Abstract data type

From Wikipedia, the free encyclopedia

In computer science, an **abstract data type** (**ADT**) is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics. An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.[1]

For example, an abstract stack could be defined by three operations: `push`, that inserts some data item onto the structure, `pop`, that extracts an item from it (with the constraint that each pop always returns the most recently pushed item that has not been popped yet), and `peek`, that allows data on top of the structure to be examined without removal. When analyzing the efficiency of algorithms that use stacks, one may also specify that all operations take the same time no matter how many items have been pushed into the stack, and that the stack uses a constant amount of storage for each element.

Abstract data types are purely theoretical entities, used (among other things) to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the type systems of programming languages. However, an ADT may be implemented by specific data types or data structures, in many ways and in many programming languages; or described in a formal specification language. ADTs are often implemented as modules: the module's interface declares procedures that correspond to the ADT operations, sometimes with comments that describe the constraints. This information hiding strategy allows the implementation of the module to be changed without disturbing the client programs.

The term **abstract data type** can also be regarded as a generalised approach of a number of algebraic structures, such as lattices, groups, and rings.[2] This can be treated as part of the subject area of artificial intelligence. The notion of abstract data types is related to the concept of data abstraction, important in object-oriented programming and design by contract methodologies for software development[*citation needed*].

Contents

- 1 Defining an abstract data type (*ADT*)
  - 1.1 Imperative view
    - 1.1.1 Abstract variable
    - 1.1.2 Instance creation
    - 1.1.3 Preconditions, postconditions, and invariants
    - 1.1.4 Example: abstract stack (imperative)
    - 1.1.5 Single-instance style
  - 1.2 Functional ADT definitions
    - 1.2.1 Example: abstract stack (functional)
- 2 Advantages of abstract data typing
- 3 Typical operations
- 4 Examples
- 5 Implementation
  - 5.1 Example: implementation of the stack ADT

- 5.1.1 Functional-style interface
- 5.2 ADT libraries
- 5.3 Built-in abstract data types
- 6 See also
- 7 References
- 8 Further
- 9 External links

Defining an abstract data type (*ADT*)

An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles.

Imperative view

In the "imperative" view, which is closer to the philosophy of imperative programming languages, an abstract data structure is conceived as an entity that is *mutable* — meaning that it may be in different *states* at different times. Some operations may change the state of the ADT; therefore, the order in which operations are evaluated is important, and the same operation on the same entities may have different effects if executed at different times — just like the instructions of a computer, or the commands and procedures of an imperative language. To underscore this view, it is customary to say that the operations are *executed* or *applied*, rather than *evaluated*. The imperative style is often used when describing abstract algorithms. This is described by Donald E. Knuth and can be referenced from here The Art of Computer Programming.

Abstract variable

Imperative ADT definitions often depend on the concept of an *abstract variable*, which may be regarded as the simplest non-trivial ADT. An abstract variable V is a mutable entity that admits two operations:

- `store`(V,x) where x is a *value* of unspecified nature; and
- `fetch`(V), that yields a value;

with the constraint that

- `fetch`(V) always returns the value x used in the most recent `store`(V,x) operation on the same variable V.

As in so many programming languages, the operation `store`(V,x) is often written $V \leftarrow x$ (or some similar notation), and `fetch`(V) is implied whenever a variable V is used in a context where a value is required. Thus, for example, $V \leftarrow V + 1$ is commonly understood to be a shorthand for `store`(V,`fetch`(V) + 1).

In this definition, it is implicitly assumed that storing a value into a variable U has no effect on the state of a distinct variable V. To make this assumption explicit, one could add the constraint that

- if U and V are distinct variables, the sequence { `store`(U,x); `store`(V,y) } is equivalent to { `store`(V,y);

`store`(*U*,*x*) }.

More generally, ADT definitions often assume that any operation that changes the state of one ADT instance has no effect on the state of any other instance (including other instances of the same ADT) — unless the ADT axioms imply that the two instances are connected (aliased) in that sense. For example, when extending the definition of abstract variable to include abstract records, the operation that selects a field from a record variable *R* must yield a variable *V* that is aliased to that part of *R*.

The definition of an abstract variable *V* may also restrict the stored values *x* to members of a specific set *X*, called the *range* or *type* of *V*. As in programming languages, such restrictions may simplify the description and analysis of algorithms, and improve their readability.

Note that this definition does not imply anything about the result of evaluating `fetch`(*V*) when *V* is *un-initialized*, that is, before performing any `store` operation on *V*. An algorithm that does so is usually considered invalid, because its effect is not defined. (However, there are some important algorithms whose efficiency strongly depends on the assumption that such a `fetch` is legal, and returns some arbitrary value in the variable's range.[*citation needed*])

Instance creation

Some algorithms need to create new instances of some ADT (such as new variables, or new stacks). To describe such algorithms, one usually includes in the ADT definition a `create`() operation that yields an instance of the ADT, usually with axioms equivalent to

- the result of `create`() is distinct from any instance *S* in use by the algorithm.

