First balance sheet of a formal approach in the teaching of Data Structures

Marc Guyomard\textsuperscript{1,2}

\textit{Enssat/Irisa, Université de Rennes 1, BP 80518, 6, rue de Keramombok}
\textit{22305 Lannion, France}

P. Alain, A. Hadjali, H. Jaudoin\textsuperscript{3}

\textit{Enssat/Irisa, Université de Rennes 1, BP 80518, 6, rue de Keramombok}
\textit{22305 Lannion, France}

G. Smits\textsuperscript{3}

\textit{Enssat, Université de Rennes 1, BP 80518, 6, rue de Keramombok}
\textit{22305 Lannion, France}

Abstract

A formal method, \textit{EB}, inspired by B method, has been used for the first time for teaching “data structures”. Starting on one hand from a set oriented specification of an abstract type and on the other from the specification of an implementation, method \textit{EB} favours the calculation of the representation of concrete operations. We present the principle of this approach as well as several examples. The context of its application in a data structures teaching module is then detailed. A first balance sheet is drawn up.

Keywords: formal methods, teaching data structures, specification, refinement, homomorphism, B method, functional methods, program derivation, induction, sets, stacks, queues, priority queues, flexible arrays.

1 Introduction

Considered recently as an autonomic discipline, and taught as such \cite{2}, most often in the last year of the course in computer science, formal methods, and notably B \cite{1}, influence more and more the teaching of other modules in computer science. This shows itself either by the use of a formal method as cross support for the specification \cite{16}, or as a methodology support for a more rigorous approach to traditional modules \cite{15}.

\textsuperscript{1} Thanks to Henry Hicks
\textsuperscript{2} Email: Marc.Guyomard@univ-rennes1.fr
\textsuperscript{3} Email: \{Pierre.Alain,Melene.Jaudoin,Allel.Hadjali,Gregory.Smits\}@univ-rennes1.fr
In this article we relate an experiment of this type in which a formal approach, EB, inspired in particular from B method, has been used in teaching “data structures and algorithms”. This experiment was carried out at Enssat, a school for engineers housed in Lannion, France, which is part of the University of Rennes 1.

The second section presents the principles of the approach using a deliberately simple example. Section 3 describes the module and its context: objectives, duration, population, content, etc. Section 4 draws the first conclusions from the experiment, emphasising the point of view of the teacher. That of the students, still being analysed, will be developed during the oral presentation. Finally section 5 contains our conclusions.

2 EB: a tutorial

2.1 The principle of the approach

The idea of formally stating the specifications before developing from them an algorithmic solution has not been the subject for debate for a long time. Its translation in the domain of data structures or of data types has itself given rise to numerous works. Those which have led to algebraic data types are worth citing [13]. The same is true for the step proposed by T. Hoare in [17] which is presented as a method of verification based on the proof of the correctness and of the completeness of the concrete model compared with the simulated one. Attempts to bring the two steps together have been carried out [5].

If none of these approaches has become an accepted standard it is in our opinion because at least one of the two following criteria has not been met:

- The language used for specifying the data must be at a “reasonable level”. Neither too high (as in algebraic abstract types where the semantic gap to the objective is a problem), nor too low (as in predicate calculus which, in particular, leads to verbose wordings).
- The process of refinement (or of transformation) must be guided to be sufficiently constructive.

The EB approach defended here results from the worry of satisfying these two criteria in searching for, firstly from the B method [1] a satisfactory specification language for data and secondly from the Dutch school [19,22,8], a well identified succession of steps of refinement.

More precisely, the EB approach is based on an abstract specification and one refinement step. The abstract specification defines the support (the domain of possible values) as well as the operations, each of them in a functional way. The refinement consists of:

(i) Formally specifying the concrete support.

(ii) Formally specifying an “abstraction function” the source of which is the concrete support and the destination the abstract support. This abstraction function specifies which abstract value corresponds to which given concrete value.

(iii) Formally specifying each operation in the form of an equation which expresses the relations which maintain the concrete operation, the abstract operation
and the abstraction function.

(iv) Calculate the representation of each concrete operation.

Out of this development we have available a functional representation of each operation. It is then possible (and generally easy) to obtain a more classical version of type “modification in situ” (MIS).

2.2 Specification of abstract type NATABST

The example which is the linking thread in this section is that of natural numbers, on which we allow two operations: addition (plus) and the relational operation < (inf). It is important to remember that the operations are always functions.

In a first stage we specify the type NATABST in question. Figure 1 presents this specification. The heading gives the type name NATABST, the name of the support (natAbst, which represents all the values which can take an entity of type NATABST), the list of internal operations (that is the list of operations which produce a value of type under consideration) limited here to plus, and finally the list of external operations (reduced here to inf).

The heading support gives a definition of the support set. Here we affirm that “to say that \( n \) is a natural numbers is the same as saying that \( n \) belongs to the set natAbst”. The heading operations re-lists the operations in the heading and gives their name, those of their parameters and the category of the represented function (thus plus is a total surjective function from natAbst \( \times \) natAbst to natAbst: \( \to \)), the precondition, which includes the typing of the arguments (plus demands that the object values of the operation both belong to the support natAbst) and finally the definition properly said, expressed by a “functional equation” of which the right hand side is an expression of the same type as the range of the function.

```
abstractType NATABST= (natAbst, (plus), (inf))
uses
  bool, N
support
  n \in \mathbb{N} \iff n \in natAbst
operations
  1) name: plus(x, y) \to natAbst
     pre: x, y \in natAbst \times natAbst
     spec: plus(x, y) = x + y
  2) name: inf(x, y) \to bool
     pre: x, y \in natAbst \times natAbst
     spec: inf(x, y) = (x < y)
end
```

Figure 1: Specification of the abstract type NATABST

2.3 Specification of concrete type NATP

Once the specification has been created, we can envisage one (or several) implementation(s). This implementation cannot always represent faithfully the abstract type.\(^5\). In this section we propose an implementation of abstract type NATABST by

---

\(^4\) The functional character implies that an existing structure is never changed.

