Algorithmics

Towards programming as a mathematical activity

Lambert Meertens
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Of the various approaches to program correctness, that of "Transformational Programming" appears to be the most helpful in constructing correct programs. The essence of the method is to start with an obviously correct—but possibly hopelessly inefficient—algorithm, and to improve it by successively applying correctness-preserving transformations. The manipulations involved are akin to those used in mathematics. Two important impediments to this method are the verbosity of algorithmic notations, making the process cumbersome, and the semantic baroqueness of many primitives, making it hard to verify the validity of transformations. Computer Science can profit here from the lessons taught by the history of Mathematics. Another major step, comparable to one made long ago in Mathematics, is not to insist on the "executability" of algorithmic descriptions. This makes it possible to treat initial high-level specifications in the same framework as the final programs. Just as Mathematics evolved from "Transformational Arithmetic", Transformational Programming may come of age as "Algorithmics".

> Mathematical reasoning does play an essential role in all areas of computer science which have developed or are developing from an art to a science. Where such reasoning plays little or no role in an area of computer science, that portion of our discipline is still in its infancy and needs the support of mathematical thinking if it is to mature. RALSTON and SHAW [25]

0. INTRODUCTION

The historical roots of Mathematics and Computing are intertwined. If we ascertain the validity of a more efficient way of doing computations—more generally, of constructing a result—, we are performing mathematics.

Nowadays, we are happy to leave the actual computing to automata. Our task is to prescribe the process, by means of a program. But however great the speed of our automaton, our need for results is greater, and an important part of the Art of Programming is finding efficient computational methods. Whoever thinks now that programming as it is practised implies routinely giving mathematical justifications—albeit informal—of the "shortcuts" employed, is deceived. This would not be an issue if making an error in programming were exceptional. The current deplorable state of affairs can certainly be partially

ascribed to the ineptitude and ignorance of many programmers. But this is not the full explanation. It is true that Computer Science has yielded a number of results that make it possible to reason mathematically about programming, i.e., constructing a program that satisfies a given specification. But what is lacking is a manageable set of mathematical instruments to turn programming into an activity that is mathematical in its methods. To make it possible to discuss the—as yet hypothetical—discipline that would then be practised, I shall use the term "Algorithmics".

Mathematicians portrayed in cartoons are invariably staring at a blackboard covered with squiggles. To outsiders, mathematics = formulae. Insiders know that this is only the surface. But, undeniably, mathematics has only taken its high flight because of the development of algebraic notation, together with concepts allowing algebraic identities.[1]

The work reported on here has been motivated by the conviction that major parts of the activities of algorithm specification and construction should and can be performed in much the same way as that in which mathematicians ply their trade, and that we can profit in this respect from studying the development of Mathematics. Earlier work, based on the same conviction, can be found in GEURTS and MEERTENS[11] and MEERTENS[19]. In brief, the idea is that algorithms are developed by manipulating "algorithmic expressions". To be able to do this, we need a language that is capable of encompassing both specifications and programs. But, and this is important, this language should not be the union of two different languages, one a specification language, and the other a programming language. Rather, the language must be homogeneous: it must be possible to view all its expressions as specifications. Some of these expressions may, however, suggest a construction process more readily than others. Alternatively, all expressions can be viewed as abstract algorithms. Some of these algorithms may be so abstract, however, that they do not suggest an implementation.

The language should be comparable to the language used by mathematicians. Its *notations* give a convenient way to express *concepts* and thus facilitate reasoning, and also sustain more "mechanical" modes of transforming expressions (in the sense in which a mathematician transforms x^2-y^2 mechanically into $(x+y)(x-y)$).

In the long run, the development of algorithmics should give us "high-level" theorems, compared to which the few transformations we have now will look almost trivial. This is only possible through the growing development of higher-level concepts and corresponding notations. To get an idea of what I am dreaming of, compare the special product above with Cauchy's Integral Theorem, or with the Burnside Lemma.

1. The term "algebraic" is not used here in the technical modern sense (as in "algebraic data type"), but with the imprecise older meaning of "pertaining to Algebra" (as in "high-school Algebra"). The word "algebra" stems from the Arabic *al-jebr*, meaning "the [art of] recombining", originally used for bone setting. In the loose sense corresponding to that etymology, an identity like $\sin(x+y) = \sin x \cos y + \cos x \sin y$, in which the left-hand side is broken into constituents that are recombined to form the right-hand side, is algebraic.

