Lecture Notes on Unbounded Arrays

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 10
June 3, 2014

1 Introduction

Most lectures so far had topics related to all three major categories of learning goals for the course: computational thinking, algorithms, and programming. The same is true for this lecture. With respect to algorithms, we introduce *unbounded arrays* and operations on them. Analyzing them requires *amortized analysis*, a particular way to reason about sequences of operations on data structures. We also briefly talk about again about *data structure invariants* and *interfaces*, which are crucial computational thinking concepts.

Note For Summer 2014: This interface and implementation of unbounded arrays described here is a bit different than the one in class and posted to `http://www.cs.cmu.edu/~rjsimmon/15122-m14/lec/10-ubarrays/arr.c0`. The most important differences are:

- Our arrays were named `arr_`, these are named `uba_`.
- We gave an initial *size* to our array (and arrays are initialized in the usual C0 fashion like all other heap-allocated memory), the interface here starts arrays at size 0 and gives them an initial limit (a part of the implementation that shouldn't necessarily be available to the programmer!)
- Our data structure invariant required that the size be strictly less than the limit, which had the result that we always wasted one cell of the array. This changed amortized analysis a little bit, but the result is the same.

2 Unbounded Arrays

In the second homework assignment, you were asked to read in some files such as the *Collected Works of Shakespeare*, the *Scrabble Players Dictionary*, or anonymous tweets collected from Twitter. What kind of data structure do we want to use when we read the file? In later parts of the assignment we want to look up words, perhaps sort them, so it is natural to want to use an array of strings, each string constituting a word. A problem is that before we start reading we don't know how many words there will be in the file so we cannot allocate an array of the right size! One solution uses either a queue or a stack. We discussed this in Lectures 9 on queues and stacks. However, the array interface that we originally introduced in Lecture 8 wouldn't work, because it requires us to bound the size of the array. It would work, however, if we had *unbounded arrays*. While arrays are a language primitive, unbounded arrays are a *data structure* that we need to implement, and we can implement it as an extension of the array data structure.

Thinking about it abstractly, an unbounded array should be like an array in the sense that we can get and set the value of an arbitrary element via its index i. We should also be able to add a new element to the end of the array, and delete an element from the end of the array.

We use the unbounded array by creating an empty one before reading a file. Then we read words from the file, one by one, and add them to the end of the unbounded array. Meanwhile we can count the number of elements to know how many words we have read. We trust the data structure not to run out of space unless we hit some hard memory limit, which is unlikely for the kind of task we have in mind, given modern operating systems. When we have read the whole file the words will be in the unbounded array, in order, the first word at index 0, the second at index 1, etc.

The general implementation strategy is as follows. We maintain an array of a fixed length $limit$ and an internal index $size$ which tracks how many elements are actually used in the array. When we add a new element we increment $size$, when we remove an element we decrement $size$. The tricky issue is how to proceed when we are already at the limit and want to add another element. At that point, we allocate a new array with a larger limit and copy the elements we already have to the new array. For reasons we explain later in this lecture, every time we need to enlarge the array we *double* its size. Removing an element from the end is simple: we just decrement $size$. There are some issues to consider if we want to shrink the array, but this is optional.

3 An Interface to Unbounded Arrays

As usual when designing a data structure, we start by thinking about its interface. We must be able to create a new unbounded array, access its elements (both for reading and writing), and add or remove elements at the end. The elements of the array should be of arbitrary type (like ordinary arrays), but we cannot achieve this form of genericity in C0 at present. We will discuss ways to write generic code later in the course when we move to C. Instead, we just indicate this by defining and using a type name `elem` (here as `string`). These considerations lead us to the following interface:

```
typedef string elem;

/* Interface of unbounded arrays */

typedef struct uba_header* uba;

uba uba_new(int initial_limit)
//@requires initial_limit > 0;
  ;

int uba_size(uba L)               /* "\length(L)" */
//@ensures \result >= 0;
  ;

elem uba_get(uba L, int index)    /* "L[index]" */
//@requires 0 <= index && index < uba_size(L);
  ;

void uba_set(uba L, int index, elem e)  /* "L[index] = e" */
//@requires 0 <= index && index < uba_size(L);
  ;

void uba_add(uba L, elem e);      /* add e at the end of L */

elem uba_rem(uba L)               /* remove last element in L */
//@requires uba_size(L) > 0;
  ;
```

Contracts on interfaces are cumulative with respect to the contracts on the implementations: both are checked when a function is called through

its interface. Note that we do not mention `is_uba`, since this function is not exposed to the client. Client code should only ever be able to obtain valid unbounded arrays if it uses the interface, so preservation of the data structure invariants should be considered an *internal invariant* of the data structure implementation.

