ABSTRACTO 84: THE NEXT GENERATION

Lambert Meertens
Mathematical Centre
Amsterdam

Abstract. Programming languages are not an ideal vehicle for expressing algorithms. This paper sketches how a language Abstracto might be developed for "algorithmic expressions" that may be manipulated by the rules of "algorithmics", quite similar to the manipulation of mathematical expressions in mathematics. Two examples are given of "abstract" algorithmic expressions that are not executable in the ordinary sense, but may be used in the derivation of programs. It appears that the notion of "refinement" may be replaced by a weaker notion for abstract algorithmic expressions, corresponding also to a weaker notion of "weakest precondition".

1. THE ABSTRACTO PROJECT

Since December 1977 IFIP Working Group 2.1 has been working on the investigation of "the properties, feasibility and usefulness of a language helping the specification and construction of good algorithms". If this description seems vague (it is so on purpose), it nevertheless describes "something" that is almost tangible by its conspicuous absence from the programmer's tool kit.

A programmer who is writing a program is in fact encoding an algorithm in a language for some machine. This need not be a piece of hardware; it can be "the" abstract machine for FORTRAN or some other high-level language. The development of an algorithm down to the machine level takes many steps, some of which require ingenuity, but the larger part of which consists of clerical manipulations and book-keeping. This is partly due to the (not always unjustified) wish of writing an efficient program, and partly to the fact that even the highest-level languages require the specification of details that are relevant to the machinery, but not to the algorithm proper.

It would be good practice if the programmer would first write down the algorithm before starting to code it as a program. But now, in what way? Some "algorithmic" language is needed. The available languages, however, are programming languages. (Hill[5] shows convincingly how unsuited natural language is for this purpose.) So we are back were we started: to write an algorithm in a programming language is to write a program.

In a nutshell, the aim of the Abstracto project is to fill the gap by designing a language specifically for the purpose of describing algorithms. The language should be a suitable vehicle for applying established programming techniques, and thereby also for teaching such techniques, without danger of having to explain ideosyncracies.

The Abstracto project is still in its early phase. There is not even an approximation of consensus about the basics of Abstracto. In this paper some ideas are presented; it should be stressed that these represent solely my position and may not be taken for opinions of WG 2.1. Although some logical formalism is used in this paper, the reader should be warned that this is only done for the purpose of conveying a meaning; nothing is alleged to be "proved" here.

2. ABSTRACTO AS A PIDGIN

When people who do not speak a common language establish a regular contact and want to communicate, an interesting phenomenon happens: they develop a "pidgin" language, clumsy but effective. A similar phenomenon has happened in Computer Science literature: a kind of pidgin ALGOL has developed there, from the need of authors to address a broad audience without having to explain over and over the meaning of all notations employed. This pidgin ALGOL is a language, although it is not frozen, let alone formalized. In fact, it has some of the characteristics from natural languages.

A major similarity is the property that this language is gradually evolving, to meet the needs in communicating algorithms. One may (and I do) take the position, thus mitigating the grimness of the situation sketched in the previous section, that pidgin ALGOL covers to some extent the need for an algorithmic language. Moreover, the "natural" course of evolution will be to tune the language to the requirements of developing programming methodology. However, we are still far away from what could be achieved even today. As long as we are faced with the situation that the language has to be mastered by picking it up from casual contacts, it will of necessity drag along trails that have been beaten years before.

Viewed in this perspective, the Abstracto effort is aimed at speeding up evolution by proposing and using suitable notations for important algorithmic concepts. Of course, it will be possible (and maybe desirable) to take a snapshot of Abstracto at regular intervals, to clean up the picture and to present it as, say, Abstracto 84. But this will not stop Abstracto from evolving on.

The obvious advantage of freezing an Abstracto X is the possibility of referring to a "standard" when publishing an algorithm. Moreover, when a language is formalized, it also becomes possible to formalize proof rules and to prove their consistency and completeness. These are not, however, the main reasons why I feel the effort of freezing a version of Abstracto at some future time may prove worth the trouble. It seems much more important to me that this forces one to clarify issues that still appear murky, thereby deepening the understanding of what is going on. Also, it may show us how to design better programming languages.