The reader should carefully distinguish between

(i) the conviction—if not belief—that it is possible to create a discipline of "Algorithmics" that can be practised in the same style as Mathematics; in particular, by creating algorithmic derivations, using algorithmic expressions, with the same flavour as mathematical derivations and expressions;
(ii) the general framework around which the current investigations are built; namely a synthesis of an "algebraic" approach to data and to transformations (of data);
(iii) the concepts selected as worthy of a special notation in the language; and
(iv) the concrete notations and notational conventions chosen.

The program of research implied in (i) is closely related to the paradigm of "Transformational Programming"; see further Section 2. It is becoming increasingly clear (at least to me; I do not claim credit for the re-invention of the wheel) that a nice algebraic structure is a prerequisite for obtaining interesting results. Otherwise, no general laws can be stated, and so each step has to be proved afresh. (In fact, this is a truism, for what is an algebraic structure but a domain with operations, such that some general laws can be formulated.) This is also a major thought underlying the work on an "algebra of programs" of BACKUS[1]. A difference with the approach described here can be found in his motivation to overcome the "von Neumann bottleneck", resulting in a determined attempt to eschew variables for values (data, objects) even in their conventional mathematical roles, generally not considered harmful. More important is that Backus's "FP" framework is restricted to function *schemata*, and has (currently?) no place for an integrated algebraic view on data. (The approach described by GUTTAG, HORNING and WILLIAMS[12] allows algebraic specifications of data types but has more the nature of grafting them on FP than of integration.) It is clear, however, that the results obtained in his approach are valuable for the approach taken here, and that the correspondence merits further study. Integration of the data algebra with the algebra of operations on data can be found in the work by VON HENKE[13]. The emphasis there is on concepts; no attention is paid to notation.

The concepts and notations used here have grown out of my attempts to use the notations suggested by BIRD[4]. In trying to develop some small examples, I was struck by the similarity of many of the laws formulated in [4] (and some more I had to invent myself). Investigating this intriguing phenomenon, I discovered the higher-level algebraic framework underlying various similar laws. This incited me to introduce modifications to the notation, aimed at exhibiting similarities in the laws. These modifications have gone through various stages; for example, the symbols for sequence concatenation and set union were initially chosen to be similar; now they have been made identical.

The specific notational conventions, of all ideas presented here, should be given the least weight. This is not to say that I feel that good conventions are of secondary importance. It is obvious, however, that much work has still to be done to strike the right balance between readability, terseness, and

dependability (freedom of surprises). Only through the use in actual algorithmic developments, by a variety of people, can progress be made.

Two examples are included. They were chosen as being the first two not completely trivial problems that I tried to do in the present framework.

1. Mathematics for shortcuts in computation

In the Introduction, it was claimed that to ascertain the validity of a more efficient way of doing computations is to perform mathematics. This is still true if the reasoning is informal: the important thing is that it *could* be formalized. A beautiful example is the feat ascribed to Gauss as a young schoolboy. Asked to compute the sum of an arithmetic progression, he astounded his teacher by turning in the correct answer while the other pupils were still labouring on their first additions. We cannot, of course, know with certainty (if the story is true at all) what his reasoning was. But a plausible possibility is the following. Assume, for concreteness, that the task was to sum the first one-hundred terms of the arithmetic progression 534776, $534776+6207 = 540983$, $540983+6207 = 547190$, $\cdots$. Think of all those numbers, written in a column, and the same numbers in a second column, but this time in reverse order. So the first number in the second column is the number on the last line of the first column, which is $534776+99\times6207 = 1149269$. Next, add the numbers horizontally, giving a third column of one-hundred numbers.

$$
\begin{array}{rcrcr}
534776 & + & 1149269 & = & 1684045 \\
540983 & + & 1143062 & = & 1684045 \\
547190 & + & 1136855 & = & 1684045 \\
\vdots & & \vdots & & \vdots \\
1136855 & + & 547190 & = & 1684045 \\
1143062 & + & 540983 & = & 1684045 \\
1149269 & + & 534776 & = & 1684045 \\
\hline
S & + & S & = & 168404500
\end{array}
$$