This axiom may be strengthened to exclude also partial aliasing with other instances. On the other hand, this axiom still allows implementations of `create`() to yield a previously created instance that has become inaccessible to the program.

Preconditions, postconditions, and invariants

In imperative-style definitions, the axioms are often expressed by *preconditions*, that specify when an operation may be executed; *postconditions*, that relate the states of the ADT before and after the execution of each operation; and *invariants*, that specify properties of the ADT that are *not* changed by the operations.

Example: abstract stack (imperative)

As another example, an imperative definition of an abstract stack could specify that the state of a stack *S* can be modified only by the operations

- `push`(*S*,*x*), where *x* is some *value* of unspecified nature; and
- `pop`(*S*), that yields a value as a result;

with the constraint that

- For any value *x* and any abstract variable *V*, the sequence of operations { `push`(*S*,*x*); *V* ← `pop`(*S*) } is

equivalent to $\{ V \leftarrow x \}$;

Since the assignment $\{ V \leftarrow x \}$, by definition, cannot change the state of S, this condition implies that $\{ V \leftarrow \texttt{pop}(S) \}$ restores S to the state it had before the $\{ \texttt{push}(S,x) \}$. From this condition and from the properties of abstract variables, it follows, for example, that the sequence

$\{ \texttt{push}(S,x); \texttt{push}(S,y); U \leftarrow \texttt{pop}(S); \texttt{push}(S,z); V \leftarrow \texttt{pop}(S); W \leftarrow \texttt{pop}(S); \}$

where x,y, and z are any values, and U, V, W are pairwise distinct variables, is equivalent to

$\{ U \leftarrow y; V \leftarrow z; W \leftarrow x \}$

Here it is implicitly assumed that operations on a stack instance do not modify the state of any other ADT instance, including other stacks; that is,

- For any values x,y, and any distinct stacks S and T, the sequence $\{ \texttt{push}(S,x); \texttt{push}(T,y) \}$ is equivalent to $\{ \texttt{push}(T,y); \texttt{push}(S,x) \}$.

A stack ADT definition usually includes also a Boolean-valued function $\texttt{empty}(S)$ and a $\texttt{create}()$ operation that returns a stack instance, with axioms equivalent to

- $\texttt{create}() \neq S$ for any stack S (a newly created stack is distinct from all previous stacks)
- $\texttt{empty}(\texttt{create}())$ (a newly created stack is empty)
- $\texttt{not empty}(\texttt{push}(S,x))$ (pushing something into a stack makes it non-empty)

Single-instance style

Sometimes an ADT is defined as if only one instance of it existed during the execution of the algorithm, and all operations were applied to that instance, which is not explicitly notated. For example, the abstract stack above could have been defined with operations $\texttt{push}(x)$ and $\texttt{pop}()$, that operate on "the" only existing stack. ADT definitions in this style can be easily rewritten to admit multiple coexisting instances of the ADT, by adding an explicit instance parameter (like S in the previous example) to every operation that uses or modifies the implicit instance.

On the other hand, some ADTs cannot be meaningfully defined without assuming multiple instances. This is the case when a single operation takes two distinct instances of the ADT as parameters. For an example, consider augmenting the definition of the stack ADT with an operation $\texttt{compare}(S,T)$ that checks whether the stacks S and T contain the same items in the same order.

Functional ADT definitions

Another way to define an ADT, closer to the spirit of functional programming, is to consider each state of the structure as a separate entity. In this view, any operation that modifies the ADT is modeled as a mathematical function that takes the old state as an argument, and returns the new state as part of the result. Unlike the "imperative" operations, these functions have no side effects. Therefore, the order in which they are evaluated is immaterial, and the same operation applied to the same arguments (including the same input states) will always return the same results (and output states).

In the functional view, in particular, there is no way (or need) to define an "abstract variable" with the semantics of imperative variables (namely, with `fetch` and `store` operations). Instead of storing values into variables, one passes them as arguments to functions.

Example: abstract stack (functional)

For example, a complete functional-style definition of a stack ADT could use the three operations:

- `push`: takes a stack state and an arbitrary value, returns a stack state;
- `top`: takes a stack state, returns a value;
- `pop`: takes a stack state, returns a stack state;

In a functional-style definition there is no need for a `create` operation. Indeed, there is no notion of "stack instance". The stack states can be thought of as being potential states of a single stack structure, and two stack states that contain the same values in the same order are considered to be identical states. This view actually mirrors the behavior of some concrete implementations, such as linked lists with hash cons.