Please read over the interface carefully to make sure you understand all of its provisions. We would like all the specified operations to take only *constant time*, that is, $O(1)$. As we will see in the remainder of this lecture this is quite tricky and we have to make some intriguing qualifications in our statement of the asymptotic complexity.

Unfortunately, C (and, by association, C0) does not provide a way to enforce that clients do not incorrectly exploit details of the implementation of a data structure. Higher-level languages such as Java or ML have interfaces and data abstraction as one of their explicit design goals. In this course, the use of interfaces is a matter of programming discipline. As we discuss further data structures we generally focus on the interface first, before writing any code. This is because the interface often guides the selection of an implementation technique and the individual functions.

4 Implementing Unbounded Arrays

According to our implementation sketch, an unbounded array needs to track three forms of data: an integer *limit*, an integer *size* and an array of elements (e.g., strings). We can put these together in a `struct` with fields *limit*, *size* and *data* as the *fields* of the struct. It is declared with

```
struct uba_header {
  int limit;               /* 0 < limit */
  int size;                /* 0 <= size && size <= limit */
  elem[] data;             /* \length(data) == limit */
};
```

Also recall the line from the interface

```
typedef struct uba_header* uba;
```

which states that a `uba` is *a pointer to* a `struct uba_header`. Recall that structs can only be allocated on the heap (rather than stored in variables), so we always work with the addresses of structs. And addresses are the values of pointers.

There are some *data structure invariants* that we maintain, although they may be temporarily violated as the elements of the structure are manipulated at a low level. Generally, when we pass a pointer to the data structure or assign it to a variable we expect these invariants to hold. C0, however, has no intrinsic support for ensuring these invariants. Instead, our method is to define a function to test them and then verify adherence to the invariants in contracts as well as loop invariants and assertions. Here, the function `is_uba` serves that purpose. In previous lectures we had functions `is_queue` and `is_stack` that fulfill a similar role.

Generally, we would like contract functions like `is_uba` not to fail with a contract exception, but to return false if the data structure invariant is violated. However, since the lengths of arrays can only be checked in contracts (they may not be available when a program is compiled without `-d` to make computation as efficient as possible) we may have to use contracts to some extent even for functions whose intended use is only in contracts.

```
bool is_uba (uba L)
{
  if (L == NULL) return false;
  if (!(0 <= L->size)) return false;
  if (!(L->size <= L->limit)) return false;
  if (!(0 < L->limit)) return false;
  //@assert L->limit == \length(L->data);
  return true;
}
```

Note that we *must* check to make sure that `L != NULL` before checking any other fields, including `L->size` and `L->data` (i.e. `L->limit == \length(L->data)`) in order to make sure the pointer dereferences on L are safe. Safety of annotations and safety of contract functions is just as indispensable as safety in the rest of the code.

To create a new unbounded array, we allocate a `struct uba_header` and an array of a supplied initial limit.

```
uba uba_new (int initial_limit)
//@requires initial_limit > 0;
//@ensures is_uba(\result);
{
  uba L = alloc(struct uba_header);
  L->limit = initial_limit;
  L->size = 0;
```

```
  L->data = alloc_array(elem, L->limit);
  return L;
}
```

Getting and setting an element of an unbounded array is straightforward. However, we do have to verify that the array access is *in bounds*. This is stricter than checking that it is within the allocated array (below `limit`), because everything beyond the current size should be considered to be *undefined*. These array elements have not yet been added to the array, so reading or writing them is meaningless. We show only the operation of writing to an unbounded array, `uba_set`.

```
void uba_set(uba L, int index, elem e)
//@requires is_uba(L);
//@requires 0 <= index && index < L->size;
{
  L->data[index] = e;
  return;
}
```

