-sticky-flexicursor` [*Class*]

The standard instantiable class for left-sticky flexicursors. It is a subclass of standard-flexicursor.

⇒ `right-sticky-flexicursor` [*Class*]

The standard instantiable class for right-sticky flexicursors. It is a subclass of standard-flexicursor.

⇒ `chain` *cursor* [*Generic Function*]

Return the underlying cursorchain of the flexicursor given as argument.

⇒ `clone-cursor` *cursor* [*Generic Function*]

Create a cursor that is initially at the same location as the one given as argument.

⇒ `flexi-position-error` [*Error Condition*]

This condition is signaled whenever an attempt is made to use position outside of the range of valid positions.

⇒ `cursor-pos` *cursor* [*Generic Function*]

Return the position of the cursor.

⇒ `(setf cursor-pos)` *position cursor* [*Generic Function*]

Set the position of the cursor. If the new position of the cursor is before the first position or after the last position of the chain, the condition `flexi-position-error` is signaled.

⇒ `at-beginning-p` *cursor* [*Generic Function*]

Return true if the cursor is at the beginning of the chain (i.e., if it has a position of 0). This operation is guaranteed to be executed in O(1) time.

⇒ `at-beginning` [*Error Condition*]

This condition is signaled whenever an attempt is made to move a cursor beyond the beginning of the chain.

⇒ `at-end-p` *cursor* [*Generic Function*]

Return true if the cursor is at the end of the chain (i.e., if it has a position equal to the length of the chain). This operation is guaranteed to be executed in O(1) time.

⇒ `at-end` [*Error Condition*]

This condition is signaled whenever an attempt is made to move a cursor beyond the end of the chain.

⇒ `incompatible-object-type` [*Error Condition*]

This condition is signaled whenever an attempt is made to insert an object of an incompatible type into an chain.

⇒ `insert` *cursor object* [*Generic Function*]

Insert an object at the position corresponding to that of the cursor. All cursors located at positions greater than the one corresponding to the cursor given as argument, as well as left-sticky cursors (possibly including the one given as argument) located at the same position as the one given as argument will have their positions incremented by one. Other cursors are unaffected.

If the type of the object does not match the type accepted by the underlying chain, the `incompatible-object-type` condition is signaled.

⇒ `insert-sequence` *cursor sequence* [*Generic Function*]

The effect is the same as if each object of the sequence were inserted using the `insert` generic function.

⇒ `delete<` *cursor* `&optional` *(n 1)* [*Generic Function*]

Delete n elements before the cursor.

⇒ `delete>` *cursor* `&optional` *(n 1)* [*Generic Function*]

Delete n elements after the cursor. ...

A sequence of insert and delete operations is guaranteed to be efficient if the positions of successive operations are not too far apart as measured by the shortest distance of the chain viewed as a circular list. Thus, the beginning and the end of the chain are considered close.

⇒ `with-editing-operations` *cursor* `&body` *body* [*Macro*]

This macro can be used to group a bunch of editing operations (insert, delete) into a body. The sequence remains locked for the duration of invocation. Other cursors of the underlying chain are updated only after the last operation has been completed, thus making it more efficient to use this macro than to use individual editing operations.

⇒ `element<` *cursor* [*Generic Function*]

Return the element immediately before the cursor. If the cursor is at the beginning, an at-beginning condition will be signaled.

⇒ `(setf element<)` *object cursor* [*Generic Function*]

Replace the element immediately before the cursor by the object given as argument. If the cursor is at the beginning, an at-beginning condition will be signaled.

⇒ `element>` *cursor* [*Generic Function*]

Return the element immediately after the cursor. If the cursor is at the end, an at-end condition will be signaled.

⇒ `(setf element>)` *object cursor* [*Generic Function*]

Replace the element immediately after the cursor by the object given as argument. If the cursor is at the end, an at-end condition will be signaled.

4 Implementation of the `flexicursor` protocol

Cursors are stored as lists of weak references so that they can be recycled when no longer referenced by client code. A vector that parallels the one holding elements of the flexichain holds per-element lists of cursors that stick to that element.

A cursor contains its *index in the vector* as opposed to its *position in the sequence*. This method avoids most updates of cursors at each insert and delete operation. Most cursors need only be updated whenever the gap moves. For left-sticky cursors, we store the index of $p - 1$, where p is the position of the cursor. For right-sticky cursors, we store p itself.

After a delete operation, cursors with indexes equal to the old value of `gap-end` need to be updated. Right-sticky cursors will be attached to the index corresponding to the new value of `gap-end`, whereas left-sticky cursors

get attached to the position immediately preceding `gap-start`.

Insert operations do not affect cursors at all.

Mixing of `flexicursor` and `flexichain` editing operations is possible thanks to an internal protocol for moving the gap. The `flexicursor` code uses :before, :after, and :around methods on the `flexichain` editing operations as well as on the code for moving the gap to update the cursors accordingly. This way, a `flexicursor` editing operation translates directly to a `flexichain` editing operation with no extra code.