Data abstraction

Contents

General considerations

Data abstraction is based on two ideas:

- definition of a type as a set of objects and a set of allowed operations on these objects,
- information hiding.

Both ideas have always been present in data types (even though operation sets were seldom explicitly defined): a representation of a number, character, string in a program differs from its representation in memory.

User-defined data types are an extension of data abstraction in built-in types, but they do not give us a systematic way of defining operations.

User-defined abstract data types (**ADT**s) should allow the programmer to have

- mechanisms (with a well-defined syntax and semantics) for describing ADTs,
- mechanism for describing export/import of constants, variables, types, subprograms.

A data abstraction unit (module, cluster, package, class) is a complete syntactic unit with:

- a specification of the type and its operations,
- a representation of this type's objects by means of other (simpler) objects,
- an implementation of the operations.

Specification:
how to use the type; the interface between operations (subprograms) and their users. The assumption is that all access to objects and manipulations on objects of this ADT must be done by invoking operations.

Representation:
how objects are constructed from built-in types and other ADTs.

Implementation:
bodies of subprograms.

All this is usually written in two parts: a program unit with a specification, another unit with the rest.

Export: which operations or special objects defined for the ADT can be used in the program—and which are only used internally. Only the name and signature of an operation is exported, but not the details of its implementation. Similarly, representation details are not visible outside the definition of the ADT.

Information on export appears in the syntactic unit that defines the ADT.

Import: which exported objects will be visible and in what form (for example, are qualified names necessary?).

This information appears in the syntactic unit that wants to use the ADT.

Export may be defined implicitly (e.g., everything in the specification part is exported) or explicitly.

Import may be defined for individual elements of an ADT, or for the whole type.

A formal notation for ADT

An ADT definition in a hypothetical notation:

```
adt stack(item);
operations
 newstack()          → stack;
 push(stack, item)   → stack;
 pop(stack)          → stack;
 top(stack)          → item;
 is_empty(stack)     → Boolean;
var s: stack; i: item;
conditions
 pop(newstack) = newstack;
 pop(push(s, i)) = s;
 top(push(s, i)) = i;
 is_empty(newstack) = true;
 is_empty(push(s, i)) = false;
errors
 top(newstack);
end stack;
```

The conditions define stacks in relation to other stacks, stack elements and stack values.

A condition can be treated as a rewriting rule that makes it possible to reason about the ADT without having to consider its representation and implementation. A few selected operations are left undefined—they are type constructors.

Examples of formal stack expressions and other expressions that involve stacks:

```
newstack

push(newstack, 17)

push(push(newstack, 17), 6)

top(push(push(newstack, 17), 6))  =
    6

pop(push(push(newstack, 17), 6))  =
    push(newstack, 17)

is_empty(push(push(newstack, 17), 6)) =
    false
```

Another ADT definition:

```
adt queue(item);
operations
 newqueue()          → queue;
 addq(queue, item) → queue;
 delq(queue)         → queue;
 frontq(queue)       → item;
 is_empty_q(queue) → Boolean;
var q: queue; i: item;
conditions
 delq(newqueue) = newqueue;
 delq(addq(q, i)) =
   if is_empty_q(q) then newqueue
   else addq(delq(q), i);
 frontq(addq(q, i)) =
   if is_empty_q(q) then i
   else frontq(q);
 is_empty_q(newqueue) = true;
 is_empty_q(addq(q, i)) = false;
errors
 frontq(newqueue);
end queue;
```


Examples of queue-related formal expressions (the constructors are `newqueue` and `addq`):

```
newqueue

addq(newqueue,17)

addq(addq(newqueue,17), 6)

frontq(addq(addq(newqueue,17), 6))  =
   frontq(addq(newqueue,17))  =
   17

delq(addq(
   addq(newqueue,17), 6))  =
addq(delq(
   addq(newqueue,17)), 6)  =
addq(newqueue, 6)

is_empty_q(addq(
   addq(newqueue,17), 6))  =
   false
```


Binary search trees (the constructors—the primitive operations—are `newtree` and `make`).

```
adt bst(item);
operations
 newtree()              → bst;
 make(bst, item, bst) → bst;
 left(bst)              → bst;
 data(bst)              → item;
 right(bst)             → bst;
 insert(item, bst)     → bst;
 isnewtree(bst)        → Boolean;
 is_in(item, bst)      → Boolean;
var L: bst; R: bst;
    i: item; j: item;
conditions
 left(make(L, i, R))  = L;
 data(make(L, i, R))  = i;
 right(make(L, i, R)) = R;
```



```
 insert(j, newtree) =
  make(newtree, j, newtree);
 insert(j, make(L, i, R)) =
  if i = j then
     make(L, i, R)
  else if i < j then
     make(L, i, insert(j, R))
  else /* i > j */
     make(insert(j, L), i, R);
 isnewtree(newtree) = true;
 isnewtree(make(L, i, R)) = false;
 is_in(j, newtree) = false;
 is_in(j, make(L, i, R)) =
  if i = j then      true
  else if i < j then is_in(j, R)
  else /* i > j */   is_in(j, L);
errors
 left(newtree);
 right(newtree);
 data(newtree);
end bst;
```

```
/* initialize - create an empty tree */
```

