Abstract Data Types

Lecture notes accompanying COL106 (Data Structures), Semester I, 2018-19, IIT Delhi

Amitabha Bagchi
Department of CS&E, IIT Delhi

August 13, 2018

1 What is an abstract data type

Abstract Data Types (ADTs) are used in programming, which is a practical field, and so definitions of ADTs vary from author to author. For the purposes of these notes we will start with the following general definition.

Definition 1 (Abstract Data Type). *An* abstract data type *is a set $\mathcal{X}$ that we will refer to in these notes as the* ground set[1]*, along with a (finite) set of* operations*, $\mathcal{O} = \{f_i : 1 \leq i \leq k\}$, where the operation f_i is a function from a domain D_i to range R_i.*

We will look at examples in more detail ahead but for now, to make this definition concrete here is a natural example: $\mathcal{X}$ could be the integers with $\mathcal{O} = \{\texttt{isEqual}, \texttt{sum}\}$, where `isEqual` is a function that takes two integers and returns 1 if they are equal and 0 otherwise, and `sum` is a function that takes two integers as arguments and returns their sum.

Remarks. We will keep the following things in mind while using Definition 1:

1. Sometimes the set $\mathcal{X}$ itself is referred to as an ADT, e.g. "the integer abstract data type." In such cases the set of operations is assumed to be understood. In general such a practice should be avoided and when referring to an ADT we must always specify *both* the ground set $\mathcal{X}$ and the set of operations $\mathcal{O}$.

2. There may be several possible operations on a given ground set, but typically we define the set $\mathcal{O}$ with two considerations in mind

   (a) It is always a good practice to only enumerate fundamental operations and not derived operations, e.g., integer division can be implemented using subtraction and comparison operations so it is an example of a derived operation. However in some cases when the programming infrastructure provides special support for derived operations, it may be better to specify derived operations as well.

   (b) Not all applications require all operations so the operation set should be specified with the application in mind, e.g., when we consider the marks a student gets in a course we may use floating point numbers since sometimes the scores may go into decimal places, but we will never need to take the square root of a student's marks so we don't need to specify a square root operation.

[1] Beware: The term *ground set* is not widely used for this purpose so do not quote it without specifying its meaning.

3. Although we have left D_i and R_i unspecified, it is understood that the domain D_i of every f_i will contain some reference to $\mathcal{X}$ although the range R_i may not, e.g., the domains of both `isEqual` and `sum` mentioned above are $\mathcal{X} \times \mathcal{X}$, where $\mathcal{X} = \mathbb{Z}$ (the integers). However the range of `isEqual` is the set $\{0, 1\}$ whereas the range of `sum` is again $\mathcal{X}$.

4. Sometimes for the purposes of programming we may need to define functions whose domain does not contain any reference to $\mathcal{X}$. For example, let us define an ADT `IntTuple` whose the ground set is $\mathbb{Z} \times \mathbb{Z}$, i.e., the set of 2-tuples of integers. We may need a function `createIntTuple` which takes two integers and returns an `IntTuple`. When we write class definitions in Java we often need such a function to create an object. *For the study of ADTs we will* not *consider such functions as* operations *of the ADT.* We will assume that objects from the ADT's ground set are already available to us.

A note on creating objects of an ADT. As discussed above we need a function to create a data element of an ADT. But create out of what? Every computing system has some fundamental system-defined types (typically integer, floating point, char etc) and these are the basic building blocks of more complex ADTs. But these too can be described as ADTs. What are they created from? The fact is that ADTs cannot be fully presented in an abstract way. At some point a concrete underlying system has to come into the picture. Typically this comes into the picture when we try to create a data object of an ADT. In the examples below we will implicitly assume this underlying system when required. In some examples a previously defined ADT will be used as the building block for another ADT.

We now look at some examples of ADTs in greater detail. Note again: different authors may present these ADTs in different ways, so if you wish to quote the material presented below please specify your definitions before doing so.

2 Integer ADTs and the Boolean ADT

We now look at three related ADTs. Two of these are well know, the Integer ADT and the Boolean ADT but we will also discuss a third one: Integers modulo n which lies somewhere between these two well known ADTs conceptually.