More interesting is the operation of adding an element to the end of an unbounded array. For that we need a function to resize an unbounded array. This function takes an unbounded array L and a new limit `new_limit`. It is required that the new limit is strictly greater than the current size, to make sure we have enough room to preserve all current elements and one more for the next one to add. We also stipulate that the size does not change by stating `L->size == \old(L->size)` in the postcondition. In general, `\old(e)` in a postcondition evaluates e just after the function is called and before the body is executed. This allows us to refer to the state of memory when the function is called in the postcondition.

```
void uba_resize(uba L, int new_limit)
//@requires is_uba(L);
//@requires L->size < new_limit;
//@ensures is_uba(L);
//@ensures L->limit == new_limit && L->size == \old(L->size);
//@ensures L->size < L->limit;
{
  elem[] B = alloc_array(elem, new_limit);
  for (int i = 0; i < L->size; i++)
    //@loop_invariant 0 <= i && i <= L->size;
```

```
    {
      B[i] = L->data[i];
    }
  L->limit = new_limit;
  /* L->size remains unchanged */
  L->data = B;
  return;
}
```

Finally we are ready to write the function that adds an element to the end of an unbounded array. We first check whether there is room for another element and, if not, double the size of the underlying array of strings. The contract just states that the array is valid before and after the operation.

```
void uba_add(uba L, elem e)
//@requires is_uba(L);
//@ensures is_uba(L);
{
  if (L->size == L->limit) {
    /* Check for overflow */
    assert(L->limit <= int_max()/2);
    uba_resize(L, 2*L->limit);
  }

  //@assert L->size < L->limit;
  L->data[L->size] = e;
  L->size++;
  return;
}
```

We check that doubling the array size would not overflow and raise an assertion failure. Using `assert` as a statement instead of inside an annotation means that the assertion will always be checked, even if the code is compiled without `-d`. It will have the same effect as, for example, the `alloc_array` function when there is not enough memory to allocate the array.

We discuss how to remove an element from an array in section 6.

5 Amortized Analysis

It is easy to see that reading from or writing to an unbounded array at a given index is a constant-time operation. However, adding an element to an array is not. Most of the time it takes constant time $O(1)$, but when we have run out of space it take times $O(\mathsf{size})$ because we have to copy the old elements to the new underlying array. On the other hand, it doesn't seem to happen very often. Can we characterize this situation more precisely? This is the subject of *amortized analysis*.

In order to make the analysis as concrete as possible, we want to count the number of writes to an array, that is, the number of assignments $A[-] = -$ that are performed. Calling the operation to add a new element to an unbounded array an *insert*, we claim:

> *The worst-case cost of n insert operations into an unbounded array is $O(n)$.*

This statement is quite different from what we have done before, when we have analyzed the cost of a particular function call like `sort` or `binsearch`. Based on the common use of unbounded arrays, we should consider the cost of multiple operations together. Many other data structures introduced later in the course will be subject to a similar analysis.

How do we prove the above bound? A simple insert (when there is room in the array) requires a single write operation, so we count it as 1. Similarly, we count the act of copying one element from one array to another as 1 operation, because it requires one write operation. Now performing a sequence of inserts, starting with an empty array of, say, size 4 looks as follows.

call	op's	*size*	*limit*
`uba_add(L,"a")`	1	1	4
`uba_add(L,"b")`	1	2	4
`uba_add(L,"c")`	1	3	4
`uba_add(L,"d")`	1	4	4
`uba_add(L,"e")`	5	5	8
`uba_add(L,"f")`	1	6	8
`uba_add(L,"g")`	1	7	8
`uba_add(L,"h")`	1	8	8
`uba_add(L,"i")`	9	9	16

We have taken 4 extra operations when inserting `"e"` in order to copy `"a"` through `"d"`. Overall, we have performed 21 operations for inserting 9 elements. Would that be $O(n)$ by the time we had inserted n elements?

We approach this by giving us an overall budget of $c * n$ operations ("tokens") before we start to insert n elements. Every time we perform a write operation we spend a token. If we perform all n inserts without running out of tokens, we have achieved the desired amortized complexity.