FIGURE 1. Reconstruction of young Gauss's mathematical reasoning

Now we see a phenomenon that is not hard to explain. If we go down by one line, the number in the first column will increase by 6207. The number in the second column will *decrease* by the same amount. The sum of the two numbers on each line will, therefore, remain constant. So the third column will consist of 100 copies of the same number, namely $534776+1149269 = 1684045$. Now, call the sum of the numbers of the first column S. (This is the number to be determined.) The second column must have the same sum, for it contains the same numbers. The sum of the numbers in the third column is then $2S$. This sum is easy to compute: it equals $100\times1684045 = 168404500$. So $S = \frac{1}{2}\cdot168404500 = 84202250$. This "reconstruction" is rendered schematically in figure 1. It is noteworthy that the proof involves an intermediate construction that, if actually performed, would double the effort. The method is

easily generalized: if a is the first term of the progression, b is the increment and n is the number of terms to be added, we find $a+(n-1)b$ for the last term, and so $S = \frac{1}{2}n\{2a+(n-1)b\}$. The use of variables does not make the reasoning any less informal, of course.

Now, this was just an example, but substantial parts of mathematics consist of showing that two different construction methods will (or would) give the same result. Often one of the two is the original formulation of a problem to be solved, and the other one gives a construction that is much easier to perform.

It is also interesting to dwell for some time on the question of when we consider a mathematical problem solved. In mathematics we make no sharp distinction between the problem space and the solution space: both "problems" and "solutions" may have the form of construction methods. To call an answer a "solution" requires in the first place that it have the form either of a construction method, or of a *problem* for which we have, in our mathematical repertoire, a standard method for solving it. This requirement is not sufficient. For example, a mathematician will respond to the problem of determining the larger root of $x^2-2x-4 = 0$ by answering: $1+\sqrt{5}$, and consider the problem to be thereby solved. But what is the meaning of "$\sqrt{5}$" but: "the larger root of $x^2-5 = 0$"? So the problem is "solved" by reducing it to another problem. It is true that we have methods to approximate $\sqrt{5}$ numerically—for most purposes the best one is the Newton-Raphson method—but such methods will serve *equally well* to approximate the larger root of $x^2-2x-4 = 0$. Apparently, "to solve" does not simply mean: "to reduce to a case that we know how to handle". If that were the meaning, any quadratic equation would be its own solution. Out of the possibly many candidates for being solutions according to this requirement, mathematicians select one that allows a concise, elegant, formulation. We shall return to this issue in a discussion of mathematical notation, in Section 3.

2. Transformational Programming

The first published method for proving program correctness with mathematical rigour is that of Floyd [10]. Essentially the same method was suggested earlier by Naur [21]. Better known is the (semantically related) axiomatic approach of Hoare [14]. A technical objection to these methods is that they require the formulation of "intermediate assertions", i.e., predicates whose domain is the state space of an abstract machine; in more complicated cases, these predicates may grow into veritable algorithms themselves, and the conventional notations from predicate logic do not suffice to write them down. What makes program proving especially unsatisfactory is the following. The *activity* of programming, even in its present undisciplined form, already implicitly contains the essential ingredients for the construction of a correctness proof. These ingredients are present in the programmer's mind while developing the program. For example, a programmer may be heard muttering: "R must be at least 1 here, otherwise this code would not be reached. So I can omit this test and ...". None of this, however, is recorded.

Program proving requires now that a unique implicit correctness proof be made explicit *after the fact*. But such a reconstruction is in general much harder than to invent some proof in the first place. Also, it would be uneconomic to attempt to prove the correctness of a given program without verifying first that it handles several test cases successfully. But it is unrealistic to assume that programmers would go—unless forced—through the effort of proving apparently "working" programs correct.

This objection does not apply to the constructive approach advocated by DIJKSTRA [8], [9] and WIRTH [27], [28]. (The technical objection mentioned, however, does.) Here, the construction of the program is a *result* of the construction of the proof. Typical to the practical use of this approach, however, is that the program-under-construction is a hybrid, in which algorithmic notations are mixed with parts that are specified in natural language. For example, if we look over the shoulder of a programmer using this method of "stepwise refinement" or "top-down programming", we might see first:

"ensure enough room for T in *curbuf*"

in one stage of development, and in the next stage

while "not enough room for T in *curbuf*" **do**
  "ensure *nxtbuf* $\neq$ **nil**";
  curbuf, *nxtbuf* := *nxtbuf*, *nxtbuf*·*succ*
endwhile.

Although a big leap forward, the imprecision of the way the undeveloped parts are specified is unsatisfactory. In the example, it is probably the case that the task to "ensure enough room for T in *curbuf*" can be solved by emptying *curbuf*, and the task to "ensure *nxtbuf* $\neq$ **nil**" by the assignment *nxtbuf* := *curbuf*. But this would, in all likelihood, be incorrect, because of certain invariants to be maintained. It is, in principle, possible to attain the desired degree of precision, but the method itself does not incite the programmer to do so.

