Lecture 8
Data Structures

15-122: Principles of Imperative Computation (Fall 2020)
Frank Pfenning, André Platzer, Rob Simmons, Iliano Cervesato

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 extends 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 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 data structure will be one of our first examples of *abstract datatypes.*

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

1 Structs

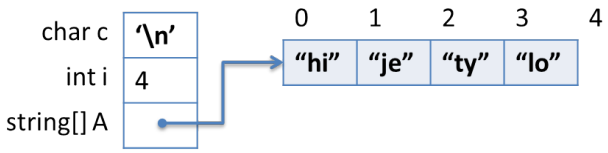
So far in this course, we've worked with five different C0 types — **`int`**, **`bool`**, **`char`**, **`string`**, and arrays *t*`[]` (there is a array type *t*`[]` for every type *t*). The character, Boolean and integer values that we manipulate, store locally, and pass to functions are just the values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

```
struct img_header {
  pixel_t[] 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 `pixel_t`s called `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.

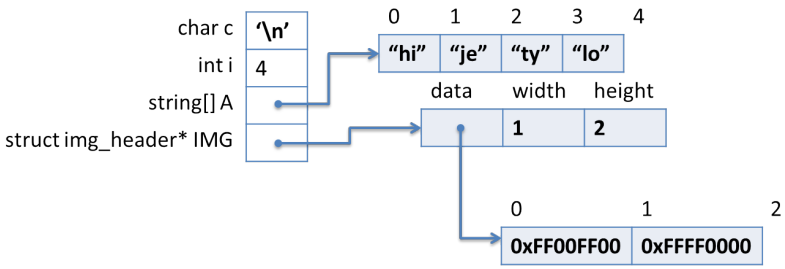
```
--> IMG->data;
(default empty int[] with 0 elements)
--> IMG->width;
0 (int)
--> IMG->height;
0 (int)
```

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_t, 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:



2 Pointers

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

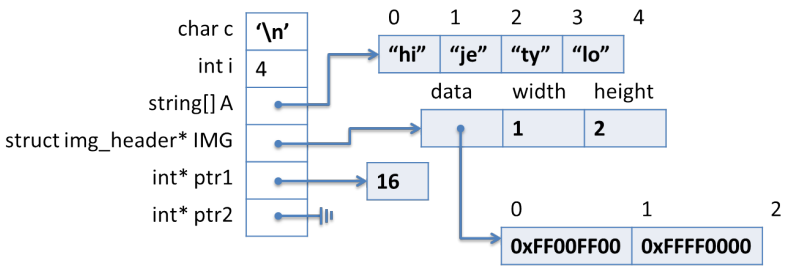
```
--> int* ptr1 = alloc(int);
ptr1 is 0xFFAFC120 (int*)
--> *ptr1 = 16;
*(ptr1) is 16 (int)
--> *ptr1;
16 (int)
```

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.

3 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.

In the rest of this lecture, we will build a data structure for *self-sorting arrays*. Like arrays, they will contain a fixed number of elements of a given type that can be accessed through an index (we will limit ourselves to self-sorting arrays of strings). Unlike C0 arrays, the values in a self-sorting array are sorted, and remain so as we update its values. That allows a programmer using our self-sorting arrays to look for elements using binary search, in $O(\log n)$ time if n is the number of contained elements.

Self-sorting arrays work mostly like arrays of strings. 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 capture these as three functions that act on an abstract type `ssa_t` (mnemonic for self-sorting array type):

```
// typedef _______ ssa_t;
ssa_t  ssa_new(int size);      // ~ alloc_array(string, size)
string ssa_get(ssa_t A, int i);           // ~ A[i]
void   ssa_set(ssa_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:

```
ssa_t  ssa_new(int size)       /*@requires size >= 0 @*/;
string ssa_get(ssa_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 `ssa_t` type. So we need an additional function in our interface to get the length, and we'll use that in our contracts.

```
int ssa_len(ssa_t A);
string ssa_get(ssa_t A, int i)
  /*@requires 0 <= i && i < ssa_len(A); @*/ ;
```

It's important to emphasize what just happened. Because we want the type `ssa_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 `ssa_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 `ssa_t` now: it can't just be **`string`**`[]`, because if it was there would be no way to implement `ssa_len`.

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