One difficulty is to guess the right constant c. We already know that $c = 1$ or $c = 2$ will not be enough, because in the sequence above we must spend 21 tokens to insert 9 elements. Let's try $c = 3$, so we start with 27

tokens.

call	op's	tokens left	*size*	*limit*
`uba_add(L,"a")`	1	26	1	4
`uba_add(L,"b")`	1	25	2	4
`uba_add(L,"c")`	1	24	3	4
`uba_add(L,"d")`	1	23	4	4
`uba_add(L,"e")`	5	18	5	8
`uba_add(L,"f")`	1	17	6	8
`uba_add(L,"g")`	1	16	7	8
`uba_add(L,"h")`	1	15	8	8
`uba_add(L,"i")`	9	6	9	16

We see that we spend 4 tokens when adding `"e"` to copy `"a"` through `"d"`, and we add a new one for the insertion of `"e"` itself.

One of the insights of amortized analysis is that we don't need to know the number n of inserts ahead of time. In order to achieve the bound of $c * n$ operations, we simply allow each call to perform c operations. If it performs fewer, these remain in the budget and may be spent later! Let's go through the same sequence of calls again.

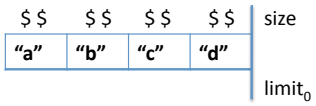
call	op's	allocated tokens	spent tokens	saved tokens	total saved tokens	*size*	*limit*
`uba_add(L,"a")`	1	3	1	2	2	1	4
`uba_add(L,"b")`	1	3	1	2	4	2	4
`uba_add(L,"c")`	1	3	1	2	6	3	4
`uba_add(L,"d")`	1	3	1	2	8	4	4
`uba_add(L,"e")`	5	3	5	-2	6	5	8
`uba_add(L,"f")`	1	3	1	2	8	6	8
`uba_add(L,"g")`	1	3	1	2	10	7	8
`uba_add(L,"h")`	1	3	1	2	12	8	8
`uba_add(L,"i")`	9	3	9	-6	6	9	16

The crucial property we need is that there are $k \geq 0$ tokens left just after we have doubled the size of the array. We think of this as an *invariant* of the computation: it should always be true, no matter how many strings we insert. In this example we reach 6 tokens after 5 inserts and again after 9 inserts.

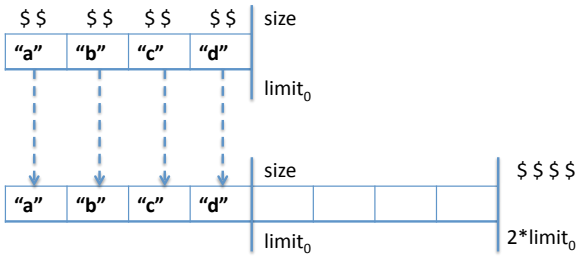
To prove this invariant, we must show that it holds the first time we have to double the size of the array, and that it is preserved by the operations.

When we create the array, we give it some initial limit $limit_0$. We run out of space, once we have inserted $limit_0$ tokens, arriving at the following

situation.

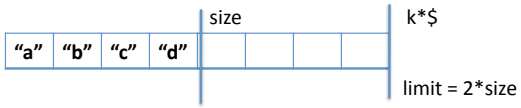


We have accrued $2 * limit_0$ tokens. We have to spend $limit_0$ of them to copy the elements so far, keeping $limit_0 > 0$ in the bank.

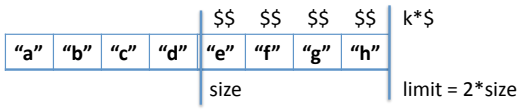


So the invariant holds the first time we double the size.

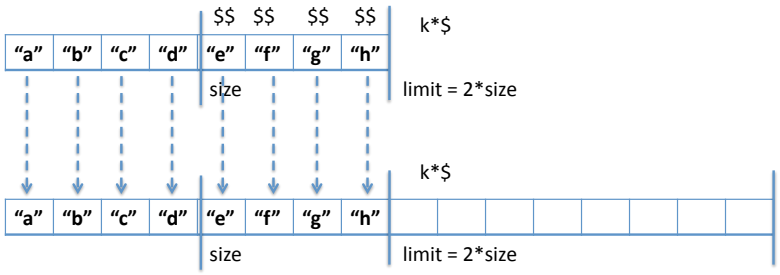
Now assume we have just doubled the size of the array and the invariant holds, that is, we have $k \geq 0$ tokens, and $2 * size = limit$.