Instead of `create`(), a functional definition of a stack ADT may assume the existence of a special stack state, the *empty stack*, designated by a special symbol like Λ or "()"; or define a `bottom`() operation that takes no arguments and returns this special stack state. Note that the axioms imply that

- `push`$(\Lambda,x) \neq \Lambda$

In a functional definition of a stack one does not need an `empty` predicate: instead, one can test whether a stack is empty by testing whether it is equal to Λ.

Note that these axioms do not define the effect of `top`(s) or `pop`(s), unless s is a stack state returned by a `push`. Since `push` leaves the stack non-empty, those two operations are undefined (hence invalid) when $s = \Lambda$. On the other hand, the axioms (and the lack of side effects) imply that `push`$(s,x) =$ `push`(t,y) if and only if $x = y$ and $s = t$.

As in some other branches of mathematics, it is customary to assume also that the stack states are only those whose existence can be proved from the axioms in a finite number of steps. In the stack ADT example above, this rule means that every stack is a *finite* sequence of values, that becomes the empty stack (Λ) after a finite number of `pop`s. By themselves, the axioms above do not exclude the existence of infinite stacks (that can be `pop`ed forever, each time yielding a different state) or circular stacks (that return to the same state after a finite number of `pop`s). In particular, they do not exclude states s such that `pop`$(s) = s$ or `push`$(s,x) = s$ for some x. However, since one cannot obtain such stack states with the given operations, they are assumed "not to exist".

Advantages of abstract data typing

- Encapsulation

Abstraction provides a promise that any implementation of the ADT has certain properties and abilities; knowing these is all that is required to make use of an ADT object. The user does not need any technical knowledge of how the implementation works to use the ADT. In this way, the implementation may be complex but will be encapsulated in a simple interface when it is actually used.

- Localization of change

Code that uses an ADT object will not need to be edited if the implementation of the ADT is changed. Since any changes to the implementation must still comply with the interface, and since code using an ADT may only refer to properties and abilities specified in the interface, changes may be made to the implementation without requiring any changes in code where the ADT is used.

- Flexibility

Different implementations of an ADT, having all the same properties and abilities, are equivalent and may be used somewhat interchangeably in code that uses the ADT. This gives a great deal of flexibility when using ADT objects in different situations. For example, different implementations of an ADT may be more efficient in different situations; it is possible to use each in the situation where they are preferable, thus increasing overall efficiency.

Typical operations

Some operations that are often specified for ADTs (possibly under other names) are

- `compare`(*s*,*t*), that tests whether two structures are equivalent in some sense;
- `hash`(*s*), that computes some standard hash function from the instance's state;
- `print`(*s*) or `show`(*s*), that produces a human-readable representation of the structure's state.

In imperative-style ADT definitions, one often finds also

- `create`(), that yields a new instance of the ADT;
- `initialize`(*s*), that prepares a newly created instance *s* for further operations, or resets it to some "initial state";
- `copy`(*s*,*t*), that puts instance *s* in a state equivalent to that of *t*;
- `clone`(*t*), that performs *s* ← `new`(), `copy`(*s*,*t*), and returns *s*;
- `free`(*s*) or `destroy`(*s*), that reclaims the memory and other resources used by *s*;

The `free` operation is not normally relevant or meaningful, since ADTs are theoretical entities that do not "use memory". However, it may be necessary when one needs to analyze the storage used by an algorithm that uses the ADT. In that case one needs additional axioms that specify how much memory each ADT instance uses, as a function of its state, and how much of it is returned to the pool by `free`.

Examples

Some common ADTs, which have proved useful in a great variety of applications, are

- Container
- Deque
- List
- Map
- Multimap
- Multiset
- Priority queue

- Queue
- Set
- Stack
- String
- Tree
- Graph

Each of these ADTs may be defined in many ways and variants, not necessarily equivalent. For example, a stack ADT may or may not have a `count` operation that tells how many items have been pushed and not yet popped. This choice makes a difference not only for its clients but also for the implementation.

Implementation

Implementing an ADT means providing one procedure or function for each abstract operation. The ADT instances are represented by some concrete data structure that is manipulated by those procedures, according to the ADT's specifications.

Usually there are many ways to implement the same ADT, using several different concrete data structures. Thus, for example, an abstract stack can be implemented by a linked list or by an array.

An ADT implementation is often packaged as one or more modules, whose interface contains only the signature (number and types of the parameters and results) of the operations. The implementation of the module — namely, the bodies of the procedures and the concrete data structure used — can then be hidden from most clients of the module. This makes it possible to change the implementation without affecting the clients.