The same problem is not present in the method of "Transformational Programming"—at least, in its ideal form. In its essence, Transformational Programming is simple: start with an evidently correct—but possibly hopelessly inefficient—program, and bring this into an acceptable form by a sequence of "correctness-preserving" transformations. In contrast to mathematics, where the symmetrical relation "$=$", i.e., "is equal to", plays a central role, the central relation here is the asymmetric "may be replaced by",[1] denoted by "$\Rightarrow$". But at all stages, one has a correct program, with a precisely defined meaning. This way of manipulating a sequence of symbols

1. A simple example of this asymmetry is in the development of the task T = "Given a prime number p, find a natural number n such that n^2+n+p is composite". The development step that comes to mind (for a programmer) is to replace T by T' = "Find *the smallest* such natural number". A mathematician would probably replace the task by T'' = "Take $n = p$". Then $T \Rightarrow T'$ and $T \Rightarrow T''$. But T' and T'' are not interchangeable; for example, if $p = 2$, then T' finds $n = 1$, and in fact, they do not produce the same value of n for any value of p.

brings us closer to the ideal of "Algorithmics" aimed at. This is expressed in the following quote from a paper by BIRD [3], describing a new technique of program transformation: "The manipulations described in the present paper mirror very closely the style of derivation of mathematical formulas." There are several impediments to the application of this method. In the first place, the more usual algorithmic notations in programming languages suffer from verbosity. This makes manipulating an algorithmic description a cumbersome and tiring process. To quote [3] again: "As the length of the derivations testify, we still lack a convenient shorthand with which to describe programs." Furthermore, most programming languages have unnecessarily baroque semantics. In general, transformations are applicable only under certain conditions; checking these applicability conditions is all too often far from simple. The asymmetry of "$\Rightarrow$" makes these transformations also less general than is usual in mathematics. The requirement that the initial form be a program already (and "evidently correct", at that), is not always trivial to satisfy. In this respect, the method is a step backwards, compared to Dijkstra's and Wirth's approach. Finally, there is a very important issue: which are the correctness-preserving transformations? Can we give a "catalogue" of transformations? Before going deeper into that question, it is instructive to give an example.

Take the following problem. We want to find the oldest inhabitant of the Netherlands (disregarding the problem of there being two or more such creatures). The data needed to find this out are kept by the Dutch municipalities. Every inhabitant is registered at exactly one municipality. It is (theoretically) possible to lump all municipal registrations together into one gigantic data base, and then to scan this data base for the oldest person registered, as expressed in figure 2a in "pidgin ALGOL".

```
input dm, mr;
gdb := ∅;
for m ∈ dm do
  gdb := gdb ∪ mr[m]
endfor;
aoi := −∞;
for i ∈ gdb do
  if i·age > aoi then
    oi, aoi := i, i·age
  endif
endfor;
output oi.
```

FIGURE 2a. Program A for determining the oldest inhabitant

A different possibility is to determine the oldest inhabitant for each municipality first. The oldest person in the set of local Methuselahs thus obtained is the person sought. This is expressed in figure 2b.

Replacing (possibly within another program) program A by program B is then a transformation. Were there no inhabitants of the Netherlands, both

```
input dm, mr;
slm := ∅;
for m ∈ dm do
  alm := −∞;
  for i ∈ mr[m] do
    if i·age > alm then
      lm, alm := i, i·age
    endif
  endfor;
  slm := slm ∪ {lm}
endfor;
aoi := −∞;
for i ∈ slm do
  if i·age > aoi then
    oi, aoi := i, i·age
  endif
endfor;
output oi.
```

FIGURE 2b. Program B for determining the oldest inhabitant

programs would have an undefined result. This is generally not seen as affecting the applicability of the transformation $A \Rightarrow B$. But if—assuming at least one inhabitant in the country—some municipality had no registered inhabitants, then program A would have a defined result, whereas the outcome of B might be undefined. (The problem is that in the line "$slm := slm \cup \{lm\}$" the variable lm has no defined value if the empty municipality is the first one to be selected by "**for** $m \in dm$ **do**".) So the transformation $A \Rightarrow B$ has the following applicability condition:

$$(\forall m \in dm\colon mr[m] = \emptyset) \vee (\forall m \in dm\colon mr[m] \neq \emptyset).$$

We happen to know that for the given application this condition is satisfied, but it is easy to think of applications of this transformation where it is less obvious and has to be checked. Overlooking such conditions that are only exceptionally not satisfied is a typical source of programming errors. Note that a human interpreter of the original descriptions in natural language would almost certainly handle exceptional cases reasonably.