3. ABSTRACTO AND TRANSFORMATIONAL PROGRAMMING

Unlike many fads in Computer Science, the relatively recent technique of "transformational programming" appears to be quite promising. One should of course not make the mistake to expect that it opens up a royal road to program construction; no technique ever will. But the basic idea is quite simple and sound, its value has been demonstrated on diverse, sometimes even not trivial, examples, and it provides a framework for expressing an expanding body of knowledge about programming and for developing new programming techniques (or applying "old" programming techniques known under the collective title of Structured Programming). In essence, the method of transformational programming consists of (a) writing an algorithm, as pure and simple as possible, to meet a given specification as to correctness, and (b) next successively transforming the algorithm, by relatively simple correctness-preserving transformations, to meet other requirements, such as those stemming from efficiency considerations.

Transformations may be global, replacing the whole program under development by a new text, but the typical transformation is local, effecting only a small part. Ideally, the algorithm at the top should be identical with the correctness specification, but we do not know in general how to go down from that level by something in the spirit of a transformation.

Well-known transformations are stepwise refinement and recursion removal. It may well happen, however, that at some stage of development recursion introduction (Bird[2]) is in order to prepare for a more advantageous step.

The nature of transformational programming is quite aptly described by Bird: "The manipulations [...] mirror very closely the style of derivation of mathematical formulas". He also remarks: "As the length of the derivations testify, we still lack a convenient shorthand with which to describe programs".

It is here that Abstracto should step in. It is important to realize that the objects one manipulates upon are not the algorithms themselves, but are expressions: algorithmic expressions. In fact, for most steps it is impossible to maintain that there occurs a change in the algorithm (unless one refuses to admit the existence of "the" Euclidean algorithm, or "the" sieve of Eratosthenes). For these algorithmic expressions, we need notations. None of the existing programming languages has been designed with a design objective as ease of manipulation. On the contrary; if one would not know better, one would in many cases be tempted to believe they were designed on purpose to be transformation resistent: the semantic peculiarities often make it devilishly hard to verify that a particular step is applicable. Moreover, the verbosity of existing notations makes it aggravating to write down the derivations and makes it hard to keep track of what is happening. It is to be expected that the introduction of better notations will prove as important for the development of "algorithmics" as it has been for mathematics.

4. DESIGNING ABSTRACTO 84

To make Abstracto catch up with the state of the art, it seems wise to go through the motions of designing a language from scratch. One should have the freedom of ignoring established but cumbersome notations and conventions.

There is, however, a much more important degree of freedom that should be explored and exploited: unlike any programming language, Abstracto is exempt from the requirement that its texts should be understandable to an automaton, let alone that it should be possible to coerce it to execute the process described by an algorithmic expression from Abstracto merely by proceeding to feed it the source text. Rather than trying to extend the machine to higher levels of abstraction by erecting scaffolds from the hardware, we can start in the blue sky and go down from there. It is nice, of course, if we can reach solid ground, but this is not a prerequisite.

Nevertheless, it should be possible to write more or less conventional programs in Abstracto also. This means that a piece of program like

```
z:=1 ; x:=2 ; z:=z·x
```

is fine. This leads to the question of types and data structures in Abstracto 84.

It is desirable that the programmer can use objects of any type conceivable. Rather than creating some heavy mechanism for adding user-defined types to the language, it is far easier to allow the definition of any new type, including the semantics of the operations characterizing the type, as preliminaries to the algorithm. If the type under consideration is well established (e.g., integers), there will often be no need to explain beforehand the various operations used. So Abstracto 84 has no predefined types (with the exception of truth values, and maybe other types linked up with control structures). Operations on objects fall outside the realm of Abstracto 84 proper. Apart from these "application oriented" types, there are types constructed from existing types (e.g., sets). Abstracto 84 may suggest some unification in the notations for some classes of such types; the question whether this "belongs" to Abstracto 84 or not is not particularly relevant.