2.1 The Integer ADT

Definition 2 (`Int` **ADT).** $\mathcal{X} = \mathbb{Z}$ *and* $\mathcal{O} = \{$`isEqual`, `lessThanEqualTo`, `sum`, `negation`$\}$ *where for all* $x, y \in \mathcal{X}$

- `Int-isEqual`$(x, y) = 1$ *if* $x = y$ *and* 0 *otherwise,*
- `Int-lessThanEqualTo`$(x, y) = 1$ *if* $x \leq y$ *and* 0 *otherwise,*
- `Int-sum`$(x, y) = x + y$, *and*
- `Int-negation`$(x) = -x$.

Along with this we will also define a creation function `Int-create` that takes a string of digits and creates an `Integer` out of it, i.e., if we write

$$a = \texttt{Int-create}('2765')$$

then the variable a will have contain the integer 2765 and all the operations defined above will be able to take it as an argument.

The set of operations, $\mathcal{O}$, is actually sufficient for most useful operations on integers. For example `Int-minus`(x, y), defined as $x - y$ can easily be expressed as `Int-sum`$(x,$ `Int-negation`(y). In order to understand this better, attempt the following exercises.

Exercise 1. *Using only the operations given in Definition 2 and* `Int-minus` *defined above, implement the following functions. Also, for each function below count the number of operations from* $\mathcal{O} \cup \{$`Int-minus`$\}$ *that you used. Note that this count will be in terms of the parameters* x *and* y *given to the functions. Under the assumption that all operations take the same time, we will call this count of the number of operation the "running time" of the function.*

1. `Int-mult`$(x, y) = x \cdot y$.
2. `Int-div`(x, y) *returns the quotient obtained when we divide* x *by* y.
3. `Int-mod`(x, y) *returns the remainder obtained when we divide* x *by* y.

Exercise 2. *You will have realised after solving Exercise 1 that instead of* `Int-sum` *if we had defined an operation* `Int-increment` *that returns* $x + 1$ *when called with parameter* x*, we could still have defined the sum function, the subtraction function and all three functions given in Exercise 1. What would the running time of those functions be in terms of number of operations used in this case?*

2.2 Integers modulo p

This definition is widely seen in mathematics and in programming:

Definition 3 (`Integers mod` p **ADT).** *Given a positive integer* p*, the ADT* `Intmod`(p) *is defined as follows:* $\mathcal{X} = \{0, 1, \ldots, p-1\}$*, and* $\mathcal{O} = \{$`Intmod-isEqual`$,$ `Intmod-sum`$\}$ *where for all* $x, y \in \mathcal{X}$

- `Intmod-isEqual`$(p)(x, y) = 1$ *if* $x = y$ *and* 0 *otherwise,*
- `Intmod-sum`$(p)(x, y) = x + y \mod p$.

There are many possible ways of implementing this ADT, but we will implement it using the `Integer` ADT defined in Section 2.1. What does this mean? It means that we will use the operations and creation function of `Integer` to realise the operations and creation function of `Intmod`(p). Let's start with the creation function:

`Intmod-create`$(p)(s)$ **where** s **is a string and** p **is already available to us as an** `Integer`**:**

- Let $a =$ `Int-create`(s).
- If `Int-lessThanEqualTo`$(a,$ `Int-create`$('-1'))$ or `Int-lessThanEqualTo`(p, a) then return "Error: Out of range",
- else return a.

It looks like the creation function is returning an `Integer`, not an `Intmod`(p). But there's nothing to worry about since our remaining operations for *this implementation* of the `Intmod`(p) ADT know this fact and will incorporate it in their definitions. To see what that means let us define `Intmod-sum`.

`Intmod-sum`$(p)(x, y)$ **where** x **and** y **belong to the ground set of** `Intmod`(p)**:**

- return `Int-mod`(`Int-sum`$(x, y), p$).