\(^5\) This is the case if the abstraction function is not a bijection. Such restrictions can be the result of physical limitations of the concrete type.
the concrete type natP (cf. fig. 2).

The heading of natP includes information similar to figure 1. The lists of abstract and concrete operations have a biunique relationship. The rubric support describes (in this example in an inductive manner) the concrete support. The third clause, that of closure, specifies that every element of natP is made up by applying the first two clauses a finite number of times. For example, zero ∈ natP and suc(suc(zero)) ∈ natP. The heading abstractionFunction defines the abstraction function A. A specifies how each element of natP can be considered as an element of natAbst. For example A(zero) = 0, A(suc(suc(zero))) = plus(plus(0, 1), 1) = 2. Here A is a bijection ⁶.

The heading operations takes up the different operations which are listed in the heading type. As far as the heading spec is concerned, we must distinguish between internal and external operations although in both cases this specification is systematic. The specification of the internal operation plus_p tells us that the abstract value which corresponds, by the abstraction function, to the concrete value obtained by use of the concrete function plus_p to two concrete arguments x and y is identical to the abstract value obtained by the application of the abstract function plus to two arguments obtained by application of the abstraction function to concrete arguments x and y. In the same way the specification of the external operation inf_p teaches us that the result obtained by application of the function inf_p to two concrete arguments x and y is identical to the result obtained by application of the abstract

<table>
<thead>
<tr>
<th>type</th>
<th>NATP = (natP, (plus_p), (inf_p))</th>
</tr>
</thead>
<tbody>
<tr>
<td>refines</td>
<td>NATAST</td>
</tr>
<tr>
<td>uses</td>
<td>bool</td>
</tr>
<tr>
<td>support</td>
<td></td>
</tr>
</tbody>
</table>
1) n = zero ⇒ n ∈ natP
2) n ∈ natP ⇒ suc(n) ∈ natP
3) closure
| abstractionFunction |
| heading: A(n) ↦ natAbst |
| pre: n ∈ natP |
| rep: A(n) = |
| if n = zero ⇒ |
| 0 |
| if n ≠ zero ⇒ |
| Let n = suc(n′) |
| plus(A(n′), 1) |
| fi |
| operations |
1) name: plus_p(x, y) ↦ natP |
| pre: x, y ∈ natP × natP |
| spec: A(plus_p(x, y)) = plus(A(x), A(y)) |
2) name: inf_p(x, y) ↦ bool |
| pre: x, y ∈ natP × natP |
| spec: inf_p(x, y) = inf(A(x), A(y)) |

Figure 2: Specification of the concrete type natP

⁶ Which means that we define in this way an isomorphism between the abstract structure and the concrete one.
operation $\inf$ to the two abstract arguments resulting from the application of the abstraction function to $x$ and $y$.

2.4 Calculation of the operations of concrete type $\text{NATP}$

The next step consists in calculating a functional representation of each operation. This calculation is based on the use of Leibniz’s axiom [12] which states that if $f$ is a function and if $x, e \in \text{dom}(f) \times \text{dom}(f)$ then

$$x = e \Rightarrow f(x) = f(e)$$

The application of this axiom to our problem consists in identifying $x$ to the internal concrete operation $(io_c)$, considered as the unknown, and the function $f$ to the abstraction function $A$. If we can produce an expression $e$ such that $A(io_c(...)) = A(e)$ then a possible solution for $io_c(...)$ is $e$. In general the search for the expression $e$ demands that we proceed by induction on the arguments of the operation $io_c$ or by a case analysis aiming to take into account the different possible values for these arguments. The result of this is an iterative step, in which each stage gives a solution guarded by the condition which acts as an hypothesis for the calculation. When all cases have been taken into account it is necessary to group the different results in order to make up the functional representation of the operation under consideration.

2.4.1 Calculation of the operation $\text{plus}_p$

Let us show how this step leads us to calculate a representation of the operation $\text{plus}_p$. We start from the expression $A(\text{plus}_p(x, y))$: 

$$A(\text{plus}_p(x, y))$$

- - specification of the operation $\text{plus}_p$, figure 2, page 4

$$\text{plus}(A(x), A(y))$$

- - definition of the operation $\text{plus}$, figure 1, page 3

$$A(x) + A(y)$$

It is, however, difficult to follow on without an hypothesis on the structure of $x$ and/or $y$. We choose to proceed by structural induction on $x$. We must consider two cases: $x = \text{zero}$ and $x \neq \text{zero}$. We start with $x = \text{zero}$.

$$A(x) + A(y)$$

- - hypothesis ($x = \text{zero}$)

$$A(\text{zero}) + A(y)$$

- - definition of the abstraction function $A$, figure 2, page 4

$$0 + A(y)$$

- - arithmetic

$$A(y)$$

Thus for $x = \text{zero}$ we have:

$$A(\text{plus}_p(x, y)) = A(y)$$
Now
\[ \mathcal{A}(\text{plus}\_p(x, y)) = \mathcal{A}(y) \]
\[ \Leftarrow \text{- - Leibniz} \]
\[ \text{plus}\_p(x, y) = y \]
This gives us the first guarded equation of the function \text{plus}\_p:
\[ x = \text{zero} \rightarrow \text{plus}\_p(x, y) = y \]
The induction case \((x = \text{suc}(x'))\) is treated as follows:
\[ \mathcal{A}(x) + \mathcal{A}(y) \]
\[ = \text{- - hypothesis } (x = \text{suc}(x')) \]
\[ \mathcal{A}(<x'>) + \mathcal{A}(y) \]
\[ = \text{- - definition of the abstraction function } \mathcal{A}, \text{ figure 2, page 4} \]
\[ \text{plus}(\mathcal{A}(x'), 1) + \mathcal{A}(y) \]
\[ = \text{- - definition of the operation plus, figure 1, page 3} \]
\[ \mathcal{A}(<x'>) + 1 + \mathcal{A}(y) \]
\[ = \text{- - arithmetic} \]
\[ \mathcal{A}(<x'>) + \mathcal{A}(y) + 1 \]
\[ = \text{- - definition of the operation plus, figure 1, page 3} \]
\[ \text{plus}(\mathcal{A}(<x'>), \mathcal{A}(y)) + 1 \]
\[ = \text{- - specification of the operation plus\_p, figure 2, page 4} \]
\[ \mathcal{A}(\text{plus}\_p(<x'>, y)) + 1 \]
\[ = \text{- - definition of the operation plus, figure 1, page 3} \]
\[ \text{plus}(\mathcal{A}(\text{plus}\_p(<x'>, y)), 1) \]
\[ = \text{- - definition of the abstraction function } \mathcal{A}, \text{ figure 2, page 4} \]
\[ \mathcal{A}(<\text{plus}\_p(<x'>, y)>)) \]
From Leibniz we obtain the second guarded equation for operation \text{plus}\_p:
\[ x \neq \text{zero} \rightarrow \]
\[ \text{Let } x = \text{suc}(x') \]
\[ \text{plus}\_p(x, y) = \text{suc}(\text{plus}\_p(x', y)) \]
We have calculated in all the representation of figure 3 for the operation \text{plus}\_p.
2.4.2 Calculation of the operation \( \text{inf}_p \)

External concrete operations (\( \text{eo}_c \)) are easier to deal with since Leibniz’ axiom is not needed. All that is needed is to calculate an expression \( e \) such that \( \text{eo}_c(\ldots) = e \). For the operation \( \text{inf}_p \), starting from \( \text{inf}_p(x, y) \) and carrying out a first structural induction on \( y \) then a second on \( x \) we obtain the solution of figure 4.

2.5 Transformation of the operations

We now have available a definite type \( \text{natP} \) to represent natural numbers. This type can, with the appropriate notation, be included as it is in most programming languages (and especially in functional languages). However, for more efficiency (cf. [18]), we can transform the existing operations to allow sequencing, variables and the consequences which flow from them (especially iteration) while at the same time suppressing – if possible – recursion. At this stage in the development, we are ready to make a choice for the low level representation of the support of the type under consideration. We will be directly making the different transformations which will lead us to the final version of the type.

As regards support we have chosen a solution using a chained list such that the length of the list represents the number under consideration. Thus, using an ADA-like formalism:

```ada
type box;
type lnk is access box;
type box is record
  next: lnk;
end record;
```

Let \( \mathcal{A}' \) be the abstraction function for this stage:
heading: $A'(x) \rightarrow \text{natP}$
pre: $x \in \text{lnk}$
rep: $A'(x) =$
  if $x = \text{null} \rightarrow$
    $\text{zero}$
  $| x \neq \text{null} \rightarrow$
    Let $x = \text{box}'(\text{next} \Rightarrow x')$
    $\text{suc}(A'(x'))$
fi

As regards the transformation of the operation $\text{plus}_p$, the first stage consists in obtaining a procedure by adding an output parameter $\text{res}$:

procedure $\text{plus}_p'(x, y, \text{res} : \text{out}) \equiv$
pre
  $x, y, \text{res} \in \text{natP} \times \text{natP} \times \text{natP}$
then
  var
    $\text{aux} \in \text{natP}$
  in
    if $x = \text{zero} \rightarrow$
      $\text{res} := y$
    $| x \neq \text{zero} \rightarrow$
      Let $x = \text{suc}(x')$
      $\text{plus}_p'(x', y, \text{aux})$;
      $\text{res} := \text{suc}(\text{aux})$
fi
end
end

It is then easy to use the implantation type $\text{lnk}$ in order to obtain a non-destructive version $\text{plus}_p''$. This is what can be found in figure 5.

The transformation of the operation $\text{inf}_p$ goes through an intermediate stage where a version satisfying the conditions of removing tail recursion is obtained (cf. for example [6]). The final, iterative, version is as it is shown in figure 6.

3 Application to the teaching of data structures

In this section we present the teaching module, its contents, organisation and development.

3.1 The teaching module proper

This module is aimed at first year student engineers, specialising in information technology (equivalent, in England, to the last year of a Bachelor of Science course and in the US to the third year). It takes place during the second semester.

The origin of the population is twofold. Some of the students come from the “preparatory classes for the Grandes Écoles” and have had little experience in programming before. On the other hand, these students are supposed to be good at
mathematics. The rest of the target group have a DUT (Diplôme Universitaire de Technologie) in computers, which is a professional qualification in IT. These students have, in general, taken a course with the same title as this module. But their level in mathematics is lower.

On arrival at ENSSAT all students take a basic course in programming. This
follows a semi-formal approach [15]. The students from the preparatory classes then follow complementary modules in programming design to familiarise them with the ideas of elementary data structures, of pointers and dynamic memory allocation.

The teaching of the data structures’ module, out of a total of 66 hours, is made up of 22 hours of lectures, 18 hours of supervised activities, 24 hours of practicals followed by an exam lasting 2 hours, where personal documentation is allowed.

Up until this scholastic year the teaching of the module was more traditional, quite similar to what is recommended in [3]: basic data structures were presented using algebraic abstract data types [11,13,7] followed by a semi-formal refinement, by structural induction, on data structures at a basic level.

