Lecture Notes on Data Structures

15-122: Principles of Imperative Computation
Frank Pfenning, André Platzer, Rob Simmons

Lecture 9
September 23, 2014

1 Introduction

In this lecture we introduce the idea of *imperative data structures*. So far, the only interfaces we've used carefully are *pixels* and *string bundles*. Both of these interfaces had the property that, once we created a pixel or a string bundle, we weren't interested in changing its contents. In this lecture, we'll talk about an interface that mimics the arrays that are primitively available in C0.

To implement this interface, we'll need to round out our discussion of types in C0 by discussing *pointers* and *structs*, two great tastes that go great together. We will discuss using contracts to ensure that pointer accesses are safe, as well as the use of *linked lists* to implement the stack and queue interfaces that were introduced last time. The linked list implementation of stacks and queues allows us to handle lists of any length.

Relating this to our learning goals, we have

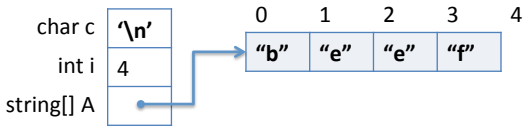
Computational Thinking: We illustrate the power of *abstraction* by considering both the client-side and library-side of the interface to a data structure.

Algorithms and Data Structures: The abstract arrays will be one of our first examples of *abstract datatypes.*

Programming: Introduction of structs and pointers, use and design of interfaces.

2 Structs

So far in this course, we've worked with five different C0 types – `int`, `bool`, `char`, `string`, and arrays `t[]` (there is a array type `t[]` for every type t). The character, string, Boolean, and integer values that we manipulate, store locally, and pass to functions are just the values themselves; the picture we work with looks like this:



When we consider arrays, the things we store in assignable variables or pass to functions are *addresses*, references to the place where the data stored in the array can be accessed. An array allows us to store and access some number of values of the same type (which we reference as `A[0]`, `A[1]`, and so on.

The next data structure we will consider is the *struct*. A *struct* can be used to aggregate together different types of data, which helps us to create data structures. In contrast, an array is an aggregate of elements of the *same* type.

Structs must be explicitly declared in order to define their "shape". For example, if we think of an image, we want to store an array of pixels alongside the width and height of the image, and a struct allows us to do that:

```
typedef int pixel;

struct img_header {
  pixel[] data;
  int width;
  int height;
};
```

Here $data$, $width$, and $height$ are *fields* of the struct. The declaration expresses that every image has an array of $data$ as well as a $width$ and a $height$. This description is incomplete, as there are some missing consistency checks – we would expect the length of $data$ to be equal to the $width$ times the $height$, for instance, but we can capture such properties in a separate data structure invariant.

Structs do not necessarily fit into a machine word because they can have arbitrarily many components, so they must be allocated on the heap (in memory, just like arrays). This is true even if they happen to be small enough to fit into a word (in order to maintain a uniform and simple language implementation).

```
% coin structdemo.c0
C0 interpreter (coin) 0.3.2 'Nickel'
Type '#help' for help or '#quit' to exit.
--> struct img_header IMG;
<stdio>:1.1-1.22:error:type struct img_header not small
[Hint: cannot pass or store structs in variables directly; use
pointers]
```

How, then, do we manipulate structs? We use the same solution as for arrays: we manipulate them via their address in memory. Instead of `alloc_array` we call `alloc` which returns a *pointer* to the struct that has been allocated in memory. Let's look at an example in coin.

```
--> struct img_header* IMG = alloc(struct img_header);
IMG is 0xFFAFFF20 (struct img_header*)
```

We can access the fields of a struct, for reading or writing, through the notation `p->f` where p is a pointer to a struct, and f is the name of a field in that struct. Continuing above, let's see what the default values are in the allocated memory.

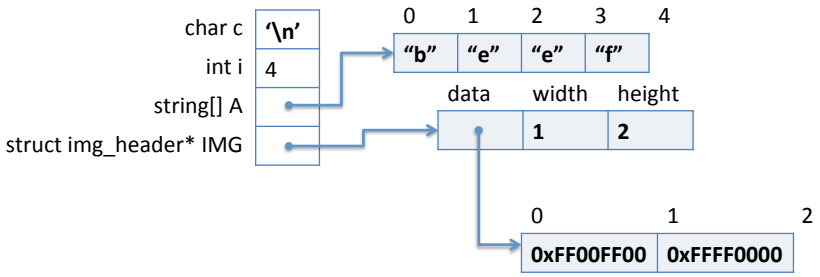
```
--> IMG->data;
(default empty int[] with 0 elements)
--> IMG->width;
0 (int)
--> IMG->height;
0 (int)
```

We can write to the fields of a struct by using the arrow notation on the left-hand side of an assignment.

```
--> IMG->data = alloc_array(pixel, 2);
IMG->data is 0xFFAFC130 (int[] with 2 elements)
--> IMG->width = 1;
IMG->width is 1 (int)
--> (*IMG).height = 2;
(*(IMG)).height is 2 (int)
--> IMG->data[0] = 0xFF00FF00;
IMG->data[0] is -16711936 (int)
--> IMG->data[1] = 0xFFFF0000;
IMG->data[1] is -65536 (int)
```

The notation `(*p).f` is a longer form of `p->f`. First, `*p` follows the pointer to arrive at the struct in memory, then `.f` selects the field `f`. We will rarely use this dot-notation `(*p).f` in this course, preferring the arrow-notation `p->f`.