```
// typedef ______* ssa_t;

int ssa_len(ssa_t A)
  /*@requires A != NULL; @*/;

ssa_t ssa_new(int size)
  /*@requires 0 <= size; @*/
  /*@ensures \result != NULL; @*/
  /*@ensures ssa_len(\result) == size; @*/;

string ssa_get(ssa_t A, int i)
  /*@requires A != NULL; @*/
  /*@requires 0 <= i && i < ssa_len(A); @*/;

void ssa_set(ssa_t A, int i, string x)
  /*@requires A != NULL; @*/
  /*@requires 0 <= i && i < ssa_len(A); @*/;
```

Admitting that `ssa_t` is a pointer also means that we have to add a lot of `NULL` checks to the interface — as the client of the `ssa_t` type, we know that a value of this type is either a valid pointer to the self-sorting array data structure, or it is `NULL` — and we disallow `NULL` as the representation of *any* valid self-sorting array.

4 The Library Perspective

When we implement the library for `ssa_t`, we will declare a type `ssa` as a synonym for **`struct`** `ssa_header`, which has a `length` field to hold the length and a `data` field to hold the actual array.

```
struct ssa_header {              // Implementation type
  int length;
  string[] data;
};
typedef struct ssa_header ssa;   // Abbreviation

typedef ssa* ssa_t;              // Interface type
```

The last line is where we make a connection between the interface type `ssa_t` exported to the user by the interface, and the type `ssa` (or equivalently `struct ssa_header`) the library uses to implement the exported functionalities. As promised, it is a pointer type. We usually write it as the very last line of the implementation.

Inside the library implementation, we'll use `ssa*` instead of `ssa_t` to emphasize that we're manipulating a pointer structure. Outside, we use exclusively `ssa_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 ssa_len(ssa* A)
//@requires A != NULL;
{
  return A->length;
}

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

In both cases, the precondition `A != NULL` allows us to say that the dereferences `A->length` and `A->data` 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 `ssa_get` just tells us that `i` is nonnegative and less than whatever `ssa_len` returns!

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

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

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

The user, remember, didn't need to know anything about this, because they were ignorant of the internal implementation details of the `ssa_t` type. As long as the user respects the interface, only creating `ssa_t`'s with `ssa_new` and only manipulating them with `ssa_len`, `ssa_get`, and `ssa_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*.

4.1 Data Structure Invariants

We can apply operational reasoning as library designers to say that, as long as the `length` field of an `ssa` is set correctly by `ssa_new`, it must remain correct throughout all calls to `ssa_get` and `ssa_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 `ssa` being handled by the user will satisfy this data structure invariant.

The above invariants for self-sorting arrays are pretty simple: a `ssa` is well-formed if it is a non-`NULL` pointer to a struct where `\length(A->data) == A->length`. They capture safety but do not say anything about the array being sorted. We will add a correctness invariant, that the array be in fact sorted. This is achieved through a function `is_sorted` which is implemented similarly to the corresponding function for arrays of integers — it can be found in the code accompanying this lecture. If we try to turn this into a mathematical statement that captures the overall well-formedness requirements for our implementation, we get the *specification function* `is_ssa`:

```
bool is_ssa(ssa* A) {
  return A != NULL
      && is_array_expected_length(A->data, A->length)
      && is_sorted(A);
}
```

While we would like `is_array_expected_length` to be a function that returns `true` when the given array has the expected length and `false` oth-

erwise, 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_array_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 structure invariants that raise assertion failures.

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

```
int ssa_len(ssa* A)
//@requires is_ssa(A);
//@ensures \result == \length(A->data);
{
  return A->length;
}

string ssa_get(ssa* A, int i)
//@requires is_ssa(A);
//@requires 0 <= i && i < ssa_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 result, and functions that modify data structures should have postconditions to ensure that none of those data structure invariants have been violated.

```
ssa* ssa_new(int size)
//@requires 0 <= size;
//@ensures is_ssa(\result);
{
  ssa* A = alloc(ssa);
  A->length = size;
  A->data = alloc_array(string, size);
  return A;
```

```
}

void ssa_set(ssa* A, int i, string x)
//@requires is_ssa(A);
//@requires 0 <= i && i < ssa_len(A);
//@ensures is_ssa(A);
{
  A->data[i] = x;

  // Move x up the array if needed
  for (int j=i; j < A->length-1 &&
                string_compare(A->data[j],A->data[j+1]) > 0;
       j++)
    //@loop_invariant i <= j && j <= A->length - 1;
    swap(A->data, j, j+1);

// Move x down the array if needed
  for (int j=i; j > 0 &&
                string_compare(A->data[j],A->data[j-1]) < 0;
       j--)
    //@loop_invariant 0 <= j && j <= i;
    swap(A->data, j, j-1);
}
```

The code below `A->data[i] = x` moves `x` up or down the array `A->data` to ensure that the sortedness invariant remains satisfied once the function returns. As a consequence, it is unlikely that a subsequent call to `ssa_get(A, i)` will return `x` as its value. This differs from C0 arrays, for which elements can always be found where we just put them.