3.2 Lectures

The plan that we had anticipated is shown in figure 7. In section 4, page 23, we will see that we have had to limit our ambitions. Section 1.2 of figure 7 develops some simple examples, in the style of the example shown in section 1 above. Section 1.3 of figure 7 makes the students aware of two types of step (functional and MIS), at the same time showing how a classical programming language can be used as much as in one style as in another. Section 1.4 of figure 7 has been little developed because it is dealt with specifically in a module “Discrete Mathematics oriented toward Computer Science”. Section 2 of figure 7 presents the principal set notations of B method as well as the extensions such as array shifting 7, conditional expressions, nondeterministic expressions 8, inductive structures and multiset (bag) notations. The data structure “list” is also studied which include catenation $l_1 \sim l_2$, inversion $l^{-1}$ and casting to set 9. The main properties (as, for example, $(a \sim b)^{-1} = (b^{-1} \sim a^{-1})$) are proved or presented.

The choice of data structures studied comes from [26]. The partition data structure has been put to one side for reasons of time. On the other hand the data structure “graphs”, also absent from this module, is the object of a particular module in the second year.

In section 3 of the plan of figure 7 we introduce informally the data structures selected before presenting them formally. Section 4, “implementation of fundamental data structures” develops some traditional implementations from among the most efficient. At this time the students are warned that the quality of a data structure is intimately linked to the group of operations selected: adding a new abstract operation could bring to naught the research attempts for efficiency of a concrete support.

We propose at present a synthesis of the contents of sections 3 and 4 from the plan of figure 7 resuming each of the data structures studied while limiting ourselves, for the abstract specification, to the heading and to the support, and for the concrete specification, to the heading, to the support and to the abstraction

---

7 For that we use the abbreviation $f \gg v$ defined for $v \in \mathbb{Z}$ and for any function $f$ such that $\text{dom}(f) \subseteq \mathbb{Z}$. By definition $f \gg v = (\lambda j \cdot (j \in \text{dom}(f) \mid j + v))^{-1} f$. In other words, if $f$ is an array, $f \gg v$ returns an array of which the domain of definition is shifted by $v$ compared to $f$, while retaining the values of array $f$.

8 This notation modifies the unbounded choice substitution of method B to make it into an unbounded choice of expressions.

9 $\text{set}(l)$ returns the set of the elements present in list $l$. 
1 Introduction
   1.1 Why a data structures module?
   1.2 General principals of the EB approach
   1.3 Functional approach vs MIS approach
   1.4 Complexity
2 Notations and tools
   2.1 Specification
   2.2 Operation representation
3 Fundamental data structures: specification
   3.1 Sets (of scalars and strings)
   3.2 Stacks
   3.3 Queues
   3.4 Priority queues
   3.5 Flexible arrays
4 Implementation of fundamental data structures
   4.1 Sets
   4.2 Stacks
   4.3 Queues
   4.4 Priority queues
   4.5 Flexible arrays

function. We assume the reader is familiar with B notations as well as with the abstract and concrete structures developed. We will not go into them unless we think it is necessary.

3.3 Sets

Historically, sets of (totally ordered) scalars were the first abstract data structures which interested computer scientists. They naturally concentrated on efficient implementations such as binary search trees, AVLs, 2-3 trees, B-trees, hash tables and so on. In the lectures we limit ourselves to the implementation by characteristic vectors, by binary search trees and by B-trees. However, see [14] for a formal development of two other implementations: by array and by AVL.

3.3.1 Sets of scalars: abstract specification

The abstract support identifies the type setAbst to the set of finite subsets of natural numbers.

\[
\text{abstractType SETABST} = (\text{setAbst}, (\text{clear}, \text{insert}, \text{remove}), (\text{isIn}, \text{isEmpty}))
\]

\[
\text{support} \quad e \in \mathbb{F}(\mathbb{N}) \iff e \in \text{setAbst}
\]
3.3.2 Sets of scalars: concrete specification by characteristic vectors
Several types of implementation by arrays are conceivable, few of them are efficient. But when the elements to be represented belong to a reasonable interval, an efficient solution exists for most of the operations. This consists of using an array of Booleans such that the value subscripted by \( i \) indicates the presence, or not, of \( i \) in the subset under consideration. This technique is known as the “characteristic vector method”. The abstraction function \( \mathcal{A} \) identifies the set represented to the set of subscripts corresponding to the value \text{true} in the array used as support.

\[
\text{type} \quad \text{setCV}(n)= \\
(setCV, (\text{clear}_\text{cv}, \text{insert}_\text{cv}, \text{remove}_\text{cv}), (\text{isIn}_\text{cv}, \text{isEmpty}_\text{cv})) \\
\ldots \\
\text{support} \\
a \in 1..n \rightarrow \text{bool} \iff a \in \text{setCV}(n) \\
\text{abstractionFunction} \\
\text{heading}: \mathcal{A}(a) \rightarrow \text{setAbst} \\
\text{pre}: a \in \text{setCV}(n) \\
\text{rep}: \mathcal{A}(a) = \text{dom}(a \triangleright \{\text{true}\})
\]

This solution is in general very efficient, save for the operation \text{clear}_\text{cv} which demands a traverse of the array.

3.3.3 Sets of scalars: concrete specification by binary search trees
The support \( \text{setBst} \) is described without difficulty by induction. The abstraction function \( \mathcal{A} \) which gives the set of values present in the tree is defined by structural induction on its argument.

\[
\text{type} \quad \text{setBst} = \\
(setBst, (\text{clear}_\text{st}, \text{insert}_\text{st}, \text{remove}_\text{st}), (\text{isIn}_\text{st}, \text{isEmpty}_\text{st})) \\
\ldots \\
\text{support} \\
1) e = \langle \rangle \Rightarrow e \in \text{setBst} \\
2) n \in \text{N} \land l \in \text{setBst} \land r \in \text{setBst} \land \max(\mathcal{A}(l)) < n \land \min(\mathcal{A}(r)) > n \\
\Rightarrow \\
\langle l, n, r \rangle \in \text{setBst} \\
3) \text{closure} \\
\text{abstractionFunction} \\
\text{heading}: \mathcal{A}(x) \rightarrow \text{setAbst} \\
\text{pre}: x \in \text{setBst} \\
\text{rep}: \mathcal{A}(x) = \\
\text{if } x = \langle \rangle \rightarrow \\
\emptyset \\
\text{else } x \neq \langle \rangle \rightarrow \\
\text{Let } x = \langle l, n, r \rangle \\
\mathcal{A}(l) \cup \{n\} \cup \mathcal{A}(r) \\
\text{fi}
\]
The operation $insert\_st$ is stated in two forms: insertion at the leaves and at the root. For this last version two techniques are studied: by cutting and by rotations. Random insertion is simply referred to.