newtree

```
/* inserting 5 */
```

insert(5, newtree) =
make(newtree, **5**, newtree)

```
/* inserting 3 into the left subtree */
```

insert(3, make(newtree, **5**, newtree)) =
make(insert(3, newtree), **5**, newtree) =
make(make(newtree, **3**, newtree), **5**, newtree)

```
/* inserting 8 into the right subtree */
```

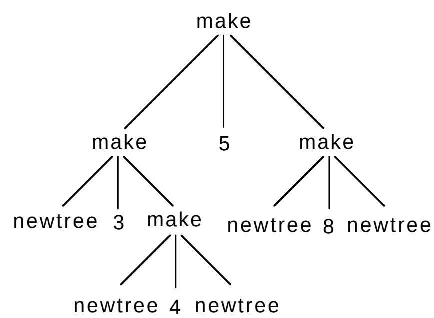
insert(8, make(make(newtree, **3**, newtree), **5**,
newtree)) =
make(make(newtree, **3**, newtree), **5**,
insert(8, newtree)) =
make(make(newtree, **3**, newtree), **5**,
make(newtree, **8**, newtree))

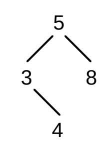

```
/* inserting 4 into the left subtree */
```

insert(4, make(make(newtree, **3**, newtree), **5**,
make(newtree, **8**, newtree))) =

```
/* ... and into its right subtree */
```

make(insert(4, make(newtree, **3**, newtree)), **5**,
make(newtree, **8**, newtree)) =
make(make(newtree, **3**, insert(4, newtree)), **5**,
make(newtree, **8**, newtree)) =
make(make(newtree, **3**,
make(newtree, **4**, newtree)), **5**,
make(newtree, **8**, newtree))





is_in(4,
make(make(newtree, 3, make(newtree, 4, newtree)),
5, make(newtree, 8, newtree))) =
is_in(4,
make(newtree, **3**, make(newtree, 4, newtree))) =
is_in(4,
make(newtree, **4**, newtree)) = true

is_in(6,
make(make(newtree, 3, make(newtree, 4, newtree)),
5, make(newtree, 8, newtree))) =
is_in(6,
make(newtree, **8**, newtree)) =
is_in(6,
newtree) = false

These were specific examples. It is more interesting to formulate (and prove!) general statements, such as:

is_in(E, insert(E, T)) = true

Example of a formal proof

Show that
⊗ is_in(E, insert(E, T)) = true

Proof by induction on the size of the tree.

Case 1: T = newtree

is_in(E, insert(E, newtree))
= is_in(E, make(newtree, E, newtree))
= true
by the 1^{st} axiom for is_in

Case 2: T = make(L, D, R)

and we assume that ⊗ holds for L and R

is_in(E, insert(E, make(L, D, R)))
= **if** D = E **then**
is_in(E, make(L, D, R))
else
if D < E **then**
is_in(E, make(L, D, insert(E, R)))
else /* D > E */
is_in(E, make(insert(E, L), D, R))

Case 2.1: D = E

is_in(E, insert(E, make(L, D, R)))
= is_in(E, make(L, D, R)) = true

Case 2.2: D < E

is_in(E, insert(E, make(L, D, R)))
= is_in(E, make(L, D, insert(E, R)))
= is_in(E, insert(E, R)) = true
by the 2^{nd} axiom and the inductive assumption

Case 2.3: D > E

is_in(E, insert(E, make(L, D, R)))
= is_in(E, make(insert(E, L), D, R))
= is_in(E, insert(E, L)) = true
by the 2^{nd} axiom and the inductive assumption