As a consequence, all of established mathematical notation is welcome in an Abstracto 84 program. The syntax of Abstracto 84 will not attempt to define what may appear on the right-hand side of an assignment. Remember that this is acceptable, since Abstracto 84 texts are not required to be interpretable by machine.

The same liberal attitude can be taken for the whole of Abstracto 84. The rule would be: any notation or convention that is sufficiently clear may be used, provided that its meaning, if not self-evident, is explained in the preliminaries. The effort in designing Abstracto 84 should go in establishing which new, or not yet commonly accepted, notations are sufficiently important to exempt them from the requirement of preliminary explanation for use in Abstracto 84 expressions. When designing a language (especially by committee) it is often quite hard to keep the language from being clogged by a multitude of things, for none of which individually there is a particularly compelling reason to ban it. Thus, the liberal rule may save many tears: cherished notations may be used anyway, even if no part of Abstracto 84 proper. In fact, it is my feeling that this rule is essential for the viability of the project. Just consider what would happen to a language Mathematico 84 for mathematical expressions that took a rigid and exclusive attitude as to what was allowed: the inevitable expressive shortcomings would be as many reasons to shun it.

In the sequel, "Abstracto 84" will refer to Abstracto 84 proper, the core of an extensible language - where the extension mechanism is not part of the language. An "algorithmic expression" (or, for short, "expression") is a piece of text written in the, possibly extended, language. It may be helpful to think of expressions as "statements", since they describe a process to be executed. Something like "z·x", conventionally called an expression, will be called a "unit" in the sequel of this paper.

It is well known that many mathematical notations are potentially ambiguous. In practice, this is not harmful: if a given mathematical expression turns out ambiguous, parentheses will do. Ambiguity here does not mean that there is more than one parse, but that there exist two or more plausible parses with different meanings. Similarly, one should not worry too much about potential ambiguities for algorithmic expressions. If priority conventions are established, their purpose is to save the writing of parentheses, not to compel insertion where the intended meaning is already clear enough. So the syntax of Abstracto 84 is abstract rather than concrete.

If S_1 and S_2 are expressions, than so is $S_1;S_2$. Expressed in operational semantics, the meaning is sequential execution. By the above rule, since $(S_1;S_2);S_3$ is clearly equivalent to $S_1;(S_2;S_3)$, we may write $S_1;S_2;S_3$, and so on. Other control mechanisms in Abstracto 84 are given by the guarded command constructs of Dijkstra[4]. However, for the ease of manipulation, we write "..." and "*(...)" rather than "IF ... FI" and "DO ... OD". So we have

$$b_1 \rightarrow S_1 \ [] \ \dots \ [] \ b_n \rightarrow S_n,$$

meaning (operationally) that some i is selected such that the guard b_i holds, whereupon S_i is executed. If no such i exists, the meaning is undefined (the same as that of an infinite loop). The meaning of the loop expression

$$*(b \rightarrow S)$$

is the same as that of

$$b \rightarrow S;*(b \rightarrow S) \ [] \ \neg b \rightarrow \underline{\text{skip}}.$$

Although it is envisaged that more control structures may be needed in Abstracto 84, it is helpful if their meaning is defined in terms of simpler expressions, so that an existing body of transformations becomes automatically available. For expressing concurrency (parallel execution), however, this is impossible with the concepts given so far. A possible notation is not hard to devise; the problem is to select a proper synchronization mechanism.

A basic type of algorithmic expression is the assignment expression. Following Dijkstra again, Abstracto 84 allows parallel assignment expressions such as

$$x,y := -y,x.$$

This is quite natural, since the assignment expression might result from transforming an assignment expression

$$z := iz$$

using $z = x + iy$.

5. ABSTRACT ALGORITHMIC EXPRESSIONS AND REFINEMENT

So far we have seen nothing exciting. If it is claimed that Abstracto 84 is of a higher level than SETL, say, this is not because it usurps by extension the notations of SETL. The reason is, rather, that expressions in Abstracto 84 need not be executable in the usual sense.

