Lecture 8 Notes
Data Structures

15-122: Principles of Imperative Computation (Spring 2016)
Frank Pfenning, André Platzer, Rob Simmons

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.

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.

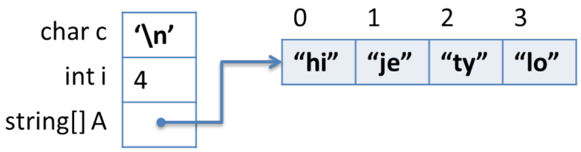
2 Structs

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, Boolean and integer values that we manipulate, store locally, and pass to functions are just the values themselves. For arrays (and strings), 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).

Therefore, when entering the following commands in Coin (the outputs have been elided),

```
--> char c = '\n';
--> int i = 4;
--> string[] A = alloc_array(string, 4);
--> A[0] = "hi";
--> A[1] = "je";
--> A[2] = "ty";
--> A[3] = "lo";
```

the interpreter will store something like the following in its memory:



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 create data structures. By 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.

C0 values such as integers, characters, the address of an array are *small*. Depending on the computer, an address is either 64 bits long or 32 bits long, which means that the *small types* take at most 64 bits to represent. Because structs can have multiple components, they can grow too large for the computer to easily copy around, and C0 does not allow us to use structs as locals:

```
% coin structs.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]
```

Therefore, we can only create structs in allocated memory, just like we can only store the contents of arrays in allocated memory. (This is true even if they happen to be small enough to fit into 32 bytes.) 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.

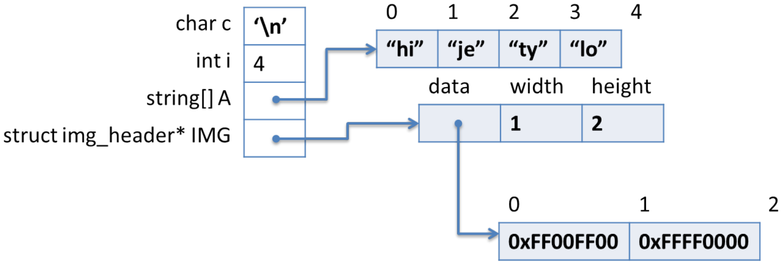
```
--> IMG->data;
(default empty int[] with 0 elements)
--> IMG->width;
0 (int)
--> IMG->height;
0 (int)
```

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:

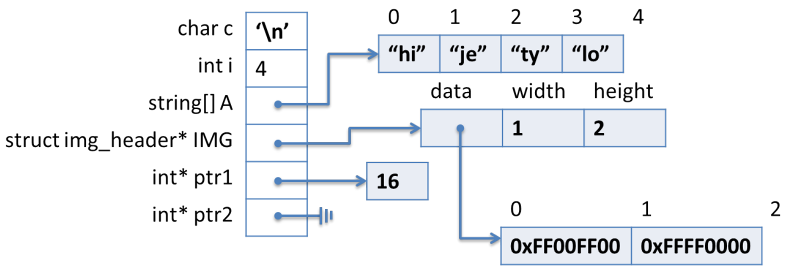
```
--> int* ptr1 = alloc(int);
ptr1 is 0xFFAFC120 (int*)
--> *ptr1 = 16;
*(ptr1) is 16 (int)
--> *ptr1;
16 (int)
```

In this case, we refer to the value of `p` 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

The next ten 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.

For this exercise, we'll build an abstract data type that functions like an array of strings; in fact, we will see our implementation will end up doing a bit more than C0. 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_t`:

```
typedef _______ arr_t;
arr_t arr_new(int size); // alloc_array(string, size)
string arr_get(arr_t A, int i); // A[i]
void arr_set(arr_t 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_t arr_new(int size) /*@requires size >= 0 @*/ ;
string arr_get(arr_t 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 `\length()` method, as it is a primitive for C0 arrays, not our new `arr_t` type. So we need an additional function in our interface to get the length, and we'll use that in our contracts.

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

It's important to emphasize what just happened. Because we want the type `arr_t` to be *abstract*, we can't use `\length` in a contract: we can only use `\length` for arrays. Because we have to be able to write a contract that explains how to use the data type safely, we need to extend our interface with a new function `arr_len`. But because this function is in the interface, the client can access the length of the array — something that can't be done, for C0 arrays, outside of a contract! So we *do* know something about `arr_t` now: it can't just be **`string`**`[]`, because if it was there would be no way to implement `arr_len`.

For this reason, we're going to say that `arr_t` is not an unknown type but that it is an unknown *pointer* type. The commented typedef below shows how we indicate this:

```
// typedef ______* arr_t;

int arr_len(arr_t A)
  /*@requires A != NULL; @*/;

arr_t arr_new(int size)
  /*@requires 0 <= size; @*/
  /*@ensures \result != NULL; @*/
  /*@ensures arr_len(\result) == size; @*/;
```

```
string arr_get(arr_t A, int i)
  /*@requires A != NULL; @*/
  /*@requires 0 <= i && i < arr_len(A); @*/;

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

Admitting that `arr_t` is a pointer also means that we have to add a lot of `NULL` checks to the interface — as the client of the `arr_t` type, we know that `arr_t` is either a valid pointer to the array data structure, or else it is `NULL`.

5 The Library Perspective

When we implement the library for `arr_t`, we will declare a type `arr` 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.

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

Inside the library implementation, we'll use `arr*` instead of `arr_t` to emphasize that we're manipulating a pointer structure. Outside, we use exclusively `arr_t` as our abstract type of supped up arrays. 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)
//@requires A != NULL;
{
  return A->limit;
}

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

In both cases, the `A != NULL` precondition allows us to say that the `A->limit` and `A->data` dereferences are safe. But how do we know `A->data[i]` is not an out-of-bounds array access? We don't — the second precondition of `arr_get` just tells us that `i` is nonnegative and less than whatever `arr_len` returns!

If we want to use the knowledge that `arr_len(A)` returns the length of `A->data`, then we'd need to add `\result == \length(A->data)` as a postcondition of `arr_len`...

...and we can only prove that postcondition true if we add the precondition `A->limit == \length(A->data)` to `arr_len`...

...and if we do that, it changes the safety requirements for the call to `arr_len` in the preconditions of `arr_get`, so we also have to add the precondition `A->limit == \length(A->data)` to `arr_get`.

The user, remember, didn't need to know anything about this, because they were ignorant of the internal implementation details of the `arr_t` type. As long as the user respects the interface, only creating `arr_t`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, as long as 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 size)
//@requires 0 <= size;
//@ensures is_arr(\result);
{
  struct arr_header* AH = alloc(struct arr_header);
  AH->limit = size;
  AH->data = alloc_array(string, size);
  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.