How large must a catalogue of transformations be before it is reasonable to expect it to contain this transformation? Obviously, unmanageably large. It is possible to have a manageable catalogue, and to require proofs of other transformations that are not in the catalogue. But how do you prove such a transformation? Hopefully, again with transformations, otherwise the practitioner of Transformational Programming needs two proof techniques instead of one. But what transformations will gradually transform A into B?

As another example, consider young Gauss's "transformation". This may be expressed as

```
input a, b, n;                                    input a, b, n;
sum, t := 0, a;                                   output (n / 2) × (2×a + (n − 1)×b)
for i from 1 to n do                    ⇒
  sum, t := sum + t, t + b
endfor;
output sum
```

Again, this is an unlikely transformation to be catalogued. Now compare this to the mathematical derivation:

$$\sum_{i=1}^{n}\{a+(i-1)b\} = \tfrac{1}{2}\left[\sum_{i=1}^{n}\{a+(i-1)b\} + \sum_{i=1}^{n}\{a+(i-1)b\}\right] =$$

$$\tfrac{1}{2}\left[\sum_{i=1}^{n}\{a+(i-1)b\} + \sum_{i=1}^{n}\{a+(n-i)b\}\right] = \tfrac{1}{2}\sum_{i=1}^{n}\{2a+(n-1)b\} =$$

$$\tfrac{1}{2}n\{2a+(n-1)b\}.$$

It is usual in presenting such derivations to omit obvious intermediate steps, and this one is no exception. For example, the first step has the pattern $S = \frac{1}{2}(S+S)$; a complete derivation would have $S = 1S = (\frac{1}{2}\cdot 2)S = \frac{1}{2}(2S) = \frac{1}{2}(S+S)$. Nevertheless, the only step that possibly requires looking twice to check it is the substitution of $n+1-i$ for one of the two summation variables i.

In what follows, an attempt is made to sketch an "algorithmic language" to overcome the drawbacks mentioned. To give a taste of what will be presented there, here, in that language, is the "transformation" $A \Rightarrow B$ of the oldest-inhabitant problem:

$$\uparrow_{age}/+/mr \star dm = \uparrow_{age}/(\uparrow_{age}/mr) \star dm .$$

Comparing this with figure 2a and 2b should explain my complaint about the verbosity of algorithmic languages. And yet that pidgin is a terse language when compared to those mountains of human achievement, from FORTRAN to Ada.® Note also the reinstatement of the symmetric "=", which will be explained in Section 6.

The emphasis on the similarity with Mathematics creates a clear difference with much of the work in the area of Transformational Programming, such as that of the Munich CIP group (BAUER *et al.* [2]). In that work, the emphasis is on creating a tool for mechanical aid in, and the verification of, program development. The prerequisite of mechanical verifiability puts its stamp on a language. Note that the language of Mathematics has not been developed with any regard to mechanical verifiability; the only important factor has been the sustenance offered in reasoning and in manipulation of formulae. In this respect, the approach of, e.g., BIRD [3] is much more closely related, even if its framework is different. To quote that paper once more: "[...] 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." The main point is, perhaps, that in my view the language should be "open", whereas mechanical verifiability requires a closed and frozen language. To prevent misunderstanding of my position, I want to stress that I sympathize with the thesis that systems for the complete verification of a development are extremely valuable, and that research and development in that area should be vigorously pursued. I hope—and, in more optimistic moments, expect—that the different line of approach followed here will, in the long run, contribute to better methods for program design and development, and to better systems for mechanical assistance in these tasks.

3. The role of notation in Mathematics

When Cardan breached his pledge of secrecy to Tartaglia and published the first general method for solving cubic equations in his *Ars Magna* (1545), he described the solution of the case $x^3 + px = q$ as follows [my translation]:

> RULE
> Raise the third part of the coefficient of the unknown to the cube, to which you add the square of half the coefficient of the equation, & take the root of the sum, namely the square one, and this you will copy, and to one [copy] you add the half of the coefficient that you have just multiplied by itself, from another [copy] you subtract the same half, and you will have the Binomium with its Apotome, next, when the cube root of the Apotome is subtracted from the cube root of its Binomium, the remainder that is left from this, is the determined value of the unknown.