Let us consider for a moment what we mean by "executable". It is the property of an expression that makes it possible to have it executed by a computer. Now, if we have a mathematical expression like "21/7", we know that its meaning is: a number x such that 7x = 21. So we can view "21/7" as a concise problem specification: find a number x such that 7x = 21. There exists a well-known algorithm to solve this type of problem. In many computers it is implemented in the hardware. High-level programming languages allow for notations to invoke that algorithm. The usual notation for that is "21/7". This is a concise specification for the solution to the above problem: divide 21 by 7; the result will be the required number. Obviously, it is a matter of viewpoint whether "21/7" specifies a problem or a solution. We have almost forgotten that it may be considered as a problem, although at some time in our lives we have certainly done so. In general, a problem specification for a problem that falls in a class where there exist known algorithms to solve the problem, may be considered simultaneously as a solution specification. In mathematical practise, the distinction between the two is very vague, a matter of taste. This vagueness is in fact beneficial.

Similarly, we need the same vagueness in Abstracto 84. It may happen that a given expression looks so suspiciously like a program that we may successfully feed it to a compiler and have it run. Now consider the subset EA (Executable Abstracto) of expressions for which this works. It is claimed that EA is a fuzzy set. As time proceeds, more and more algorithms may be incorporated in the semantics of programming languages to cover parts of Abstracto that were, until then, deemed "unexecutable". By that act, EA grows. Thus, the experience gained by using Abstracto may serve as a guideline for the development of programming languages.

Abstracto 84 should provide expressive capabilities for a broad range, covering very clearly problem specifications on one end, and very clearly solution specifications on the other. The notion of "algorithmic expression" encompasses the whole range. By applying the arts and techniques of Algorithmics, these expressions may be manipulated. (To my taste the term "algorithmics", by analogy to "mathematics", is far better than the usual "Transformational Programming". After all, mathematics is more than "Transformational Arithmetic", even though much mathematical effort is aimed at evaluating expressions). The field of algorithmics is still underdeveloped, of course; mathematics could only take its flight when suitable notations came to be developed.

It may prove that the most important part of Abstracto 84 is the in-between range: no longer clearly a problem, but not yet clearly a solution. This is the part where notations are most lacking.

Even though the notion of "executability" is fuzzy, it is useful to have some terminology to indicate the concept. Since I prefer a more neutral terminology, I propose to call an expression "concrete" if it is free of "unexecutable" notations, and "abstract" otherwise. The task of a programmer is to derive concrete expressions from abstract ones.

It should be stressed that "abstract" does not imply "vague". An abstract expression may have a very precise meaning. But this meaning need not be defined in terms of: first do this, next that, and so on.

In order to search for powerful abstract expressions, we must have an idea in what way we want to use them. In mathematics, the central notion is that of equality. In algorithmics, however, another, asymmetric relationship plays a central role: that of refinement. Speaking informally, an expression S is refined by another expression S′ if any concrete realization of S′ is also a concrete realization of S. Note that this does not exclude the possibility that S is concrete and S′ is abstract.

It is necessary to define the meaning of refinement more formally. For p and q assertions, and S an expression, let the correctness formula {p}S{q} stand for: a concrete realization of S, executed with precondition p, will terminate and result in the postcondition q. S is then refined by S′ if

$$\text{for all p and q, if } \{p\}S'\{q\}, \text{ then } \{p\}S\{q\}.$$

This definition is, however, circular, since a concrete realization of S is a concrete expression C such that S is refined by C. We need an independent characterization of the semantics of abstract expressions. From the various, more or less equivalent, methods for defining semantics, that of weakest preconditions seems quite convenient, since it allows in a natural way to express the indeterminacy of the meaning of abstract expressions. Let wp(S,q) stand for the weakest precondition of S ensuring termination with q. Then $S \leq S'$ means:

$$\text{for all q, } wp(S,q) \text{ implies } wp(S',q).$$

This notion of refinement is identical to that in the work of Back[1], which provides a rigorous mathematical foundation. It is obvious that the relationship is reflexive and transitive:

$$S \leq S;$$
$$\text{if } S \leq S' \text{ and } S' \leq S'', \text{ then } S \leq S''.$$

