AB42.4.8

REMARKS ON ABSTRACTO*

Leo Geurts
Lambert Meertens
Mathematisch Centrum, Amsterdam

1. ABSTRACTO LIVES

If an author wants to describe an algorithm, he has to choose a vehicle to express himself. The "traditional" way is to give a description in some natural language, such as English. This vehicle has some obvious drawbacks. The most striking one is that of the sloppyness of natural languages. Hill [1] gives a convincing (and hilarious) exposition of ambiguities in ordinary English, quoting many examples from actual texts for instructional or similar purposes. The problem is often not so much that of syntactical ambiguities ("You would not recognise little Johnny now. He has grown another foot.") as that of unintended possible interpretations ("How many times can you take 6 away from a million? [...] I can do this as many times as you like."). A precise and unambiguous description may require lengthy and repetitious phrases. The more precise the description, the more difficult it is to understand for many, if not most, people. Another drawback of natural languages is the inadequacy of referencing or grouping methods (the latter for lack of non-parenthetical parentheses). This tends to give rise to GOTO-like instructions.

With the advent of modern computing automata, programming languages have been invented to communicate algorithms to these computers. Programming languages are almost by definition precise and unambiguous. Nevertheless, they do not provide an ideal vehicle for presenting algorithms to human beings. The reason for this is that programming languages require the specification of many details which are relevant for the computing equipment but not for the algorithm proper. The primitives of the programming language are on a much lower level than those of the algorithm itself.

The evolution of high-level programming languages is one in which the level of the available primitives increases towards the abstractions that human beings use when thinking about algorithms. Still, the gap is very, very large. Unfortunately, recent progress is not yet reflected in any major, generally known programming language.

However, high-level programming languages have had a direct influence on the presentation of algorithms in the literature. Many an author now employs a kind of pidgin ALGOL to express himself. The pidgin characteristics are all present: (a) the language is primarily a contact language, used between persons who do not speak each other's language; although each "speaker" may have his own variant, there is mutual understandability; (b) there is a limited vocabulary, and the syntax is stripped down to the bare necessities, with elimination of the grammatical subtleties that can only be mastered by a regular user; (c) the language is not frozen but permits adaptation to various universes of discourse. The main advantages to the author (and his audience) are that there is no need for a preliminary and boring exposition of the algorithmic notation, that mathematical notions and notations may freely be employed, and that the resulting description is sufficiently precise to convey the algorithm

---

* This paper is registered at the Mathematical Centre as IW 97.

without the deleterious burden of irrelevant detail.

This pidgin ALGOL is a language. It is not really a programming, nor a natural language, but it has characteristics from both. It is not steady, but evolving. How it will evolve we cannot know. But as any man-made thing, its evolution can be influenced by our conscious effort. This language on-its-way may be dubbed Abstracto. (The name "Abstracto" arose from a misunderstanding. The first author, teaching a course in programming, remarked that he would first present an algorithm "in abstracto" (Dutch for "in the abstract") before developing it in ALGOL 60. At the end of the class, a student expressed his desire to learn more about this Abstracto programming language.)

Abstracto '77 is a clumsy language, like any pidgin. Only when a pidgin language becomes a mother tongue, which is not picked up in casual contacts but is the primary language one learns and uses, can it become the versatile tool that allows the expression of complicated thoughts in a natural way.

There are at least two reasons for programming-linguists to study Abstracto. The first is that we may hope to speed up the evolution of Abstracto, by proposing and using suitable notations for important concepts, either derived from existing programming languages, or newly coined. (An excellent example are Dijkstra's guarded commands.) The second is that Abstracto may show us how to design better programming languages.

2. THE LANGUAGE OF MATHEMATICS

It is possible to draw a parallel with the language of mathematics. Only a few centuries ago, the simplest algebraic equation could only be described in an unbelievably clumsy way. This very clumsiness stood directly in the way of mathematical progress.

