Program Design & Data Structures (course 1DL201)
Uppsala University – Autumn 2012 / Spring 2013
Homework 1: Flexible Vectors

Prepared by Pierre Flener

Lab: Tuesday 18 December 2012
Submission Deadline: 18:00:00 on Thursday 17 January 2013
Lesson: Friday 25 January 2013
Resubmission Deadline: 18:00:00 on date to be determined

Vectors

The vector (see slides 60–63 of course topic 3 or `http://www.standardml.org/Basis/vector.html`) is a very useful data structure, as one can access any element in constant time, provided its index is known. For example, if one has a vector `v` of type `int vector`, then one can use the expressions

```
Vector.sub (v, 997)
Vector.update (v, 3, x)
```

in order to access the value of element `v[997]` and update the value of element `v[3]` to the value of `x`, respectively. However, vectors require that the memory needed to store all the elements is allocated contiguously and at once, as well as (for static vectors) that the maximum vector size is known in advance. A rather important consequence thereof is that if for some reason one uses only a few elements in a large vector, then one wastes a lot of memory. For example, assume one declares a vector of a million elements, but for some reason only the values at the indices 3 and 997 are used in a given run of a program; the declaration

```
val v = Vector.tabulate(1000000, fn _ => 5)
```

then allocates contiguous memory for a vector `v` of one million integers (at indices 0 to 999999) and initialises them all to the default value `5`, but only a very tiny percentage of this memory is used, which is very wasteful.

Flexible Vectors

This is where the concept of flexible vector comes in. A *flexible vector* is like a normal vector as far as how one uses it is concerned: one can access and update the element at a given index as in a normal vector. ***Functionally***, there is thus no difference, but ***computationally*** there are differences in resource consumption. The first difference is that if one needs a vector of a million elements, say, but really uses only a small number thereof, then the flexible vector requires a much smaller amount of memory. This flexibility comes at a price though: the second difference is that the access to an element of known index is not constant-time any more.

Representing Flexible Vectors

The size of a flexible vector is not a defined concept. One can represent a flexible vector as a list of chunks. Each *chunk* has a small fixed-size vector of the same type of objects as the flexible vector, as well as the starting index within the flexible vector of the chunk. The polymorphic `'a flexVector` datatype has the following definition:

```
datatype 'a flexVector = Flex of int * 'a * 'a chunk list
```

where the first tuple component (the integer) is the chunk size and the second component (of type `'a`) is the default value for chunk elements when creating a new chunk. The polymorphic `chunk` datatype has the following definition:

```
datatype 'a chunk = Chunk of int * 'a vector
```

where the first tuple component (the integer) is the starting index of the second component (the vector) within the flexible vector. Chunks in the list of a flexible vector `f` must be ordered increasingly by their starting indices, ***which must be integer multiples of the chunk size of*** `f`.

Example 1 A flexible vector `f` with two chunks of $s = 10$ elements and with currently only the elements at indices 3 and 997 being used is depicted in Figure 1 (careful, the default value is not shown). We have only allocated memory for 20 elements of the flexible vector although it seems to have 1000 elements. If we need to store a value at index 6, then we first go to the initial chunk (which holds the elements at indices 0 to 9) and then store that value at index 6 in that chunk. If we need to update the value at index 997, then we first search for a chunk with starting index ℓ such that $\ell \leq 997 < \ell + s$ and then store the new value at index 7 in the vector of that chunk; in this case, we have such a chunk, namely the last one, with $\ell = 990$. If we need to store a value at index 556, then we first create a new chunk for indices 550 to 559, link it in between the two existing chunks, and then store that value at index 6 in that new chunk.

Work To Be Done

Implement the following functions in a file called `flexVector.sml`, making sure that they pass the training test cases at `http://www.it.uu.se/edu/course/homepage/pkd/ht12/assignments/assignment1-training.sml`, which are ***not*** be included:

- `vector (s, d)` returns the empty flexible vector of integer chunk size `s` and default value `d`, under the pre-condition `s` > 0;
- `sub (f, i)` returns `f[i]`, which is possibly the default value of flexible vector `f`, under the pre-condition `i` ≥ 0;
- `update (f, i, x)` returns a new flexible vector, identical to `f`, except that the element at index `i` is set to `x`, under the pre-condition `i` ≥ 0.