Now that we have added data structure invariants, our operational reasoning for why `ssa_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.

Invariants Aren't Usually Part of the Interface

When we have interfaces that hide implementations from the user, then the data structure invariant, captured here by the function `is_ssa`, 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 `ssa` according to the interface,

their invariants should be satisfied.

This applies to all aspects of `is_ssa`, in particular to the correctness invariant checked by the function `is_sorted`. But how shall the client be sure that the elements of a self-sorting array are and remains sorted as it is used? The client may trust the developer of the data structure library. Another option is to write a client-side version of `is_sorted` using *only the operations exported by the interface*.

4.2 Debugging a Library Implementation

The implementation of the functions `ssa_len` and `ssa_get`, and even `ssa_new`, is pretty straightforward. But `ssa_set` is more involved since it moves elements around to keep the array sorted. Having a bug in our first version is not unlikely. How to notice our code has a bug? And once we know there is a bug, how to find where it is so that we can fix it?

There are two complementary approaches to noticing that our code contains a bug. The first is to have a good reason to believe that potentially unsafe operations are actually safe (array accesses out of bounds in our example). As we did earlier, using contracts to *reason* about our code is a great way to do so: in this way we can identify and fix errors even before we try to run our code the first time. However, some bugs are quite subtle and we may miss them when reasoning about our program. This is when the second approach comes into play: *testing*. Once we have a program we are relatively confident about, we want to write a battery of *test cases* to identify any lingering bugs. Except sometimes for very large test cases, we always want to run out tests in debug mode (by compiling them with the `-d` flag).

Once testing reveals a bug, we need to find out exactly where it is so that we can fix it. Sometimes, this is quite easy: a simple inspection of the function that fails the test will do it. Often, however, bugs are not as obvious. There are two may ways to identify a bug. The first one relies on contracts, in particular data structure invariants. When a test fails because of a contract violation, the execution will abort by reporting the exact line where the contract failed. From there, it is usually easy to find what went wrong and to fix it.

The second approach to figuring out what is causing a bug is to use *print statements*. This is useful when the bug does not cause a contract to fail (for example if our code for `ssa_set` had written the new string `x` in every position in the array `A->data`) or because our contracts are not strong enough (e.g., if we wrote incorrect data structure invariants). One particularly useful use of print statements is to print the contents of the data structure the

library is implementing. To this end, we write a *print function* `print_ssa` that we can use in our implementation to inspect the contents of the data structure anytime we feel something may be off. Here it is:

```
void print_ssa(ssa* A)
//@requires is_ssa(A);
{
  int len = A->length;
  print("SSA length: "); printint(len);
  print("; SSA data: [");
  for (int i = 0; i < len; i++)
  //@loop_invariant 0 <= i && i <= len;
  {
    print(A->data[i]);
    if (i < len-1) print(", ");
  }
  print("]");
}
```

Like the specification function `is_ssa`, the print function lives in the library implementation and is not exported to the client. (Some libraries may want to export a print function, but it is best to give it a different name.) Like `is_ssa`, it is good practice to write `print_ssa` before implementing any library function. Notice that `print_ssa` uses the fields of the underlying implementation type rather than the functions exported by the interface (e.g., `A->length` as opposed to `ssa_len(A)`). This is because we want to use it to catch bugs in these very functions! (We sometimes relax this rule for library functions that are particularly simple.) Here, we gave the print function the precondition `is_ssa(A)`. This is what we want to do when we are confident the specification function is correct. If we had a bug in `is_ssa`, it may be worth commenting out this precondition temporarily so that we can use our print function inside the code of `is_ssa` to find the bug (but then we want to put the precondition back).

5 The Client Perspective

The client of our self-sorting array library can use any function exported by the library (plus the type `ssa_t`) *and nothing else*. As an example, we write a client-side printing function.

```
void display_this_ssa(ssa_t A)
```

```
//@requires A != NULL;
{
  println("");
  for (int j = 0; j < ssa_len(A); j++)
  //@loop_invariant 0 <= j;
  {
    printint(j); print(" => "); println(ssa_get(A, j));
  }
}
```

Notice that this code uses the functions exported by the interface (here `ssa_len` and `ssa_get`). Using the fields `length` and `data` of its implementation would violate the interface, and most likely result in bugs when compiling this code with another implementation of this library. Similarly, defining `display_this_ssa` as

```
void display_this_ssa(ssa_t A)
//@requires A != NULL;
{
  print_ssa(A);
}
```

would be a violation of the interface because `print_ssa` is not exported by the library.