Take, for example, Cardan's description of the solution of the cubic equation $x^3 + px = q$, as published in his Ars Magna (1545). The following translation from Latin is as literal as possible, with some explanations between square brackets that would have been obvious to the mathematically educated sixteenth-century reader:

> RULE
> Bring [Raise] the third part of the number [coefficient] of things [the unknown] [i.e., p] to the cube, to which you add the square of half the number [coefficient] of the equation [i.e., q], & take the root of the whole [sum], namely the square one, and this you will [must] sow [copy], and to one [copy] you join [add] the half of the number [coefficient] which [half] you have just brought in [multiplied by] itself, from another [copy] you diminish [subtract] the same half, and you will have the Binomium with its Apotome [respectively], next, when the cube root of the Apotome is taken away [subtracted] from the cube root of its Binomium, the remainder that is left from this, is the estimation [determined value] of the thing [unknown].

Nowadays, there is a large basic arsenal of mathematical notions and corresponding notations that may be freely used without further explanation. Each specialism has, in addition, its own notations. Nevertheless, each author is free to introduce new notations as the circumstances require.

Which notations survive in the struggle for life is determined by several factors, of which the ease of manipulating expressions is probably

the foremost one. Still, several notations may coexist, each with its own advantages and disadvantages (like Newton's versus Leibnitz's notation for derivatives). Generally, mathematicians do not bother too much about syntactical ambiguity and do not even stoop down to indicate operator priorities, as long as the intended meaning is conveyed to the gentle reader. (How different from that adversary, the automaton!)

The wildgrowth of notations in new fields can, under circumstances, be effected beneficially by a more or less authoritative body (possibly one person). Donald Knuth's proposal [2] for, among others, the use of a Greek letter theta to denote the class of functions of some order, constitutes an intervention for lack of an established notation. Such interventions are not to be confused with standardization efforts! Only in a frozen field is it possible to standardize, else we have a case of death by premature exposure to frost (hopefully of the standard).

It is difficult to characterize what constitutes good notational practice. Not only is "elegant" vague, but where notation is concerned, it is just a synonym for "good to use". Some criteria are: conciseness, similarity to notations for similar concepts, and relative independence of context. There are, of course, enough dubious notations, such as lim f(x) = a, where the equality sign has a subtly different meaning. (An extremely bad case in ALGOL 60 is the switch declaration SWITCH s := l1, 12, 13.)

3. IN SEARCH OF ABSTRACTO 84

We expect that the introduction of better notations will prove as important for the development of "algorithmics", as it has been - and still is - for mathematics. One must, of course, first identify the concepts before a notation can be developed. It seems unlikely that progress will come from selecting mind-blowing concepts, if only because it is hard enough to think about algorithms without having one's mind blown. If the parallel with mathematics is not deceptive, the important point is the manipulation of "algorithmic expressions". From a paper by Bird [3], describing a new technique of program transformation, we quote: "The manipulations described in the present paper mirror very closely the style of derivation of mathematical formulas [...] As the length of the derivations testify, we still lack a convenient shorthand with which to describe programs, but this will come with a deeper understanding about the right sequencing mechanisms."

At first sight it may seem attractive to view an algorithm as a (constructive) solution satisfying a correctness formula

$$\{p\}\ X\ \{q\}.$$

One can develop a notation, like Schwarz's generic command $p \Rightarrow q$ [4], for a solution (or the set of solutions) of the correctness formula. There must be some constraint on the variables that may be altered by the algorithm, since it is hardly helpful to know that

$$x = x_0 \wedge y = y_0 \Rightarrow x = GCD(x_0, y_0)$$

is solved by

$$x := x_0 := y_0 := 3.$$

If v stands for the alterable variables, and we write q[v := e] for the result of substituting e for v in q, then p ⇒ q can already be expressed in Abstracto '77 by

$$v := \epsilon \{e : p \supset q[v := e]\},$$

where "ϵ" denotes the (indeterminate) selection operator.

If one interprets p ⇒ q at the same time as a formula expressing the (proved) existence of a solution, some proof rules may be given. For example, we have a proof rule

$$\frac{p \supset q[v := e]}{p \Rightarrow q}$$

(corresponding to the solution v := e), the proof rule

$$\frac{p \Rightarrow q,\ q \Rightarrow r}{p \Rightarrow r}$$

(corresponding to p ⇒ q; q ⇒ r), and the proof rule

$$\frac{p1 \Rightarrow q1,\quad p2 \Rightarrow q2}{p1 \vee p2 \Rightarrow q1 \vee q2}$$

(corresponding to IF p1 → p1 ⇒ q1 ▯ p2 → p2 ⇒ q2 FI). By turning a derivation of p ⇒ q upside down, a solution is constructed. Unfortunately, there is no suitable rule for a solution of the form

DO b → p ∧ b ⇒ p OD.