Do ***not*** write any code for catching exceptions. In a separate report in PDF format, give an explicit reasoning (including recurrences and their closed forms) establishing the average-case and worst-case runtime complexities of your functions, assuming that the chunk size `s` is a constant.

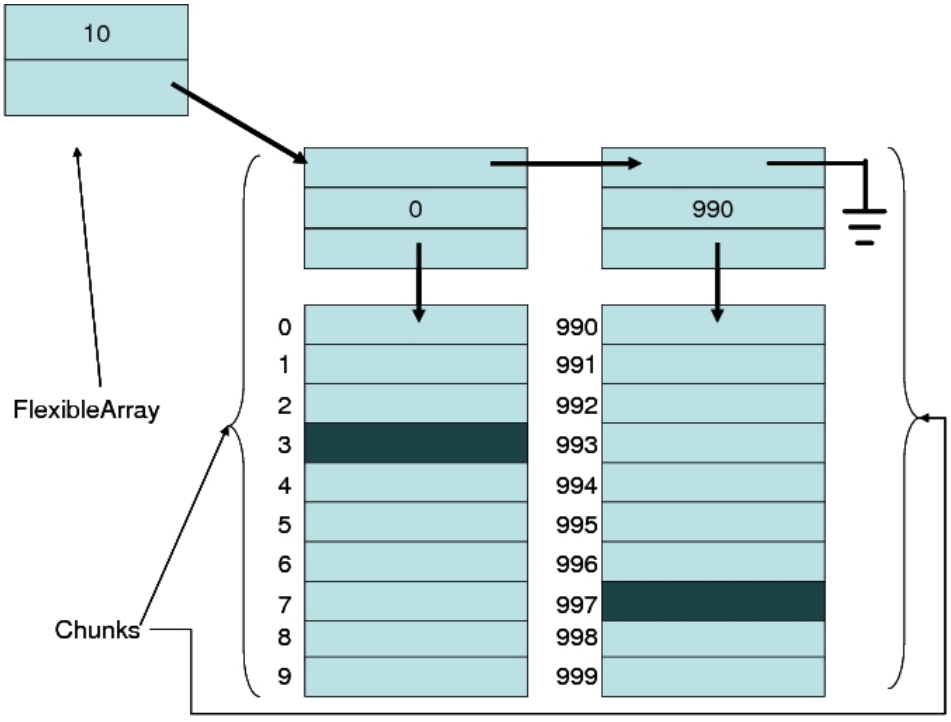

Figure 1: A flexible vector with chunks of size $s = 10$.

Grading

Your solution is graded with 0 to 100 points in the following way:

1. If your solution was submitted by the deadline, your program loads under Poly/ML version 5.5.0 (Uppsala modified 1) and passes ***all*** the training test cases, and your solution is deemed to be a serious attempt at implementing, commenting (under at least the coding convention), ***and*** analysing ***all*** the requested functions, then you get 20 points; otherwise (including when no solution was submitted by the submission deadline), you get a U grade for this homework (even if an insufficiently commented program is actually correct). **Hint:** Test your submitted code in a freshly started ML interpreter in order to make sure your code does not work because of some old data or functions lying around in the interpreter.

2. Your program is run on an unspecified number of orthogonal grading test cases that satisfy all pre-conditions but ***also*** check boundary conditions. For each fully correct test result, you get a suitable amount of points, the total being 40 points. We reserve the right to run these tests automatically, so be careful to match exactly the imposed file names, function names, and argument orders.

3. Your program is graded for style and comments (including function specifications, datatype representation conventions and invariants, as well as recursion variants), provided it does not fail on all the grading test cases. This covers 20 points.

4. Your complexity analysis is graded for correctness of results and explicitness of reasoning. This covers the remaining 20 points.

We assume that by submitting a solution you are certifying that it is solely the work of your group, except where explicitly attributed otherwise. We reserve the right to use plagiarism detection tools and point out that they are extremely powerful.