A very important property is the following. Let f(S) be an algorithmic expression, containing S as a component expression. Then we have:

$$\text{if } S \leq S', \text{ then } f(S) \leq f(S').$$

(This property crucially depends on the way the meaning of expressions is defined in terms of the meanings of their component expressions. A sufficient condition is that the weakest precondition of a composite expression is a positive monotone functional of the weakest preconditions of its components. This is certainly the case for all conventional composition methods.)

It appears that the notion of $\leq$-refinement is stronger than is necessary for abstract expressions. Let C be restricted below to the set of concrete algorithmic expressions. Then we can define $S \leq' S'$ to mean:

$$\text{for all C, if } S' \leq C, \text{ then } S \leq C.$$

This corresponds to the original informal definition. Clearly, if $S \leq S'$, then $S \leq' S'$. The converse need not hold. The important thing to notice, however, is that $S \leq' C$ implies $S \leq C$. In other words, if it is possible to derive a concrete expression for S using $\leq'$-refinement, this is also a correct derivation under $\leq$-refinement. It may be possible that the weaker type of refinement does lead us into blind alleys, but in no way does it lead to incorrect programs.

It is clear that we have lost some "guidance", so a legitimate question is what we have gained. First, one should realize that the original refinement definition is no guarantee against blind alleys in the derivation process. In many cases, one proceeds with a goal in mind, knowing beforehand that this road leads to success. The gain is know that, hopefully, the weaker requirements for the applicability of a refinement step are easier to verify.

It is possible to define a corresponding type of (weaker) weakest preconditions:

$$wp'(S,q) = \bigwedge_{S \leq C} wp(C,q).$$

Then $S \leq' S'$ is equivalent to

$$\text{for all q, } wp'(S,q) \text{ implies } wp'(S',q).$$

Unfortunately, it is not clear how a calculus might be developed for wp′. A practical approach may, however, be found along the following lines. Let cr ("concretely realizable") stand for any predicate over the expressions, chosen such as to satisfy

(i) for all C, cr(C) holds, and
(ii) for all S, wp(S,<u>true</u>) implies cr(S).

Take for wp* any predicate transformer satisfying

$$wp(S,q) = wp^*(S,q) \;\&\; cr(S).$$

Any wp* thus defined satisfies

$$wp(S,q) \text{ implies } wp^*(S,q), \text{ and}$$
$$wp^*(S,q) \text{ implies } wp'(S,q).$$

Now define $S \leq^* S'$ by:

$$\text{for all } q,\ wp^*(S,q) \text{ implies } wp^*(S',q).$$

This $\leq^*$-refinement has again all desirable properties, like reflexivity and transitivity. The freedom in choosing cr is quite large. One extreme is to choose cr(S) identically true for all S; this leads to wp* = wp. The other extreme is to consider termination a prerequisite for concreteness, and to choose cr(S) = wp(S,true). This allows the choice for wp* of the weakest precondition for partial correctness (without termination). In general, given a choice for cr, the range of choice for wp*(S,q) has as extremes at the strong end wp(S,q), and at the weak end $cr(S) \supset wp(S,q)$. The freedom of choice should be used to obtain manageable formulas and rules.

It may appear that cr also has to satisfy

$$\text{if } S \leq^* S' \text{ and } cr(S'), \text{ then } cr(S).$$

In fact, this is not necessary. It is sufficient if we have:

$$\text{if } S \leq^* C, \text{ then } S \leq C.$$

This is indeed the case, as is easily verified.