An updated picture of memory, taking into account the initialization above, looks like this:



3 Pointers

As we have seen in the previous section, a pointer is needed to refer to a struct that has been allocated on the heap. In can also be used more generally to refer to an element of arbitrary type that has been allocated on the heap. For example:

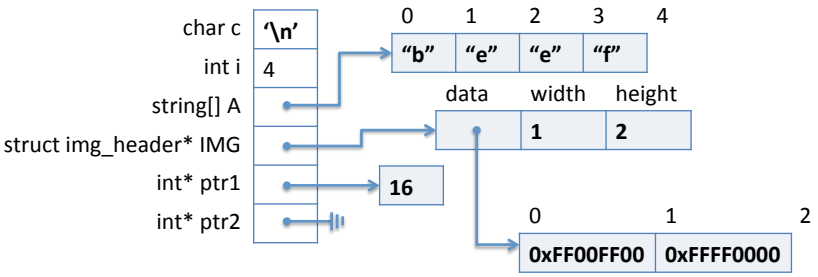
```
--> int* ptr1 = alloc(int);
ptr1 is 0xFFAFC120 (int*)
--> *ptr1 = 16;
*(ptr1) is 16 (int)
--> *ptr1;
16 (int)
```

In this case we refer to the value using the notation `*p`, either to read (when we use it inside an expression) or to write (if we use it on the left-hand side of an assignment).

So we would be tempted to say that a pointer value is simply an address. But this story, which was correct for arrays, is not quite correct for pointers. There is also a special value `NULL`. Its main feature is that `NULL` is not a valid address, so we cannot dereference it to obtain stored data. For example:

```
--> int* ptr2 = NULL;
ptr2 is NULL (int*)
--> *ptr2;
Error: null pointer was accessed
Last position: <stdio>:1.1-1.3
```

Graphically, `NULL` is sometimes represented with the ground symbol, so we can represent our updated setting like this:



To rephrase, we say that a pointer value is an address, of which there are two kinds. A valid address is one that has been allocated explicitly with `alloc`, while `NULL` is an invalid address. In C, there are opportunities to create many other invalid addresses, as we will discuss in another lecture.

Attempting to dereference the null pointer is a safety violation in the same class as trying to access an array with an out-of-bounds index. In C0, you will reliably get an error message, but in C the result is undefined and will not necessarily lead to an error. Therefore:

> *Whenever you dereference a pointer* `p`*, either as* `*p` *or* `p->f`*, you must have a reason to know that* `p` *cannot be* `NULL`*.*

In many cases this may require function preconditions or loop invariants, just as for array accesses.

4 Creating an interface

For roughly the next ten lectures – almost until the second midterm – the lectures for this class will focus on building, analyzing, and using different data structures. When we're thinking about implementing data structures, we will almost always use pointers to structs as the core of our implementation.

We've also seen two kinds of interfaces in our programming assignment: the pixels interface in the early programming assignments, and the string bundle interface in the DosLingos programming assignment. For this lecture, we will work through an intellectual exercise: what if C0 did not provide arrays (we'll limit ourselves to arrays of strings) as a primitive type in C0? If we wanted to use something like strings, we'd have to introduce them from scratch as an abstract type, like pixels or string bundles.

If C0 didn't have built-in, primitive support for arrays, how might we introduce an equivalent type? The primitive operations that C0 provides on string arrays are the ability to create a new array, to get a particular index of an array, and to set a particular index in an array. We could capture these as three functions that act on an abstract type `arr`:

```
typedef _______ arr;
arr    arr_new(int limit);        // alloc_array(string, limit)
string arr_get(arr A, int i);           // A[i]
void   arr_set(arr A, int i, string x); // A[i] = x
```

But this is not a complete picture! An interface needs to also capture the preconditions necessary for using that abstract type *safely*. For instance, we know that safety of array access requires that we only create non-negative-length arrays and we never try to access a negative element of an array:

```
arr    arr_new(int limit)     /*@requires limit >= 0 @*/ ;
string arr_get(arr A, int i)  /*@requires 0 <= i; @*/ ;
```

This still isn't enough: our contracts need to ensure an upper bound so that we don't access the element at index 100 of a length-12 array. We don't have the primitive `\length()` method, as those are a primitive for C0 arrays, not our new `arr` type. So we need an additional function in our interface to get the length, and we'll use that in our contracts:

```
typedef _______ arr;

int arr_len(arr A);

string arr_get(arr A, int i)
  /*@requires 0 <= i && i < arr_len(A); @*/ ;

void arr_set(arr A, int i, string x)
  /*@requires 0 <= i && i < arr_len(A); @*/ ;

arr arr_new(int limit)
  /*@requires limit >= 0 @*/
  /*@ensures arr_len(\result) == limit @*/ ;
```

Now the contracts for our array interface match the C0 array primitives, and the contracts match the safety requirements on arrays. One exception is that, while the C0 primitive `\length(A)` is restricted to contracts, we can use the `arr_len(arr A)` function anywhere in our code.[1]

5 The Library Perspective

When we implement the library for `arr`, we will declare the `arr` type to be a pointer to a `struct arr_header`, which has a `limit` field to hold the length and a `data` field to hold the actual array.

