WHAT IS AN ABSTRACT DATATYPE?

Jan BERGSTRA
Institute of Applied Mathematics and Computer Science, University of Leiden, Wassenaarseweg 80, Leiden, The Netherlands

Received September 1977

Abstract datatype, implementing process, relative implementability

1. Introduction

Consider the integers Z with constant 0 and operations S (successor) and P (predecessor) and with relation ZERO(x) ($x = 0$).

We conceive $Z = \langle Z, S, P, ZERO, 0\rangle$ as a datatype. An implementation of this datatype allows us to execute the following instructions.

NEW-ZERO(x) for some $x \in N$ (a set of names), here the result is that x denotes a new integer which currently has value 0.

S(x), for $x \in N$ which has been introduced by a NEW-ZERO(x) instruction.

The effect is that the value of x is increased by one.

P(x), similar.

ZERO(x) asks for $x = 0$, if so the implementation answers T otherwise F.

It seems common to look at an abstract datatype as an algebraic structure, specified by axioms [3]. This is also our point of view in [2]. The present note aims at a definition from an operational point of view which is as general as possible. Some advantages of this generality are discussed in Section 3. Roughly we specify (define) a datatype by means of the collection of its (correct) implementations. To describe implementations we will introduce processes.

2. Processes

A process P communicates with the outside world W and is permanently influenced by this communication. W poses questions in a language $\mathcal{L}_Q$ and gives answers in a language $\mathcal{L}_A$. Mathematically P is a name which denotes at any stage a process extension ext(P). A *process extension* is a function of type

$$L_Q^* \to L_A = \mathrm{PE}(L_Q, L_A),$$

ext(P)($\langle l_1^q, \ldots, l_n^q\rangle$) = l^a has the following meaning: If ext(P) is the extension of P (at a certain stage) then the answer of P after successive questions $l_1^q, \ldots, l_n^q$ is l^a. Answering a question l^q results in redefining ext(P) as follows:

$$\mathrm{ext(P)} := \lambda\sigma_{\in \mathcal{L}_Q^*}\, \mathrm{ext(P)}(\langle l^q\rangle * \sigma).$$

We denote the initial extension of P by P_ϵ and the extension after sequence σ of questions by P_σ.

In the case of the example in Section 1 we can take:

$$\mathcal{L}_Q = \{\text{NEW-ZERO}(x), S(x), P(x), \text{ZERO}(x) | x \in N\}$$

and

$$\mathcal{L}_A = \{T, F, \text{BLANK}\}.$$

An implementation of Z is now a process which answers T or F on questions of the form ZERO(x) and BLANK on other questions, and which gives correct answers w.r.t. the natural interpretation of the questions (instructions).

Definition 1. An abstract datatype D is a triple

$$D = \langle \mathcal{L}_Q, \mathcal{L}_A, I\rangle$$

where $\mathcal{L}_Q$ and $\mathcal{L}_A$ are question and answer languages and I is a collection of (implementing) processes.

Remarks. (i) This definition is very general and includes datatypes for which no mathematical semantics can be found.

(ii) The reason to include a collection of processes rather than a single one lies in the possibility to include nondeterministic datatypes as well. (See Section 3 for a comment on this).

(iii) It may be useful to have some apriori restrictions on the languages $\mathcal{L}_Q$ and $\mathcal{L}_A$. For example if we concentrate on datatypes which have a semantics in terms of a mathematical structure with operations and relations we may restrict $\mathcal{L}_A$ to {T, F, BLANK} and $\mathcal{L}_Q$ to operation (Op(N_1, ..., N_k)) and introduction (NEW-VAL(x)) instructions.

Definition 2. Let $D = \langle \mathcal{L}_Q, \mathcal{L}_A, I \rangle$, $D' = \langle \mathcal{L}'_Q, \mathcal{L}'_A, I' \rangle$ be two datatypes then we say that D is implementable relative to D' if there exists a (partial) recursive transformation T on processes such that for all $P' \in I'$ $T(P) \in I$.

Remark. In [1] we describe a recursion theory for processes. We introduce computations having processes and natural numbers as arguments (suppose that $\mathcal{L}_Q$ and $\mathcal{L}_A$ are coded in N). Then we define machines which answer by such a computation on questions and compute a new state from which the next question will be answered. This leads to a notion of relative recursion and of partial recursive mapping on processes. It may be useful to consider restricted forms of relative recursion in Definition 2 especially if I contains a (general) recursive process.

3. Comments

Data protection

An important reason for concentrating on abstract datatypes is to prevent a user to use the relevant information, or something in connection with representation, in a non intended way. From the viewpoint of recursion theory information is, say a function f: $N \to N$. It can be coded in a datatype with $\mathcal{L}_Q = \mathcal{L}_A = N$ and $I = \{P_f\}$ where P_f answers always $f(x)$ on question x. Recursion theory suggests one way to protect information of f. Namely let the user only use f' for some function f' recursive in f. The present context, however, suggests a user of P_f to be provided with a process P' which is recursive in P_f. It is not difficult to show that the second method is essentially more general in the way that P' may have no equivalent of the form f' with $f' \leqslant f$.

Nondeterministic datatypes (NDDT)

Finally we give some motivation for considering nondeterministic datatypes. Suppose we work with finite subsets of a set E of elementary objects with as operations (relations)

union with singleton: $x \cup \{e\}$,
deletion of a single element: $x/\{e\}$,
predicate for the empty set: EMPTY(x),
selection of element: $x \to e \in x$ if x not empty.

It is quite natural to assume that the selection is nondeterministic. Namely if the sets are represented by unordered lists selection can be implemented by choosing the first, if any, element of the list. Given languages $\mathcal{L}_Q$, $\mathcal{L}_A$ for this datatype it is quite clear which are the correct implementations. Here we have an example of an NDDT. It can be used to implement more complicated operations like union and intersection, then obtaining again a deterministic datatype.

It seems that this approach is especially meaningful in the case of an algebraic datatype which has an underlying domain of the form $A/\equiv$ where A is some inductively defined set and $\equiv$ is an equivalence relation. To obtain a "structured" implementation of the operations on $A/\equiv$ it can be useful to reflect the natural recursion on A in a nondeterministic recursion on $A/\equiv$, thus obtaining an NDDT. This NDDT is used for further implementation of complex operations.

References

[1] J.A. Bergstra, Recursion theory on processes, Leiden, techn. report July 1977.
[2] J.A. Bergstra, A. Ollongren and Th.P. van der Weide, An axiomatisation of the rational dataobjects, Proceedings FCT, Poland (September 1977).
[3] J. v. Guttag, The specification and application to programming of abstract datatypes (Thesis Toronto 1975).