When implementing an ADT, each instance (in imperative-style definitions) or each state (in functional-style definitions) is usually represented by a handle of some sort.[3]

Modern object-oriented languages, such as C++ and Java, support a form of abstract data types. When a class is used as a type, it is an abstract type that refers to a hidden representation. In this model an ADT is typically implemented as a class, and each instance of the ADT is an object of that class. The module's interface typically declares the constructors as ordinary procedures, and most of the other ADT operations as methods of that class. However, such an approach does not easily encapsulate multiple representational variants found in an ADT. It also can undermine the extensibility of object-oriented programs. In a pure object-oriented program that uses interfaces as types, types refer to behaviors not representations.

Example: implementation of the stack ADT

As an example, here is an implementation of the stack ADT above in the C programming language.

Imperative-style interface

An imperative-style interface might be:

```
typedef struct stack_Rep stack_Rep;        /* Type: instance representation (an opaque record
typedef stack_Rep *stack_T;                /* Type: handle to a stack instance (an opaque poi
typedef void *stack_Item;                  /* Type: value that can be stored in stack (arbitr

stack_T stack_create(void);                /* Create new stack instance, initially empty. */
```

```
void stack_push(stack_T s, stack_Item e);  /* Add an item at the top of the stack. */
stack_Item stack_pop(stack_T s);           /* Remove the top item from the stack and return i
int stack_empty(stack_T ts);               /* Check whether stack is empty. */
```

This implementation could be used in the following manner:

```
#include <stack.h>            /* Include the stack interface. */
stack_T t = stack_create();   /* Create a stack instance. */
int foo = 17;                 /* An arbitrary datum. */
stack_push(t, &foo);          /* Push the address of 'foo' onto the stack. */
…
void *e = stack_pop(t);       /* Get the top item and delete it from the stack. */
if (stack_empty(t)) { … }     /* Do something if stack is empty. */
…
```

This interface can be implemented in many ways. The implementation may be arbitrarily inefficient, since the formal definition of the ADT, above, does not specify how much space the stack may use, nor how long each operation should take. It also does not specify whether the stack state t continues to exist after a call $s \leftarrow \mathrm{pop}(t)$.

In practice the formal definition should specify that the space is proportional to the number of items pushed and not yet popped; and that every one of the operations above must finish in a constant amount of time, independently of that number. To comply with these additional specifications, the implementation could use a linked list, or an array (with dynamic resizing) together with two integers (an item count and the array size)

Functional-style interface

Functional-style ADT definitions are more appropriate for functional programming languages, and vice-versa. However, one can provide a functional style interface even in an imperative language like C. For example:

```
typedef struct stack_Rep stack_Rep;          /* Type: stack state representation (an opaque r
typedef stack_Rep *stack_T;                  /* Type: handle to a stack state (an opaque poin
typedef void *stack_Item;                    /* Type: item (arbitrary address). */

stack_T stack_empty(void);                   /* Returns the empty stack state. */
stack_T stack_push(stack_T s, stack_Item x); /* Adds x at the top of s, returns the resulting
stack_Item stack_top(stack_T s);             /* Returns the item currently at the top of s. *
stack_T stack_pop(stack_T s);                /* Remove the top item from s, returns the resul
```

The main problem is that C lacks garbage collection, and this makes this style of programming impractical; moreover, memory allocation routines in C are slower than allocation in a typical garbage collector, thus the performance impact of so many allocations is even greater.

ADT libraries

Many modern programming languages, such as C++ and Java, come with standard libraries that implement several common ADTs, such as those listed above.

Built-in abstract data types

The specification of some programming languages is intentionally vague about the representation of certain built-in data types, defining only the operations that can be done on them. Therefore, those types can be viewed as "built-in ADTs". Examples are the arrays in many scripting languages, such as Awk, Lua, and Perl, which can be regarded as an implementation of the Map or Table ADT.

See also

- Initial algebra
- Concept (generic programming)
- Design by contract
- Formal methods
- Functional specification
- Liskov substitution principle
- Object-oriented programming
- Type system
- Type theory
- Algebraic data type
- Generalized algebraic data type

References

1. ^ Barbara Liskov, Programming with Abstract Data Types, in Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, pp. 50--59, 1974, Santa Monica, California
2. ^ Rudolf Lidl (2004). *Abstract Algebra*. Springer. ISBN 81-8128-149-7., Chapter 7,section 40.
3. ^ Robert Sedgewick (1998). *Algorithms in C*. Addison/Wesley. ISBN 0-201-31452-5., definition 4.4.

Further

- Mitchell, John C.; Plotkin, Gordon (July 1988). "Abstract Types Have Existential Type" (http://theory.stanford.edu/~jcm/papers/mitch-plotkin-88.pdf). *ACM Transactions on Programming Languages and Systems* **10** (3).

External links

- Abstract data type (http://www.nist.gov/dads/HTML/abstractDataType.html) in NIST Dictionary of Algorithms and Data Structures