After $size$ more inserts we are at $limit$ and added another $2 * size = limit$ tokens.



On the next insert we double the size of the array and copy *limit* array elements, spending *limit* tokens.



Our bank account is reduced back to k tokens, but we know $k \geq 0$, preserving our invariant.

Since we only save a constant number of tokens on each operation, in addition to the constant time the operation itself takes, we never perform more operations than a constant times the number of operations. So our claim above is true: any sequence of n operations performs at most $O(n)$ steps. We also say that the insert operation has *constant amortized time*.

This completes the argument.

In the example, the number of tokens will now never fall below 6. If we add another 8 elements, we will also put $2 * 8 = 16$ tokens into the bank. We will need to spend these to copy the 16 elements already in the array and we are back down to 6.

Tokens are a conceptual tool in our analysis, but they don't need to be implemented. The fact that there are always 0 or more tokens during any sequence of operations is an invariant of the data structure, although not quite in the same way as discussed before because it tracks sequences of operations rather than the internal state of the structure. In fact, it would be possible to add a new field to the representation of the array that would count tokens and raise an exception if it becomes negative. That would alert us to some kind of mistake, either in our amortized analysis or in our program. This would, however, incur a runtime overhead even when assertions are not checked, so tokens are rarely, if ever, explicitly implemented.

This kind of analysis is important to avoid serious programming mistakes. For example, let's say we decide to increase the size of the array only by 1 whenever we run out of space. The token scheme above does not work, because we cannot set aside enough tokens before we need to copy the array again. And, indeed, after we hit *limit* the first time, the next

sequence of n inserts takes $O(n^2)$ operations, because we copy the array on each step until we reach $2 * limit$.

6 Removing Elements

Removing elements from the end of the array is simple, and does not change our amortized analysis, unless we want to shrink the size of the array.

A first idea might be to simply cut the array in half whenever $size$ reaches half the size of the array. However, this cannot work in constant amortized time. The example demonstrating that is an alternating sequence of n inserts and n deletes precisely when we are at the limit of the array. In that case the total cost of the $2 * n$ operations will be $O(n^2)$.

To avoid this problem we cut the size of the array in half only when the number of elements in it reaches $limit/4$. The amortized analysis requires two tokens for any deletion: one to delete the element, and one for any future copy. Then if $size = limit/2$ just after we doubled the size of the array and have no tokens, putting aside one token on every delete means that we have $size/2 = limit/4$ tokens when we arrive at a size of $limit/4$. Again, we have just enough tokens to copy the $limit/4$ elements to the new, smaller array of size $limit/2$.

The code for `uba_rem` ("remove from end"):

```
elem uba_rem(uba L)
//@requires is_uba(L);
//@requires L->size > 0;
//@ensures is_uba(L);
{
  if (L->size <= L->limit/4 && L->limit >= 2)
    uba_resize(L, L->limit/2);
  L->size--;
  elem e = L->data[L->size];
  return e;
}
```

We explicitly check that `L->limit >= 2` to make sure that the limit never becomes 0, which would violate one of our data structure invariants.

One side remark: before we decrement $size$, we should delete the element from the array by writing `L->data[L->size] = ""`. In C0, we do not have any explicit memory management. Storage will be reclaimed and used for future allocation when the garbage collector can see that data are

no longer accessible from the program. If we conceptually remove an element from an unbounded array, but actually keep the element in the (now conceptually empty) array element, the garbage collector can not determine that we will not access it again, because the array still contains a reference to it. In order to allow the garbage collector to free the space occupied by the strings stored in the array, we therefore should overwrite the array element with the empty string `""`, which is the default element for strings. This, however, makes the code specific to strings, which we try to avoid.

Exercises

Exercise 1 *When removing elements from the unbounded array we resize if the limit grossly exceeds its size. Namely when* `L->size <= L->limit/4`*. Your first instinct might have been to already shrink the array when* `L->size <= L->limit/2`*. We have argued by example why that does not give us constant amortized cost* $O(n)$ *for a sequence of* n *operations. We have also sketched an argument why* `L->size <= L->limit/2` *gives the right amortized cost. At which step in that argument would you notice that* `L->size <= L->limit/2` *is the wrong choice?*