3.3.4 Sets of scalars: concrete specification by B-trees

B-trees have first been described in an article by R. Bayer and E. McCreight in 1972 [4]. Since then, numerous variants of B-trees have been suggested. We will consider B-trees of order $n$ ($n > 0$) which are characterised by the following properties:

1. For any given node, all sub-trees have the same height.
2. Each node has between $n$ and $2n$ values, except the root.
2 b. The root has between 1 and $2n$ values.
3. The values at the nodes are sorted in strict ascending order.
4. Each node made up of $p$ values refers to $p+1$ B-trees, all of the same height.
4 b. Each leaf made up of $p$ values is linked to $p+1$ empty B-trees.
5. A B-tree is a search tree: the inorder traversal encounters values in strict ascending order.

![Figure 8. An example of 2-ordered B-tree](image)

The main difficulties in the definition of the support come from the fact that (1) the root of a B-tree could have only one element, (2) at the time of an update (an insertion, for example) the structure could not be a strict B-tree: a node can momentarily become overflowed. The parameters of type $nBT$ take into account these particularities: $n$ is the order of the B-tree, $low$ the minimum number of elements of a node, $up$ the maximum number. $tt$ is an auxiliary type which allows the description of nodes. $h(t)$ is the function which produces the height of the tree $t$. In the rest of the paper, contrary to B, we admit, together with [10], that if $s \subset \mathbb{N} \land s = \emptyset$ then $\max(s) = 0$ and $\min(s) = +\infty$.

The abstraction function, on the other hand, presents no difficulties: it is defined by structural induction on the support $nBT$. 
Guyomard

type NBT(n, low, up) =
    (nBT, (clear_bt, insert_bt, remove_bt), (isIn_bt, isEmpty_bt))

constraints
    n ∈ ℕ₁ ∧
    low .. up ∈ \{1 .. 2.n, n .. 2.n, 1 .. 2.n + 1, n .. 2.n + 1\}
    ...

support
    a ∈ 1 .. 2.n + 1 → ℕ ∧ sz ∈ low .. up ∧ ∀i·(i ∈ 1 .. sz − 1 ⇒ a(i) < a(i + 1))
    ⇔
    a ∈ tt(sz)
    1) t = {} ⇒ t ∈ nBT(n, low, up)
    2) t = \{te, tl, sz\} ∧
       sz ∈ low .. up ∧
       te ∈ tt(sz) ∧
       tl ∈ 0 .. 2.n + 1 → nBT(n, n, 2.n + 1) ∧
       ∀i·(i ∈ 1 .. sz ⇒ h(tl(i)) = h(tl(0))) ∧
       ∀i·(i ∈ 1 .. sz ⇒ max(A(tl(i − 1))) < te(i) ∧ min(A(tl(i))) > te(i))
    ⇒
    t ∈ nBT(n, low, up)
    3) closure

abstractionFunction
    heading: \(A(x) \rightarrow \text{setAbst}\)
    pre: \(x \in nBT(n, low, up)\)
    rep: \(A(x) =\)
    if \(x = {}\) →
        ∅
    | \(x \neq {}\) →
        Let \(x = \{te, tl, sz\}\)
        te[1 .. sz] ∪ \bigcup i·(i ∈ 0 .. sz | A(tl(i)))

In the lectures we limit ourselves to the calculation of the operation \(insert_{bt}\)
in its ascending version\(^{10}\). The possibility of descending versions or of exploiting
rotations is only mentioned.

3.3.5 Sets of strings
Sets of strings are also broached. The implementation used is that of tries. However,
Patricia tries [24] are not explained.

3.4 Stacks
For an experienced computer scientist the choice of stacks as a fundamental data
structure is unavoidable. This is not the case for beginners for whom the experience
that they have of recursive procedures is often not enough for them to understand
the link between embedded structures and stacks.

\(^{10}\)Which means the possible splitting of a node happens when ascending.
3.4.1 Stacks: abstract specification
A stack is specified as a pair, the first element of which is a function whose upper limit is not restricted and the second is a pointer to the last element recorded. The domain restriction on the interval going as far as the pointer is an array.

\[
\text{abstractType } \text{STACKABST}(T) = \\
(\text{stackAbst}, (\text{clear}, \text{push}, \text{pull}), (\text{topVal}, \text{isEmpty})) \\
\ldots \\
\text{support} \\
t \in \mathbb{N}_1 \rightarrow T \land ip \in \mathbb{N} \land 1 \ldots ip < t \in 1 \ldots ip \rightarrow T \Leftrightarrow (t, ip) \in \text{stackAbst}(T)
\]

3.4.2 Stacks: concrete specification by lists
The implementation by arrays is very close to the abstract specification. For this reason its development has no great interest, it is simply mentioned in the lectures. The only implementation studied is that based on lists. Its characteristics are shown below.