Here `Int-mod` is the derived `Integer` operation you defined in Exercise 1. Don't confuse it with `Intmod`! Note here that the definition of `Intmod-sum` "knows" that although x, y are from the `Integer mod` p ADT, *they have been implemented using the* `Integer` *ADT.*

Exercise 3. *Define* `Intmod-isEqual`*.*

2.3 The Boolean ADT

There are many ways of defining the Boolean abstract data type. Here is one of them:

Definition 4. *The ADT* `Bool` *has ground set* $\mathcal{X} = \{'T', 'F'\}$ *where* T *and* F *are two characters.*[2] $\mathcal{O} = \{$`Bool-not`, `Bool-or`, `Bool-and`$\}$ *where*

- `Bool-not`(x) *is* F *if* $x = T$ *and* T *if* $x = F$*.*
- `Bool-or`(x, y) *is* T *if either* x *or* y *is* T *and* F *otherwise.*
- `Bool-and`(x, y) *is* T *if both* x *and* y *are* T *and* F *otherwise.*

Exercise 4. *Using the operations of* $\mathcal{O}$*, define the derived operation* `Bool-isEqual`(x, y) *which returns 1 if* x *and* y *are identical and 0 otherwise.*

Let us try and implement `Bool` using the ADT `Intmod`(2). Let's start with the create function:

`Bool-create`(s) **where** s **is either** T **or** F**.**

- if $s = T$ return `Intmod-create`(2)('1') else return `Intmod-create`(2)('0')

The rest of the implementation is part of the next exercise.

Exercise 5. *Assume that* `Bool` *is implemented using* `Intmod`(2) *and the creation function is as described above.*

1. *Argue that the following implementation of* `Bool-not` *is correct:*

$$\texttt{Bool-not}(x) = \texttt{Intmod-sum}(2)(x, \texttt{Intmod-create}(2)('1')).$$

2. *Define the remaining operations of* `Bool` *using the operations of* `Intmod`(2)*.*

3 Basic ADTs for collections: Lists, Arrays and Sets

Now we discuss some more complex ADTs: those associated with collections of objects. Unlike Section 2 we will skip the details of how these are implemented, i.e., we will not provide details of the creation function. We will assume that if we need to implement them we will use whatever programming constructs are available in the system we are using (e.g. arrays, linked lists etc). The full implications of this will become clear as we proceed.

[2] In the following we will just write T or F with the understanding that we mean $'T'$ or $'F'$ respectively.

3.1 The List ADT

Before we proceed we introduce some notation: Given a set S, we denote by $\sigma(S)$ the set of all sequences of any length k, $k \geq 0$ whose individual elements are taken from S, e.g., if S is the set of integers then any sequence of integers of any length k, $k \geq 0$. Note that strings can be viewed as $\sigma(C)$ where C is the set of characters.

Definition 5. *Given an ADT* `A` *with ground set* $\mathcal{X}_A$*, the* `AList` *ADT is defined as having ground set* $\sigma(\mathcal{X}_A)$*, i.e., the set of all sequences whose elements are taken from* $\mathcal{X}_A$*. The set of operations is* $\mathcal{O} =$ {`AList-isEmpty`, `AList-insertFront`, `AList-deleteFront`, `AList-readFront`, `AList-isMember`} *where for all* $x \in \mathcal{X}_A$ *and all* $l \in \sigma(\mathcal{X}_A)$*,*

- `AList-isEmpty`(l) *returns 1 if* l *has no elements and 0 otherwise,*
- `AList-insertFront`(l, x)*: if* $l = x_1 x_2 \ldots x_n$ *then this returns* $x x_1 x_2 \ldots x_n$*.*
- `AList-deleteFront`(l)*: if* l *is empty then this returns "Error: List empty." Otherwise if* $l = x_1 l'$ *for some* $x_1 \in \mathcal{X}_A, l' \in \sigma(\mathcal{X}_A)$ *then this returns* l'*.*
- `AList-readFront`(l)*: if* l *is empty then this returns "Error: List empty." Otherwise if* $l = x_1 l'$ *for some* $x_1 \in \mathcal{X}_A, l' \in \sigma(\mathcal{X}_A)$ *then this returns* x_1 *without changing* l*.*
- `AList-isMember`(l, x) *returns 1 if* x *is present in* l *and 0 otherwise.*

It appears that we have defined a very limited set of operations. For example, consider the operation `AList-deleteRear` that deletes the last element in a list l. How would we derive this operation using only the four operations of $\mathcal{O}$? One way is this: take a helper list l_1. Delete an element from the front of l and then check if l has now become empty. If not then insert the deleted element into l_1 and continue. If yes then discard the deleted element and now reverse the process by deleting from the front of l_1 and inserting into the front of l till l_1 becomes empty.