[1] Thanks to Jamie Crawford for finishing this initally unfinished sentence.

```
struct arr_header {
  int limit;
  string[] data;
};
```

Using this knowledge, we can begin to implement the array interface from the library side, though we immediately run into safety issues.

```
int arr_len(arr A) {
  return A->limit;
}
```

How do we know that this dereference of the pointer `A` is safe? We don't! We could add the precondition `A != NULL`. But that won't allow us to safely access `arr_get`:

```
string arr_get(arr A, int i)
//@requires 0 <= i && i < arr_len(A);
{
  return A->data[i];
}
```

We obviously need to require that `A != NULL`, but we also need to know that the array access `A->data[i]` is not accessing the array out-of-bounds! Because the user's precondition was in terms of `arr_len(A)`, it would suffice to know that that function returns the true length. In other words, we need for `arr_len` to `/*@ensure \result == \length(A->data); *@/`. But how do we know *that* about `arr_len`? Again, we don't, unless it's a requirement.

The user, remember, didn't need to know anything about this, because they were ignorant about the internal implementation details of the `arr` type. As long as the user respects the interface, only creating `arr`s with `arr_new` and only manipulating them with `arr_len`, `arr_get`, and `arr_set`, they should be able to expect that the contracts on the interface are sufficient to ensure safety. But we don't have this luxury from the library perspective: all the functions in the library's implementation are going to depend on all the parts of the data structure making sense with respect to all the other parts. We'll capture this notion in a new kind of invariant, a *data structure invariant*.

6 Data Structure Invariants

We can apply operational reasoning as library designers to say that, because the `limit` field of an `arr` is set correctly by `arr_new`, it must remain correct throughout all calls to `arr_get` and `arr_set`. But, as with operational reasoning about loops, this is an error-prone way of thinking about our data structures. Our solution in this case will be to capture what we know about the well-formedness of an array in an invariant; we expect that any `arr` being handled by the user will satisfy this data structure invariant.

The invariants of arrays are pretty simple: a `arr` is well-formed if it is a non-`NULL` pointer to a struct where `\length(AH->data) == AH->limit`. If we try to turn this into a mathematical statement, we get `is_arr`:

```
bool is_arr(struct arr_header* AH) {
  return AH != NULL
    && is_arr_expected_length(AH->data, AH->limit);
}
```

While we would like `is_arr_expected_length` to be a function that returns `true` when the given array has the expected length and `false` otherwise, the restriction of length-checking to contracts makes this impossible to write in C0. In this one case, we'll allow ourselves to write a data structure invariant that might raise an assertion error instead of returning `false`:

```
bool is_arr_expected_length(string[] A, int length) {
  //@assert \length(A) == length;
  return true;
}
```

Whenever possible, however, we prefer data structure invariants that return `true` or `false` to data structures that raise assertion failures.

The data structure invariant, then, implies the postcondition of `arr_len`, and so the function `arr_get` will require the data structure invariant to hold as well, satisfying the precondition of `arr_len`.

```
int arr_len(arr A)
//@requires is_arr(A);
//@ensures \result == \length(A->data);
{
  return A->limit;
}
```

```
string arr_get(arr A, int i)
//@requires is_arr(A);
//@requires 0 <= i && i < arr_len(A);
{
  return A->data[i];
}
```

Functions that create new instances of the data structure should ensure that the data structure invariants hold of their results, and functions that modify data structures should have postconditions to ensure that none of those data structure invariants have been violated.

```
arr arr_new(int limit)
//@requires 0 <= limit;
//@ensures is_arr(\result);
{
  struct arr_header* AH = alloc(struct arr_header);
  AH->limit = limit;
  AH->data = alloc_array(string, limit);
  return AH;
}

void arr_set(arr A, int i, string x)
//@requires is_arr(A);
//@requires 0 <= i && i < arr_len(A);
//@ensures is_arr(A);
//@ensures string_equal(arr_get(A, i), x);
{
  A->data[i] = x;
}
```

Now that we have added data structure invariants, our operational reasoning about what it means for why `arr_get` was safe can be formalized as an invariant. Any client that respects the interface will only ever get and will only ever manipulate arrays that satisfy the data structure invariants, so we know that the data structure invariants we're counting on for safety will hold at runtime.

7 Invariants Aren't Usually Part of the Interface

When we have interfaces that hide implementations from the user, then the data structure invariant should *not* be a part of the interface. Clients don't need to know that the internal invariants are satisfied; as long as they're using `arr` according to the interface, their invariants should be satisfied.