Dijkstra[4] gives rules for computing wp for compound expressions. It is desirable that the same rules go through for wp*, even if the component expressions are abstract. (However, for the loop expression we need the weaker precondition given by Boom[3], because of the indeterminacy allowed in abstract expressions.) Also, for an expression like $S_1;S_2$, we want $cr(S_1;S_2)$ to hold whenever $cr(S_1)$ and $cr(S_2)$ both hold, and so on. This turns out possible. If we choose

$$cr(S_1;S_2) = cr(S_1) \;\&\; (wp^*(S_1,\underline{true}) \supset wp^*(S_1,cr(S_2))),$$

then it is straightforward to verify that

$$wp^*(S_1;S_2,q) = wp^*(S_1,wp^*(S_2,q))$$

is acceptable as definition. Similarly, one can take

$$cr(b_1 \rightarrow S_1 \;\square\; b_2 \rightarrow S_2) = (b_1 \supset cr(S_1)) \;\&\; (b_2 \supset cr(S_2))$$

as definition and obtain the usual formula for wp*, and so on.

6. EXAMPLES OF ABSTRACT ALGORITHMIC EXPRESSIONS

Before giving two examples of abstract expressions, one notation has to be explained. Let A stand for an algorithmic expression or an assertion, v for a list of variables and u for a list (of the same number of elements) of units. Then the notation

$$A[\![v:=u]\!]$$

stands for A with all free occurrences of v in A replaced by u. A more conventional notation would be A[u/v]. However, if other than simple variables are allowed, the implied substitution should not be performed literally. For example,

$$(a[4] > 0)[\![a[2+2]:=b]\!] = (b > 0).$$

Using this notation, we can express the weakest precondition of assignment expressions quite elegantly:

$$wp(v:=u,q) = q[\![v:=u]\!].$$

Let us start at a high point. Many problems can be described as the task of going from a precondition p to a postcondition q. Thus, we are led to consider problem descriptions of the form

$$\{p\}?\{q\}.$$

There is, however, something essential lacking. This can be seen by looking at the description

$$\{x=x_0,y=y_0\}?\{x=x_0,y=y_0,z=GCD(x,y)\}.$$

This has many presumably unintended solutions, like

$$x,x_0,y,y_0,z := 1,1,1,1,1.$$

There should be a way of indicating the variables that may be changed in the process. This leads to

$$\{p\}v:=?\{q\}.$$

This would do, but it is cumbersome. A better notation for this "problem expression" is

$$v:=[p \Rightarrow q],$$

where v stands for a list of variables. (Warning: $[p \Rightarrow q]$ is not a unit list, so a substitution $[\![v:=[p \Rightarrow q]]\!]$ is meaningless.) In pseudo-operational semantics, the meaning is: set v to some value such that, if initially p held, then now q holds. If p does not hold, any value will do. (One might also not require termination in the latter case; the merits of this variant definition have not been explored sufficiently.)

An example of a problem expression is

$$y:=[x \geq 0 \Rightarrow y^2=x \;\&\; y \geq 0].$$

This could be realized by the concrete expression

$$y:=sqrt(x).$$

If we compute the precondition by transposing this in the formalism of Back[1] and using his rules, we obtain

$$wp(v:=[p \Rightarrow q],r) = (p \supset (\exists v'\!:\; q[\![v:=v']\!])) \;\&\; (\forall v\!:\; q \supset r).$$

Clearly, we may take

$$cr(v:=[p \Rightarrow q]) = p \supset (\exists v'\!:\; q[\![v:=v']\!])$$

and

$$wp^*(v{:=}[p \Rightarrow q], r) = \forall v\colon q \supset r.$$

In fact, $cr(v{:=}[p \Rightarrow q]) = wp(v{:=}[p \Rightarrow q], \underline{true})$.

Some properties of the new type of expression are given by the following list of rules:

(a) If p implies p′ and q′ implies q, then
$v{:=}[p \Rightarrow q] \le^* v{:=}[p' \Rightarrow q']$;
(b) $v{:=}[p \Rightarrow q] \le^* v,v'{:=}[p \Rightarrow q]$, where v′ is a fresh list of variables;
(c) $v{:=}[p \Rightarrow r] \le^* v{:=}[p \Rightarrow q];\ v{:=}[q \Rightarrow r]$;
(d) $v{:=}[p_1 \vee p_2 \Rightarrow q] \le^*$
$p_1 \rightarrow v{:=}[p_1 \Rightarrow q] \ [\!]\ p_2 \rightarrow v{:=}[p_2 \Rightarrow q]$.