Exercise 6. *Prove that the helper list method for implementing* `AList-deleteRear` *returns the correct answer. Is it possible to implement this operation without using a second list?*

Exercise 7. *We define two operations (i)* `AList-insertAfterFirst`(l, x, y) *inserts the element* y *after the* first *occurrence of the element* x *and (ii)* `AList-insertAfterEvery`(l, x, y) *inserts a copy of the element* y *after* every *occurrence of the element* x*. Use the helper list method to implement both of these operations. You may assume that you have the method* `A-isEqual` *that compares two elements of the ADT* `A`*.*

While attempting Exercises 6 and 7 keep in mind that we don't know anything about how the `AList` is implemented, so be careful you don't let any assumption about the implementation creep into your solution of those exercises or the ones below.

Exercise 8. *Let us consider the* `IntList` *ADT, i.e. the* `AList` *ADT where the type* `A` *is* `Int` *as given in Definition 2 with all the operations of that definition available to us. Implement the following operations using only the operations of* `Int` *and the* `IntList` *operations given in Definition 5 assuming that* `A` *is* `Int`*. Keep in mind that the input list should remain as before unless otherwise specified. You may use helper lists where required.*

1. `IntList-sumList`(l) *returns the sum of all elements in* l*.*
2. `IntList-lessThanList`(l, x) *returns a list containing all those elements of* l *that are less than or equal to the* `Int` x*.*
3. `IntList-altSumList`(l)*: if the list is* $x_1 x_2 \ldots x_n$ *then this function returns* $x_1 - x_2 + x_3 + \cdots (-1)^{n+1} x_n$*. Use the* `Bool` *ADT and its operations to implement this operation.*

3.2 The Array ADT

Again some notation: given a set S we define $f(S)$ as the set of all functions defined from a finite set of non-negative integers to S, i.e., $f(S)$ is a set of tuples $\{(x_i, k_i) : 1 \leq i \leq n\}$ where n is a finite positive integer, $x_i \in S$ for all $1 \leq i \leq n$ and $k_i \neq k_j$ whenever $i \neq j$ (i.e. all the k_is are distinct).

Definition 6. *Given an ADT* `A` *with ground set* $\mathcal{X}_A$*, the* `AArray` *ADT is defined as having ground set* $f(\mathcal{X}_A)$*, where* $f(S)$ *is the set of all functions defined from a finite set of non-negative integers to* S*. The set of operations is* $\mathcal{O} = \{$`AArray-isEmpty`, `AArray-readIndex`, `AArray-insert`, `AArray-delete`$\}$ *where for all* $x \in \mathcal{X}_A$*, all* $i \in \mathbb{Z}_+ \cup \{0\}$*, and all* $A \in f(\mathcal{X}_A)$*,*

- `AArray-isEmpty`(A) *returns 1 if* A *has no elements and 0 otherwise,*
- `AArray-readIndex`(A, i)*: if* (x, i) *exists in* A *this returns* x *otherwise returns "Error: Out of range".*
- `AArray-insert`(A, x, i)*: it inserts* (x, i) *in* A*. If some* (y, i) *already existed in* A *it overwrites it without returning any error.*
- `AArray-delete`(A, i)*: if* (x, i) *exists in* A *it deletes it and returns "Success" otherwise returns "Error: Out of range".*

Let's explore this ADT a little.