\[
\text{type } \text{STACKL}(T) = \\
(\text{stackL}, (\text{clear}_L, \text{push}_L, \text{pull}_L), (\text{topVal}_L, \text{isEmpty}_L)) \\
\ldots \\
\text{support} \\
1) a = \langle \rangle \Rightarrow a \in \text{stackL}(T) \\
2) v \in T \land p \in \text{stackL}(T) \Rightarrow \langle v, p \rangle \in \text{stackL}(T) \\
3) \text{closure} \\
\text{abstractionFunction} \\
\text{heading}: \ A(s) \mapsto \text{stackAbst}(T) \\
\text{pre}: \ s \in \text{stackL}(T) \\
\text{rep}: \ A(s) = \\
\text{if } s = \langle \rangle \rightarrow \\
\text{any } t \text{ where } t \in \mathbb{N}_1 \rightarrow T \text{ then } (t, 0) \text{ end} \\
| s \neq \langle \rangle \rightarrow \\
\text{Let } s = \langle v, p \rangle \text{ and } A(p) = (t, ip) \\
(t \leftarrow \{ (ip + 1, v) \}, ip + 1) \\
\text{fi}
\]

The students are told that this implementation is very efficient, but as soon as one tries to extend the range of operations, for example by allowing access to a value other than the one at the top of the stack, the solution by list has to be abandoned.

3.5 Queues
3.5.1 Queues: abstract specification
The justification of the choice of queues as a fundamental data structure is easy to understand, examples from daily life are sufficiently illustrative.

Below, the abstract specification is made on the lists’ base. The elements of queues are keys, consequently an element cannot appear several times in the list.
This property materialises in the support by the existence of the conjunct \( v \notin \text{set}(l) \) which prohibits duplicates in the list. From this support it is easy to specify the operations, thus the insertion (at the end of the queue \( q \)) of the value \( v \) is expressed by \( \langle v, q^{-1} \rangle \).

There are at least three different representations of queues: (1) the representation by array, with a pointer to the head and another to the tail, (2) the representation by list with insertion at the head of the list and suppression from the tail, and, less often taught, (3) representation by double lists. We have decided to select solution (1), to put to one side solution (2) (solutions using pointers or by \texttt{deque} are however possible, cf. \cite{8}), and to detail solution (3) which has the added advantage of a nice introduction to the theory of amortized complexity.

3.5.2 Queues: concrete specification by arrays

Conceptually simple, this solution has, however, technical difficulties. The first being that it is necessary, in the definition of support, to distinguish the case where the pointer to the head is less than or equal to the pointer to the tail from the complementary case. The second difficulty comes from the fact that the nature of the abstraction function \( \mathcal{A} \) demands that the reverse function \( \mathcal{A}^{-1} \) is also specified.

\[
\text{type } \text{QUEUEA}(T, n) = \\
(queueA, (\text{clear}_a, \text{insert}_a, \text{remove}_a), (\text{head}_a, \text{isEmpty}_a))
\]

\[
\text{support} \\
1) \quad a = \langle \rangle \Rightarrow a \in \text{queueAbst}(T) \\
2) \quad v \in T \land l \in \text{queueAbst}(T) \land v \notin \text{set}(l) \Rightarrow \langle v, l \rangle \in \text{queueAbst}(T) \\
3) \quad \text{closure}
\]

3.5.2 Queues: concrete specification by arrays

Conceptually simple, this solution has, however, technical difficulties. The first being that it is necessary, in the definition of support, to distinguish the case where the pointer to the head is less than or equal to the pointer to the tail from the complementary case. The second difficulty comes from the fact that the nature of the abstraction function \( \mathcal{A} \) demands that the reverse function \( \mathcal{A}^{-1} \) is also specified.

\[
\text{type } \text{QUEUEA}(T, n) = \\
(queueA, (\text{clear}_a, \text{insert}_a, \text{remove}_a), (\text{head}_a, \text{isEmpty}_a))
\]

\[
\text{support} \\
\text{if } f \in 1..n \to T \land h \in 1..n \land t \in 1..n \land \\
(h \leq t \Rightarrow h..t-1 \cup f \in h..t-1 \to T) \land \\
(h > t \Rightarrow t..h-1 \cup f \in (1..t-1 \cup h..n) \to T) \\
\Rightarrow \\
(f, h, t) \in \text{queueA}(T, n)
\]

\[
\text{abstractionFunction} \\
\text{heading: } \mathcal{A}((f, h, t)) \mapsto \text{queueAbst}(T) \\
\text{pre: } (f, h, t) \in \text{queueA}(T, n) \\
\text{rep: } \mathcal{A}((f, h, t)) = \\
\text{if } h = t \to \\
\langle \rangle \\
\text{if } h \neq t \to \\
\langle f(h), \mathcal{A}((f, h+1 \mod n, t)) \rangle
\]
3.5.3 Queues: concrete specification by double lists

To implement directly the specification of a queue using one list would lead, for the insertion operation, to traversing the list to its end before carrying out the insertion itself. The cost of such an operation is prohibitive. An other solution consists in managing two lists, one for suppression and the other for insertion. Manipulation is always done at the head: suppression is done at the head of the first list and addition at the head of the second. However, at the time of a suppression, if the first list is empty it is necessary to invert the second into the first as it is shown in the following example:

<table>
<thead>
<tr>
<th>Operations</th>
<th>Concrete representation</th>
<th>abstract queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion of 2</td>
<td>([4,8],[7,3])</td>
<td>→ [4,8,3,7]</td>
</tr>
<tr>
<td>Removal</td>
<td>([4,8],[2,7,3])</td>
<td>→ [4,8,3,7,2]</td>
</tr>
<tr>
<td>Removal</td>
<td>([8],[2,7,3])</td>
<td>→ [8,3,7,2]</td>
</tr>
<tr>
<td>Removal</td>
<td>([],[2,7,3])</td>
<td>→ [3,7,2]</td>
</tr>
<tr>
<td>Removal</td>
<td>([3,7,2],[])</td>
<td>→ [7,2]</td>
</tr>
</tbody>
</table>

In supposing the data structure “list” is available, the concrete specification is simple. The abstraction function defines the abstract queue as the catenation of the first list and the inverse of the second.
3.6 Priority queues

Priority queues are the object of much scientific literature. Several implementations have been suggested. They are distinguished one from another by the existence or not of the merging operation. If merging is not part of the set of operations, a simple solution exists: perfect heaps. In the opposite case efficient solutions are based on sophisticated data structures. In the lectures we study one from among them: binomial queues [25,26].