(The rule

$$\frac{p \wedge b \Rightarrow p}{p \Rightarrow p \wedge \neg b}$$

does not express termination and allows the derivation of p ⇒ p ∧ ¬ b for arbitrary p and b.)

There are several other courses one may follow to search for more constructive elements of Abstracto. One is similar to the way high-level programming language elements originate: consider existing (Abstracto) programs, and find similar "code sequences" that appear to be the expression of the same more abstract concept. Just like

```
L1: IF NOT condition GOTO L2
    perform something
    GOTO L1
L2:
```

may be expressed more clearly by

```
DO condition → perform something OD,
```

one might wish to express

```
vopt := ∞;
FOR e ∈ s
DO IF ok1 (e)
   THEN IF v < vopt
        THEN eopt, vopt := e, v
        FI WHERE v = f1 (e)
   ELIF ok2 (e)
   THEN IF v < vopt
        THEN eopt, vopt := e, v
        FI WHERE v = f2 (e)
   FI
OD
```

as

```
eopt, vopt := FOR e ∈ s
              OPT ok1 (e) → f1 (e)
                □ ok2 (e) → f2 (e)
              TPO.
```

(This is not a serious proposal, but neither is it a mere joke.)

Instead of this bottom-up approach a more analytical consideration of the human way of thinking about algorithms may prove, in the long run, more fruitful. In contrast to the process of developing a program, given an algorithm, it appears that little is known about this subject. Descriptions of algorithms in natural languages do not provide much insight, presumably because of the poor expressiveness for algorithmic notions. (One tendency, however, is very noticeable, and is maybe an indication that is worth following up: what might be called the "and-so-on" descriptions, and the "afterthoughts". We surmise that this reflects the emergence of algorithms as the jump to the limit of a sequence of approximations.)

Perhaps the best approach is the following. Suppose a textbook has to be written for an advanced course in algorithmics. Which vehicle should be chosen to express the algorithms? Clearly, one has the freedom to construct a new language, not only without the restraint of efficiency considerations, but without any considerations of implementability whatsoever.

The following is an attempt to indicate some desiderata for Abstracto 84.

Orthogonality is a must. For a lingua franca without frozen and formal description, exceptions are out of the question.

Abstracto 84 has an ALGOL flavor, but is certainly not committed to the control structures or any other particular construct of any ALGOL whatsoever.

With the exception of truth values, Abstracto 84 has no predefined types, but only ways to construct new types from "application oriented" types. Operations on objects are outside the realm of Abstracto 84 proper, except such operations as have a generic meaning for a class of types constructed by means provided by Abstracto 84 (cf. Wilkes [5]).

Although there are variables for objects of any type, these variables

are not considered as new objects. There are no pointer values (except when introduced for a specific application).

Similarly, procedures are not considered as objects which may be assigned etcetera.

Conditions may contain defining identifiers which are also bound in the controlled clause selected if the condition succeeds.

4. GLIMPSES OF ABSTRACTO 84

Due to our near-sightedness, it is difficult to discern more than some outlines of Abstracto 84. Of some prominent features a glimpse may now and then be caught. It should go without saying that all mathematical notation remains welcome to Abstracto.

First of all, it is clearly settled, even in this early stage, that Abstracto is rich in "iterators" (operators or other constructs that operate on generators in an Alphard-like sense). For example, one may write a condition

$$\exists\ e \in s\colon p(e),$$

and if this succeeds, then in the scope of the selected clause, if any, e accesses some element from s satisfying the predicate p. Such constructions may provide a clear and concise description that is quite close to the algorithm originally conceived. Also, if it is immaterial for the algorithm in which order elements are selected, it is important that this be expressed.

The control structures of Abstracto 84 seem to be centered around guarded command sets (Dijkstra [6]) of the form:

$$C_1 \to S_1 \ \square\ C_2 \to S_2 \ \square \ \ldots \ \square\ C_n \to S_n.$$

The basic meaning of such a form is: if at least one of the C_i holds (where the evaluation of a condition is supposed to have no side effects), then some corresponding S_i is selected (but not yet evaluated). In the terminology of the ALGOL 68 Report, a scene is selected, composed from that S_i and an environ whose most recent locale may have been added because of the declarative form of C_i.