Exercise 9. *Write a function to find the sum of all the elements in an* `IntArray`*. What is the time taken by this function? How would you modify the definition of the ADT to minimise the time taken to sum the elements?*

Exercise 10. *Here is one possible way of making summing the elements of* `IntArray` *faster and may also be useful for other purposes. Along with each* `AArray` *you maintain an* `IntList` *that stores the indexes of the elements in ascending order. So if your array is* $\{(a, 7), (b, 3), (d, 11), (z, 26)\}$ *you will store an* `IntList` $3 \cdot 7 \cdot 11 \cdot 26$*. Let's call this ADT that contains an array with this helper* `IntList` *as* `AFastArray`*.*

1. *Implement all the operations of* `AArray` *for* `AFastArray`*. You will now have to ensure that your helper* `IntList` *is always correctly maintained.*
2. *What is the time taken to sum the elements of* `IntFastArray`*? Is it better than the time taken in Exercise 9?*

Exercise 11. *Define a new ADT* `AContiguousArray` *which is more like the normal array we have used in programming, i.e., where if you have* n *elements then they are associated with indices 0 to* $n - 1$*. Define four operations similar to the ones in Definition 6 and implement this new ADT using the operations of* `AArray`*.*

Clearly the `AArray` ADT is quite different from the `AList` ADT. Which one is more powerful? What does more powerful mean? One definition of more powerful is this: if ADT `A` can be "easily" implemented using ADT `B` then ADT `B` can be thought of as more powerful. Let's try out this notion.

Exercise 12. *Assuming that you have the* `AArray` *and* `Int` *ADTs and all their operations available, implement the* `AList` *ADT's operations.*

Exercise 13. *Reverse what you did in Exercise 12, i.e., implement* `AArray` *using* `AList` *and the other ADTs we have defined.*

Which one of these exercises was easier? One way to decide this is to account for how many operations were needed. Was extra storage required or not? If yes then how much?

3.3 The Set ADT

Finally we define an ADT for sets of elements. It may seem to be a simpler concept than the lists and arrays we have already dealt with but it is actually not as simple as it appears.

Definition 7. *Given an ADT* `A` *with ground set* $\mathcal{X}_A$*, the* `ASet` *ADT is defined as having ground set* $g(\mathcal{X}_A)$*, where* $g(\mathcal{X}_A)$ *is the set of all* finite *subsets of* $\mathcal{X}_A$*. The set of operations is* $\mathcal{O} = \{$`ASet-isEmpty`, `ASet-isMember` *where, for all* $x \in \mathcal{X}_A$ *and* $S \in g(\mathcal{X}_A)$

- `ASet-isEmpty`(S) *returns 1 if* S *is empty and 0 otherwise,*
- `ASet-isMember`(S, x) *returns 1 if* $x \in S$ *and 0 otherwise,*
- `ASet-insert`(S, x) *returns the set* $S \cup \{x\}$*,*
- `ASet-delete`(S, x) *returns* $S \setminus \{x\}$ *if* $x \in S$ *and "Error: item is not in set" otherwise.*

Note that the key difference between the `Set` ADT and the other ADTs we have seen so far in Section 3 is that elements *cannot* be repeated in this ADT by the way sets are defined in mathematics. All the other ADTs we have defined for collections before this allow for repetitions of elements. What is the implication of this difference. Let's discuss it after we solve the following exercises.

Exercise 14. *Implement the* `ASet` *ADT using the* `AList` *ADT and using all the different array ADTs we defined in Section 3.2.*

If you attempted Exercise 14 before reading forward you would have realised that it is *not* possible to solve it in general. It is only possible to solve it if the ADT `A` has some comparison function `A-isEqual` that can tell you if two elements are the same or not. It seems like a mild condition but there are many cases where it is not: floating point numbers are not a bad example where an equality function is not obvious to define.

What happens in the other direction. Is it possible to implement lists and the arrays using sets? Clearly the restriction on `A` that Exercise 14 requires should not be needed here. But does that make our life significantly easier. There's only one way to find out:

Exercise 15. *Implement the* `AList` *ADT and all the different array ADTs we defined in Section 3.2 using the* `ASet` *ADT. Use extra space or whatever other helper ADTs you require.*