3.6.1 Priority queues: abstract specification

Conceptually a priority queue is similar to a bag (multiset). In order to avoid an implementation by a double refinement, we have decided to consider that the everyday notations on the bags are available in the language used for specification. To make them easier to remember by the students we have used the following convention: a multiset symbol is the “squared” counterpart of the set symbol (thus $\emptyset$ corresponds to $\emptyset$, $\cup$ corresponds to $\cup$, $\{}$ corresponds to $\{}$, etc.). As regards $\mathbb{B}(\mathbb{N})$ it represents the finite set of bags of natural numbers.

abstractType PQABST =
(pqAbst, (clear, insert, remove, merge), (prioVal, isEmpty))

... support
$q \in \mathbb{B}(\mathbb{N}) \quad \Leftrightarrow \quad q \in pqAbst$

3.6.2 Priority queues: concrete specification by binomial queues

Binomial queues are defined by cross recursion with binomial heaps. A binomial heap (of support $bH$) is a couple with one part made up by a natural number value and the other by a binomial queue. The whole makes a heap. $h$ is the function which produces the height of the tree.

The use of a binomial heap demands the use of the function $bh$ which returns the bag of values present in the tree. A binomial queue is defined as a list (possibly empty) of binomial heaps in strict height descending order. The example in figure 9, page 20, shows a binomial queue made up of 4 binomial heaps of heights 4, 3, 2,
and 1 respectively. It is defined inductively, using the function bh to convert the binomial heap at the head of the queue.

The merging operation plays a central role in the calculation of operations. In fact, insertion can be defined as the merging of the queue with the value to be added, converted to a binomial queue. Suppression can in the same way be defined as the suppression of the smallest value followed by a merging of 1) the initial queue deprived of the heap affected by the suppression, and 2) of the heap deprived of its root. The operation merge\(_{bq}\) obtained by calculation is slightly different from that supplied in [25]. Moreover it makes explicit use of the function \(h\). This solution is only viable under the condition that, at the time of looking for the eventual implantation data structure, the calls to \(h\) can be replaced by the consultation of a field in the data structure. This is the case since the function \(h\) is decomposable (cf. [8]). We thus obtain a very efficient solution for all operations.
3.7 Flexible arrays

In its standard version a flexible array is a data structure which, following the example of a single dimensional array, is defined as a total function on the domain $1..n$ ($n \geq 0$), but with the difference that this one allows an insertion in the interval $1..n+1$ or a removal in the interval $1..n$, bringing with it a variation of the upper limit $n$ at run time. Our specification is inspired by [26] but restricted interesting forms of flexibility exist such as [19,8].

3.7.1 Flexible arrays: abstract specification

The operation $\text{insert}(v, i, a)$ inserts the value $v$ in the flexible array $a$ at position $i$; the values initially in position after $i-1$ have their subscripts increased by one. The array’s length is increased by one. Inversely the operation $\text{remove}(i, a)$ deletes the element in position $i$ and the values placed initially after $i$ have their subscripts reduced by one. The array’s length decreases by one. The operation $\text{val}(i, a)$ returns the value in position $i$.

```
abstractType \text{FAABST}(T, s) =
    (\text{faAbst}, (\text{clear, insert, remove, catenate}), (\text{val, #}))
\cdots

support
    t \in \mathbb{N}_1 \rightarrow T \land s \in \mathbb{N} \land 1..s \triangleleft t \in 1..s \rightarrow T \iff t \in \text{faAbst}(T, s)
```

A flexible array is specified as a total function defined on the domain $1..s$.

3.7.2 Flexible arrays: concrete specification by lists

The implementation of flexible arrays by lists defines the abstraction function $\mathcal{A}$ by induction on the rest of the list using the shifting operation $\gg$. 
This solution is obviously costly in so far as all the direct access to any position in the array is obtained by sequential access to the list.

3.7.3 Flexible arrays: concrete specification by heaps
An acceptable solution is obtained by adopting as support a heap memorising all the values in the array.

```plaintext
type FAHP(T) =
(faHp, (clear_hp, insert_hp, remove_hp, catenate_hp), (val_hp, #hp))

support
1) a = ⟨⟩ ⇒ a ∈ faHp(T)
2) n ∈ T ∧ l, r ∈ faHp(T) × faHp(T) ∧
   n ≤ max(set(l)) ∧ n ≤ max(set(r))
   ⇒
   ⟨l, n, r⟩ ∈ faHp(T)
3) closure
abstractionFunction
heading: A(a) → faAbst(T, s)
pre: a ∈ faHp(T) ∧ s ∈ N ∧ s = #a
rep: A(a) =
if a = ⟨⟩ →
1..0 ⊖ ∅
| a = ⟨⟩ →
Let a = ⟨l, n, r⟩
A(l) ≜ \{ (w(l) + 1, n) \} ≜ A(r) ≃ w(l) + 1
fi
```

Guyomard
Formalising the support uses the operation set on the tree in order to express the heap condition. set(t) provides the set of the values belonging to the heap t. The abstraction function is defined by structural induction on the heap. It uses the function w which returns the weight of the heap passed as an argument.