The meaning of IF ... FI and DO ... OD may now be defined easily. It appears, however, that in Abstracto 84 several other control structures may be defined with the guarded commands at their cores, as suggested by the FOR ... OPT ... TPO construct in the previous section. The basic simplicity of the concept, in conjunction with its indeterminacy, should warrant ease of manipulation.

Many types, specifically those that can be treated satisfactorily by so-called axiomatic/algebraic specifications, can be defined in the way exemplified below:

```
tree ::= nil | atom (val: item) | pair (left, right: tree).
```

(We write "::=" to stress the similarity with BNF, although this "syntax" of objects is more abstract than usual, since the nodes in the "parse tree" of an object are labelled; in the example, "nil", "atom" and "pair" are node labels.) This notation is similar to Hoare's notation for recursive

data structures [7]; it carries no other information than is relevant from an abstract algorithmic point of view. There are three nice things about this way of defining types. In the first place, it is easy to derive in a straightforward way "axiomatic" specifications in the style of Guttag [8], but the notation is much more compact. (For the above example, we would obtain nine lines for the discernible functions and eighteen for the axioms.) Secondly, this way of defining offers a unification of three well-known concepts:

records, as in

```
complex ::= pair (re, im: real);
```

(disjoint) unions, as in

```
arithmetical ::= i (val: int) | r (val: real);
```

PASCAL scalars, as in

```
color ::= red | blue | green.
```

Finally, it is easy to instruct a compiler to handle such definitions. The only drawback is the inefficiency, reason why such definitions are maybe Abstracto rather than Concreto.

Objects of a thus defined type can now be subjected to a "conformity condition", as in

```
DO t FITS
   pair (t1, t2) → t := t2
OD.
```

In this example, if the condition succeeds, t2 accesses the tree t.right.

5. A POSSIBLE PITFALL

Unless we are very mistaken, program development by successive "program transformations", i.e., a sequence of manipulations on expressions which represent algorithms, has a promising future. Each transformation rule is a theorem. To us, computer maniacs, the perspective is tempting to create a data base of transformations to be applied mechanically. Since the applicability of each transformation is also checked mechanically, we have done away with all bugs (except for those in the original, pure, algorithm, possibly a problem specification). What vista! Of course, we must invent for our Abstracto language some syntactic notions to allow expression of the applicability of transformations.

The last sentence should make it clear already that the pursuit of this Utopian concept - unless one contents oneself with trivial transformations that might as well be applied directly by a compiler - spoils the simplicity of Abstracto. Worse yet, the concept wholly ignores the fact that in mathematics for none but the simplest theorems the applicability may be checked by "syntactical" means. If computers would have dated back to the inception of modern mathematical notation and only mechanizable transformations would have been studied, the so-called special products would, presumably, still be among the high-lights of mathematical knowledge.

To quote once more Bird [3]: "we did not start out, as no mathematician

ever does, with the preconception that such derivations should be described with a view to immediate mechanization; such a view would severely limit the many ways in which an algorithm can be simplified and polished."

REFERENCES

[1] Hill, I.D., Wouldn't it be nice if we could write computer programs in ordinary English - or would it?, Computer Bull. 12 (1972) 306-312.

[2] Knuth, D.E., Big omicron and big omega and big theta, SIGACT News 8 (1976) 2, 18-24.

[3] Bird, R.S., Improving programs by the introduction of recursion, Comm. ACM 20 (1977) 856-863.

[4] Schwarz, J., Generic commands - a tool for partial correctness formalisms, Computer J. 20 (1977) 151-155.

[5] Wilkes, M.V., The outer and inner syntax of a programming language, Computer J. 11 (1968) 260-263.

[6] Dijkstra, E.W., Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM 18 (1975) 453-457.

[7] Hoare, C.A.R., Recursive data structures, Stanford University Report CS-73-400 (1973).

[8] Guttag, J.V., Abstract data types and the development of data structures, Comm. ACM 20 (1977) 396-404.