Abstract Data Types

An abstract data type (or ADT), roughly speaking, is a class of objects with operations defined on them. The operations come with a signature (which is a description of the type of arguments and result values that they return) and a description of how those operations behave.

The stack ADT, for instance, has the following signature:

```
empty : () -> stack
push : (int stack) -> stack

isEmpty : (stack) -> bool
top : (stack) -> int
pop : (stack) -> stack
```

Intuitively, `(empty)` creates an empty stack, `(push a s)` creates a new stack with `a` on top of stack `s`, `(isEmpty s)` checks if `s` is an empty stack, `(top s)` returns the element at the top of stack `s`, and `(pop s)` returns the stack obtained by removing the top element of `s`.

You can split functions in the signature of an ADT into two classes: the *constructors* (or *creators*), in charge of constructing new values of the ADT, and *selectors* (or *predicates*, or *operations*), in charge of extracting information or observations from a value of the ADT. For stacks, `empty` and `push` are constructors, and `isEmpty`, `top`, and `pop` are selectors.

A stack is a LIFO (Last-In, First-Out) structure, meaning that the last element you put in is the first element you take out.

An ADT does not specify an implementation. Indeed, for most ADTs, there are multiple implementations available, each with different characteristics in terms of how efficient they are, how much memory space they use, etc.

The question becomes: how can we make precise what the ADT operations are doing, without discussing a specific implementations? There are many approaches people have developed to write specifications for ADTs, we will focus on a particular one called *algebraic specifications*.

The idea is simple: we write an equation describing how each selector interact with each constructor (when the interaction makes sense). This requires some thinking, usually.

For stacks, here is an algebraic specification:

```
(isEmpty (empty)) = true
(isEmpty (push a s)) = false
(top (push a s)) = a
(pop (push a s)) = s
```

There are no equations describing `(top (empty))`—intuitively, it makes no sense to look at the value at the top of an empty stack. That there is no specification means that an implementation can do whatever it wants in that case. So in Java, we could have an exception being raised, in Scheme, a runtime error, in ACL2, a default value being returned.

I claim that the above completely describes how stacks behave, at least with respect to the given operations.

As an implementor, if you wanted to provide an implementation for stacks, you would have to make sure it satisfies that specification. Let's play that game in ACL2. Let's write an implementation for stacks in ACL2. First, we have to choose a representation for stacks. We'll use a (true) list of elements.

```
(defun stackp (x)
  (if (endp x)
      (= x  NIL)
    (and (integerp (car x)) (stackp (cdr x)))))

(defun empty () nil)

(defun push (a s) (cons a s))

(defun isEmpty (s) (endp s))

(defun top (s)
  (if (endp s)
      NIL
    (car s)))

(defun pop (s)
  (if (endp s)
      NIL
    (cdr s)))
```

We have a logic in hand, so we can formally prove that the above implementation satisfies the specification. First, though, we need to write down the specification in ACL2. It's a

minor point, but it still needs to be done:

$$\texttt{(isEmpty (empty))}$$
$$\texttt{(integerp a)} \wedge \texttt{(stackp s)} \Longrightarrow \neg\texttt{(isEmpty (push a s))}$$
$$\texttt{(integerp a)} \wedge \texttt{(stackp s)} \Longrightarrow \texttt{(= (top (push a s)) a)}$$
$$\texttt{(integerp a)} \wedge \texttt{(stackp s)} \Longrightarrow \texttt{(= (pop (push a s)) s)}$$

(For uniformity, we add conditions on the inputs to the arguments of the operations. We can often prove more general versions of these theorems, depending on the implementation, but as far as ADT algebraic specifications are concerned, we care only about the equations holding when operations are given values of the right types.)

It is an easy exercise to prove that these formulas are provable given the above implementation of stacks.

Note that using the algebraic specification of an ADT, we can predict the result of any sequence of operations. For instance, suppose we push 4 and 3 on top of an empty stack, we pop that stack, then we push 2 and 1, and then we pop again, and then look at the top of the stack:

```
(top (pop (push 1 (push 2 (pop (push 3 (push 4 (empty))))))))
```

If we simplify the above using the algebraic specification, we can see what the result of the operations is:

```
  (top (pop (push 1 (push 2 (pop (push 3 (push 4 (empty))))))))
= (top (push 2 (pop (push 3 (push 4 (empty))))))
= 2
```

Boom. We can predict the result of any sequence of operations, *even though we still do not have an implementation for stacks.* In that sense, the specification captures how stack behave. Any implementation of stacks that implements the specification will return value 2 when asked to perform the above sequence of operations. Of course, it may do it faster or slower, depending on the implementation.