This method avoids having to memorise explicitly the absolute value of each subscript, a solution which brings with it a prohibitive cost for updating operations. The weight w ensures the relation between the subscript of an element in the flexible array and its position in the heap. The call to function w appears explicitly in the operations as they are derived. The remark on page 19 about the function h applies here to the function w: the result is only viable on the condition that the call to the function w could be replaced by an equivalent field in the final data structure. It is possible since the function w is, for a heap, separable. We thus obtain a solution which is efficient on average but which could degenerate into $O(n)$ in the worst case.

The teaching support offered to the students for the lectures are, on the one hand, a hand-out containing the essential points of the course (abstract and concrete specifications, functional representation of operations) and on the other hand a Web access to the development of operations, allowing them to go over the course at home.

3.8 Supervised work

The 18 hours of supervised work have, as an objective, allowing the student to understand in depth the lectures by the intermediary of assimilation and application exercises. The 18 hours are made up as shown in figure 10.

Here is an example of an exercise suggested in section 1 of the supervised works:

Let $n \in \mathbb{N}$ and $\text{tab}$ the support defined by:

$$ t \in 1 \ldots n \rightarrow \mathbb{N} \iff t \in \text{tab}. $$

Let $s \in \text{tab}$. Specify the operation $\text{sevenBefore23}$ which returns an array holding the same bag of values as $s$ but such that all the 7s come before all the 23s.

In section 2, figure 10, we are studying lists and we show some of their properties.
We also make up some simple exercises on binary trees. In sections 4 to 8 we perform exercises which complete the developments carried out in the lectures (calculation of a new version of an operation, or calculation of a yet undeveloped operation). We also suggest new data structures as, for example, sets of couples, or queues with tokens.

3.9 Practical work

The objective of the practical work is to familiarise the students with the practical aspects of data structures. A typical session would consist of, starting from a data structure in which all the operations have been calculated, supplying the development necessary in order to obtain either a functional solution or an MIS solution for this data structure. In the first two sessions simple solutions are developed in parallel using the two methods. The last sessions are allotted to projects in which all the conception and the development are to be done following a given specification. As an example, the students have to develop a key based priority queue where an already present client could take place in the queue with a different priority. The students work on their own.

4 First appraisal

As we have said in the introduction, the appraisal which we present here reflects principally the observations and the points of view of the teachers of the module. The general feeling of the contributors is that this method has re-awakened interest in a scientific area which has suffered from routine. The following points are those most mentioned:

(i) The amount of time allowed for lectures was enough for about 60% of the course previously covered. Replying off the cuff, our strategy has been to maintain the depth of knowledge at the expense of the width. Consequently, the implementation of certain data structures has had to be transferred to supervised work (as for example is the case for stacks or queues by arrays). Others have purely and simply disappeared (as is the case for sets of strings by tries).

(ii) One classical consequence of the use of a formal approach is the ease with which properties are expressed. Using EB, this advantage is accentuated by the functional character of the operations as far as it is possible to formulate a property bearing on the “new” structure as well as on the “old”, the two existing together. Thus, if one wishes to state that the insertion in an AVL increases the height of the tree of in most one, it is possible to simply write

\[ h(\text{insert}_{\text{avl}}(v, a)) - h(a) \in \{0, 1\} \]

(iii) The EB approach looks fairly useless in the discovery of efficient data structures. This is a privilege which remains the sole right of the human being. On the other hand, we have noticed that the operations obtained by calculation can bring something new compared with known solutions. Thus, we have noticed that the operation of merging binomial queues obtained by calculation
Guyomard is different from those presented in [25]. The latter, in adopting the metaphor of addition suggests a solution which goes from right to left. Our solution produces a calculation which advances from left to right.

This is also the case in the implementation of sets by AVL (cf. [14]): the insertion version obtained by calculation does not demand, in contrast to traditional versions, to convey a parameter indicating if the height has changed or not, since the functional character of the operation means that we have access to the height of the new tree as well as that of the old one.\footnote{Of course, this advantage disappears if the operation loses its functional character.}

(iv) The use of formal approaches leads to a beneficial change in the way of thinking. This has often been observed (cf. [6,9,20,10]).

Concerning an appraisal which would reflect the students’ point of view, a questionnaire will be provided for all at the end of the module and the results will be analysed carefully. Some discussions at the end of the course have allowed a glimpse of their impressions. Some of these points are:

- Difficulty in understanding the nature of and the interest in a specification.
• A feeling of surprise that programs can be calculated ("but what do you mean by calculate programs?").

• Some difficulty in the beginning to absorb the formalisms and the derivational style, but for most a fairly quick assimilation (about 6 hours of supervised work).

• Hermetism in the face of the difficulties and of the issues of the discipline. Let us illustrate with a dialogue showing what happens at the end of a course where the teacher had spent about one hour constructing and producing a program of about 15 lines:

  **Student:** Does it always take that long to develop a program?

  **Teacher:** Yes, a program is the often the result of complex reasoning. It is necessary to explain the different aspects of its construction.

  **Student:** But I can do that in 5 minutes!

5 Conclusion

The scientific character of our discipline is a reality, every computer scientist feels this intimately. It is however a facet of our activity which is only too rarely shown. The main reason is the excessive room offered to descriptive approaches, notably for those to do with the teaching of data structures. We have tried to show that there is no inevitability in this state of affairs. We have applied the formal approach EB for the first time in the teaching of data structures aimed at student engineers starting a course of three years. The principle chapter headings of an existing course have been preserved, the contents have been adapted to EB, offering an homogenous presentation of each data structure under the form:

(i) Informal presentation of each abstract data structure.

(ii) Formal specification of abstract data structures (support, operations).

(iii) Informal presentation of each concrete implementation.

(iv) Formal specification of concrete data structures (support, abstraction function, operations).

(v) Calculation of the functional representation of each operation.

The response of the teaching team is very positive. Not one contributor wishes to go back to the previous situation. The students’ point of view will be presented at the conference.

References