Rule (a) corresponds to the usual rule of consequence. Rule (b) allows the introduction of auxiliary variables. As to (c) and (d), these correspond to the usual rules for sequential and conditional composition.

The verification is quite straightforward, but is left as an exercise to the interested reader.

The next abstract expression is less of a problem specification, but still quite abstract. It is the "bound expression"

$$S \mid v{:}p,$$

where v is a list of variables, p is an assertion and S is another algorithmic expression not containing elements of v in the left-hand side position of an assignment expression, problem expression or otherwise (if more expressions with the nature of an assignment are introduced). Informally, its meaning is: execute S where v is chosen such that p is satisfied. An example is given by

$$y{:=}v \mid v\colon x \ge 0 \supset (v^2{=}x \ \&\ v \ge 0).$$

The variables in v are bound to the expression. The semantics is given by computing wp:

$$wp(S \mid v{:}p, q) = (\exists v\colon p) \ \&\ (\forall v\colon p \supset wp(S,q)).$$

We may take

$$cr(S \mid v{:}p) = \exists v\colon p$$

and

$$wp^*(S \mid v{:}p, q) = \forall v\colon p \supset wp(S,q).$$

We can now express some more rules, where $S =^* S'$ stands for $S \le^* S' \ \&\ S' \le^* S$.

(e) $v{:=}[p \Rightarrow q] =^* v{:=}v' \mid v'\colon p \supset q[\![v{:=}v']\!]$, where v′ is a list of fresh variables of the same length as v;
(f) $v{:=}[p \Rightarrow p \& \neg b] =^*$
$*(b \rightarrow v{:=}v' \mid v'\colon$
$p \& b \supset p[\![v{:=}v']\!] \ \&\ O(v') < O(v))$,
where v′ is again a list of fresh variables of the proper length, and O is a mapping from objects of the type of v to the elements of some well-ordered set (e.g., the ordinals), which may be chosen freely;
(g) If p′ implies p, then $S \mid v{:}p \le^* S \mid v,v'{:}p'$, where v′ is a (possibly empty) list of fresh variables;
(h) $S \mid v{:}p \le^* S[\![v{:=}u]\!] \mid v'{:}p[\![v{:=}u]\!]$, where u is a list of units of the proper length and v′ is a list of variables that are either fresh or an element of v, sufficiently large to bind all variables of v that remain present after the step;
(i) $S \mid \varepsilon{:}\underline{true} \le^* S$ (where ε stands for the empty list).

Rules (e) and (f) allow the elimination of problem expressions. If the variant definition hinted at above is adopted, we would only have refinement in one direction. Rule (f) is probably the most powerful one in practice. It corresponds to rules in other proof systems that cover the WHILE loop. The mapping O ensures termination. It can be shown that mapping to the natural numbers (the initial segment of the ordinals) gives the same power, but at the cost of introducing mappings that are sometimes much more complicated than necessary (cf. Boom[3]). In (g) we find another application of the rule of consequence. It might have been combined with (h); for the sake of simplicity, this has not been done. Rule (h) is also quite powerful. By application of this rule one may arrive at (i), where the bound expression is eliminated. One has to go through this rule once for each abstract expression introduced.

Again, the verification is left to the reader. A simple proof of (f) is found by separating partial correctness and termination.

7. AN EXAMPLE

The usefulness of the abstract expressions introduced in the previous section may not be obvious. The test can only be the application to practical examples. In fact, they have been used on a variety of problems of diverse complexity, generally reasonably succesfully. There are two aspects in judging the measure of success. One is how naturally the original problem may be expressed, and one is how easy it is to massage the resulting expression in the intended direction of concreteness. Note, however, that the expressions themselves give no guidance as to what refinement steps are best applied. The freedom of choosing u in rule (h) is beneficial only if one has some expertise in programming (or algorithmics).