Let's look at another ADT, one that is slightly more interesting. Let's look at queues. A queue is a structure in which elements are inserted at one end and removed from the other end. Think of queues at the ATM. Queues are FIFO (First-In, First-Out) structures. Here is a signature for queues, again, for simplicity, taken to hold integers:

```
empty : () -> queue
enqueue: (int queue) -> queue
isEmpty : (queue) -> bool
front : (queue) -> int
dequeue: (queue) -> queue
```

The algebraic specifications for the queue operations are not so straightforward. Part of the problem is, for instance, that to look at the front of a queue on which we just enqueued a value 10, we do not just return 10, but rather, we need to burrow down into the queue and find the first thing we enqueued.

```
(isEmpty (empty)) = true
(isEmpty (enqueue a q)) = false
(front (enqueue a q)) =
        a           if isEmpty(q)=true
        (front q)   otherwise
(dequeue (enqueue a q)) =
        (empty)                     if isEmpty(q)=true
        (enqueue a (dequeue q))     otherwise
```

Note that some of the equations are conditional equations. Again, this specification can be used to predict the effect of operations. For instance: if we enqueue 1, 2 and 3 onto an empty queue, then dequeue, and then look at the value at the front of the queue, we should get the value 2. And indeed:

```
  (front (dequeue (enqueue 3 (enqueue 2 (enqueue 1 (empty))))))
= (front (enqueue 3 (dequeue (enqueue 2 (enqueue 1 (empty))))))
= (front (enqueue 3 (enqueue 2 (dequeue (enqueue 1 (empty))))))
= (front (enqueue 3 (enqueue 2 (empty))))
= (front (enqueue 2 (empty)))
= 2
```

A standard implementation of queues is to use a list. Intuitively, we dequeue from the head of the list, and we enqueue by adding elements at the end of the list. Here is an implementation, complete with a predicate for queues:

```
(defun queuep (q) (int-true-listp q))

(defun empty () NIL)

(defun enqueue (a q) (append q (list a)))

(defun isEmpty (q) (endp q))

(defun front (q)
  (if (endp q)
      NIL
    (car q)))
```

```
(defun dequeue (q)
  (if (endp q)
      NIL
    (cdr q)))
```

Can we prove that this implementation satisfies the algebraic specification? Let's first convert the algebraic specification to ACL2. Let's introduce a little subtlety—not a big deal now, but it will become a big deal next lecture.[1] We have an implementation for the queue ADT. Depending on your representation, equality for queues need not be exactly `=`. So for equations that regulate equality of queues (or in general, of whatever ADT you have at hand), you need to use an implementation of equality that your ADT implementation provides. We did not bother doing that for stacks (though we should have to be uniform), because in our implementation, two stacks were equal when they were equal with respect to the usual `=`. Here too, but not next time, so let's define

```
(defun queue-= (q1 q2) (= q1 q2))
```

and use `queue-=` for queue equalities in the specification:

`(isEmpty (empty))`

`(integerp a)` $\wedge$ `(queuep q)` $\Longrightarrow$ $\neg$`(isEmpty (enqueue a q))`

`(integerp a)` $\wedge$ `(queuep q)` $\wedge$ `(isEmpty q)`
$\Longrightarrow$ `(= (front (enqueue a q)) a)`

`(integerp a)` $\wedge$ `(queuep q)` $\wedge$ $\neg$`(isEmpty q)`
$\Longrightarrow$ `(= (front (enqueue a q)) (front q))`

`(integerp a)` $\wedge$ `(queuep q)` $\wedge$ `(isEmpty q)`
$\Longrightarrow$ `(queue-= (dequeue (enqueue a q)) (empty))`

`(integerp a)` $\wedge$ `(queuep q)` $\wedge$ $\neg$`(isEmpty q)`
$\Longrightarrow$ `(queue-= (dequeue (enqueue a q)) (enqueue a (dequeue q)))`

These are all easy to prove, either by hand, or using the theorem prover.

[1] I messed this part up in lecture; this supercedes what I said then.