All in all,

is_in(E, insert(E, make(L, D, R)))
= true

and

is_in(E, insert(E, newtree)) = true

This means, for all T,

is_in(E, insert(E, T)) = true

Data abstraction in Modula-2

A data abstraction unit is called a module, and it is written in two parts.

```
DEFINITION MODULE integer_q_module;
 TYPE queue;
 PROCEDURE newqueue: queue;
 PROCEDURE addq(Q: queue; I: INTEGER):
           queue;
 PROCEDURE delq(Q: queue): queue;
 PROCEDURE frontq(Q: queue): INTEGER;
 PROCEDURE is_empty_q(Q: queue): BOOLEAN;
END integer_q_module;
```



```
IMPLEMENTATION MODULE integer_q_module;
 TYPE q_ptr = POINTER TO q_node;
      q_node = RECORD
                elem: INTEGER;
                next: q_ptr
               END;
      queue = RECORD fr, tl: q_ptr END;

 PROCEDURE newqueue: queue;
  VAR QQ: queue; P: q_ptr;
  BEGIN
   NEW(P);      P^.next := NIL;
   QQ.fr := P;  QQ.tl := P;
   RETURN QQ;
  END;

 PROCEDURE addq(Q: queue; I: INTEGER):
               queue;
  BEGIN
   Q.tl^.elem := I;     NEW(Q.tl^.next);
   Q.tl := Q.tl^.next;  Q.tl^.next := NIL
   RETURN Q;
  END;
```

```
PROCEDURE delq(Q: queue): queue;
  BEGIN
   IF Q.fr <> Q.tl (* not empty *) THEN
   BEGIN
    Q.fr := Q.fr^.next;  RETURN Q;
   END
   ELSE (* signal an error/exception *)
  END;

 PROCEDURE frontq(Q: queue): INTEGER;
  BEGIN
   IF Q.fr <> Q.tl (* not empty *) THEN
   RETURN Q.fr^.elem;
   ELSE (* signal an error/exception *)
  END;

 PROCEDURE is_empty_q(Q: queue): BOOLEAN;
  BEGIN
    RETURN Q.fr = Q.tl;
  END;
END integer_q_module;
```


This may be used as follows:

```
MODULE main;

  FROM integer_q_module
  IMPORT
    addq, delq, newqueue,
    frontq, is_empty_q, queue;

  FROM InOut
  IMPORT
    Read, ReadLn, EOL, ReadInt,
    WriteLn, WriteInt (*etc.*);

  VAR MY_Q: queue;

(*  proceed to use MY_Q, e.g.:
 MY_Q := newqueue;
 MY_Q := addq(MY_Q, 6);
    and so on
*)
```


Data abstraction in Ada

A data abstraction unit is a package. Again, it is defined in two parts. First, a specification.

```
package bst_pkg is
 type bst is limited private;
 function newtree return bst;
 function make(L: bst; I: integer; R: bst)
          return bst;
 function left(T: bst) return bst;
 function data(T: bst) return integer;
 function right(T: bst) return bst;
 function insert(I: integer; T: bst)
          return bst;
 function isnewtree(T: bst) return Boolean;
 function is_in(I: integer; T: bst)
          return Boolean;
 private
  type node is
   record
    left: bst; info: integer; right: bst
   end record;
  type bst is access node;
end bst_pkg;
```



```
package body bst_pkg is
 function newtree return bst is
  begin
   return null;
  end newtree;
 function make(L:bst; I:integer; R:bst)
          return bst is
  begin
   return new bst (L, I, R);
  end make;
  -- etc. etc.
end bst_pkg;
```

This may be used as follows:

```
with bst_pkg;  -- compile with bst_pkg
               -- as the "context"
use  bst_pkg;  -- import all operations
               -- from this package
procedure main is
 MY_T: bst;  -- Full type name: bst_pkg.bst
 -- MY_T := newtree;
 -- MY_T := insert( 17, MY_T);
 -- and so on
```

A generalization: generic packages.

```
generic
 type item is private;
 with function "<"(L,R: item) return Boolean;
package bst_pkg is
 type bst is limited private;
 function newtree return bst;
 function make(L: bst; I: item; R: bst)
          return bst;
 function left(T: bst) return bst;
 function data(T: bst) return item;
 function right(T: bst) return bst;
 function insert(I: item; T: bst) return bst;
 function isnewtree(T: bst) return Boolean;
 function is_in(I: item; T: bst)
          return Boolean;
 private
  type node is
   record
    left: bst; info: item; right: bst
   end record;
  type bst is access node;
end bst_pkg;
```


An application of this generic package:

```
with TEXT_IO, bst_pkg;
use  TEXT_IO, bst_pkg;
procedure main is
 -- TEXT_IO, a predefined generic package,
 -- contains a generic package INTEGER_IO.
 -- A generic instantiation of INTEGER_IO:
 package INT_IO is new INTEGER_IO(integer);
 -- Import get, put from INT_IO:
 use INT_IO;
 -- A generic instantiation of bst_pkg:
 package INT_bst is new bst_pkg(integer);
 -- Import newtree, make etc. from INT_bst:
 use INT_bst;
  MY_T: bst;  -- Full type name: INT_bst.bst
  -- MY_T := newtree;
  -- MY_T := insert( 17, MY_T);
  -- put(data(MY_T));
    -- or, with fully qualified names:
    -- INT_IO.put(INT_bst.data(MY_T));
  -- and so on
```


Summary

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....