No attempt has been made yet to apply the present modest approximation of Abstracto to a large-scale, real-life problem from the top to the bottom. Therefore it is not known how well it will stand up. In theory, any program may be derived that can be written with WHILE loops, but the actual effort may be quite impractical. However, I have some confidence that the situation will not be that bad.

The use of algorithmic expressions will now be demonstrated on a very simple example, treated by Dijkstra[4] and also by Back[1]. The problem is to compute X^Y, where Y is a natural number, without using the exponentiation operator.

This problem can be specified by the abstract expression

$$z{:=}[\underline{true} \Rightarrow z{=}X^Y].$$

Using (b) and (c) of Lemma 1, we refine this to

(S1) $z,x,y := [\underline{true} \Rightarrow z \cdot x^y{=}X^Y]$;
(S2) $z,x,y := [z \cdot x^y{=}X^Y \Rightarrow z{=}X^Y]$.

First we proceed with the easy part, (S1). Where the refinements are given here in two steps, a trained algorithmician would immediately jump to the final version, much like a mathematician is used to do. From (e) we obtain

$$z,x,y := z',x',y' \mid z',x',y': z' \cdot x'^{y'}=X^Y.$$

By using the unit list u = 1,X,Y in (h), this simplifies to

$$z,x,y := 1,X,Y \mid \varepsilon : \underline{\text{true}}.$$

This gives us the final, concrete expression, since now rule (i) is applicable:

$$z,x,y := 1,X,Y.$$

As to (S2), this fits (f) with the assertion $z \cdot x^y=X^Y$ for p and $y \neq 0$ for b. For the mapping 0 we can simply take the identity, since the "goal" is to get y to 0. We thus refine (S2) to

$$*(y\neq 0 \rightarrow z,x,y := z',x',y' \mid z',x',y': z\cdot x^y = X^Y \ \& \ y\neq 0 \supset z'\cdot x'^{y'}=X^Y \ \& \ y'<y).$$

Using (g), this may again be refined to

$$*(y\neq 0 \rightarrow z,x,y := z',x',y' \mid z',x',y',r: z'= z\cdot x^r \ \& \ x'= x\cdot x \ \& \ y=2y'+r \ \& \ (r=0 \vee r=1)).$$

If operations / and % are available, satisfying y = 2(y/2)+(y%2) and (y%2=0 ∨ y%2=1), the use of the unit list u = ZZ,x·x,y/2,y%2 in (d) of Lemma 2, where ZZ is shorthand for $(y\%2=0 \rightarrow z \ \square \ y\%2=1 \rightarrow z\cdot x)$, allows to simplify this to

$$*(y\neq 0 \rightarrow z,x,y := ZZ,x\cdot x,y/2).$$

Here (i) has also been applied. It has now been shown that

$$z:=[\underline{\text{true}} \Rightarrow z=X^Y] \leq z,x,y := 1,X,Y; \ *(y\neq 0 \rightarrow z,x,y := ZZ,x\cdot x,y/2).$$

(Note that we may use "≤" rather than "≤*", since the right-hand side is concrete.)

This proof is admittedly quite lengthy (and boring) for the feat it performs. But this would also be the case for attempts to determine an indefinite integral, say, by following the rules from the calculus book step for step and displaying all intermediate results. A more appropriate proof might read: "this concretization is obtained by keeping $z\cdot x^y=X^Y$ invariant".

Acknowledgements

Many of the ideas presented here, and especially the idea of Abstracto itself, have taken shape in discussions with Leo Geurts. I am indebted to Jaco de Bakker for drawing my attention to the connection between the present work and the work by Back.

REFERENCES

[1] Back, R.-J., On the Correctness of Refinement Steps in Program Development, Report A-1978-4, Department of Computer Science, University of Helsinki, 1978.

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

[3] Boom, H.J., A weaker precondition for loops, Report IW 104/78, Mathematical Centre, Amsterdam, 1978.

[4] Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, 1976.

[5] 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.