This description strikes us as clumsy, but at the time, no better method was available. This "clumsiness" stood directly in the way of mathematical progress. Take, in contrast, a description of the same solution in present-day notation:

Solution of the equation $x^3 + px = q$.

Let $c = \sqrt{d}$, *where* $d = \left[\frac{p}{3}\right]^3 + \left[\frac{q}{2}\right]^2$, *and let* $b = c + \frac{q}{2}$ *and* $a = c - \frac{q}{2}$.

Then $x = \sqrt[3]{b} - \sqrt[3]{a}$ *is a root of the equation.*

What are the advantages of this notation? Obviously, it allows for a more concise description. Also, in Cardan's description, there might be some doubt whether "the half of the coefficient" itself, or its square, has to be added to and subtracted from the copies. In present-day notation, there is (in this case) no room for this doubt, and in general, parentheses will disambiguate (if necessary) anything. Both of these advantages, however, are insignificant compared to what I see as the major advantage of the "algebraic" notation used now, namely that it is possible to manipulate the formula for x algebraically.

So we see readily that

$$\begin{aligned} x^3 &= b - 3\sqrt[3]{b^2 a} + 3\sqrt[3]{ba^2} - a \\ &= (b-a) - 3\sqrt[3]{ba}\,(\sqrt[3]{b} - \sqrt[3]{a}) \\ &= q - (3\sqrt[3]{ba})\,x, \end{aligned}$$

and since

$$ba = c^2 - \left[\frac{q}{2}\right]^2 = \left[\frac{p}{3}\right]^3,$$

we see that indeed $x^3 + px = q$. No more than high-school mathematics was needed to verify the solution. A similar verification is impossible for the formulation in natural language. If, at the time, our notations had been available, then the solution of the cubic equation would not have had such a romantic history. A disadvantage of modern notation is its *suggestion* of abstruseness, of being an esoteric code. Undeniably, people can only profit substantially from the major advantage mentioned above if they not only know the meaning of the diverse squiggles, but are intimately familiar with them, which takes time and practice. I want to emphasize, however, that a description in natural language, as the one given by Cardan, is utter gibberish too to the mathematically uneducated reader. This point would have been obvious, had I chosen to use the "most literal" translation of the words in the Latin original, instead of present-day terminology. The rule would then have started: "Bring the third part of the number of things to the cube, ...".

In Section 1 I stated that a requirement for "solutions" is that their formulation be "elegant". This issue is connected to that of notation. It is matter of context, taste, conventions and tacit agreement between mathematicians, what constitutes "elegance". It is hard for us to understand why the ancient Egyptians were so keen on expressing fractions in terms of quantities $\frac{1}{n}$, as in

$$\frac{41}{45} = \frac{1}{2} + \frac{1}{5} + \frac{1}{9} + \frac{1}{10} = \frac{1}{2} + \frac{1}{3} + \frac{1}{20} + \frac{1}{36}.$$

For some reason, forms like $\frac{41}{45}$ did not belong to their solution space, but quantities like $\frac{1}{9}$ did. If we were to agree that, say, $Q(p, q, r, s)$, denoting the largest root of the equation $x^5 + px^3 + qx^2 + rx + s = 0$, belongs to our solution space, then suddenly the general quintic equation becomes solvable "algebraically". There is a reason for mathematicians not to take this way out. The squiggle approach is helpful only if mathematical practitioners can acquire sufficient familiarity with the squiggles, which imposes a limit on their number. Given this limitation, some criterion must determine which concepts are the winners in the contention for a notational embodiment. Two aspects determine the viability of a proposed notation. One is the *importance* of the concept: is it just applicable in some particular context, or does it come up again and again? The other is the amenability to *algebraic manipulation*: are there simple powerful algebraic identities expressible in terms of the notation considered? The Q-notation suggested above will be found lacking in both respects.

4. Notational conventions for functions and operations

A program operates on input and produces output. Whether that input be a "value", a data base, or a stream of requests, say, is immaterial to this abstract viewpoint. Similarly, it is immaterial if the output consists of values, modifications to a data base, or a stream of responses. In the usual approaches to programming languages, the distinction is, unfortunately, paramount in the concrete embodiment of the program. This obscures the deeper similarities in possible program development steps. So the first thing required is a uniform notation, reflecting a unified conceptual framework. The notation used here is that of a "function" operating on an "object". The result is a style that may be called "functional". However, I feel that the cherished distinction between a functional (or "applicative") style of programming, and a procedural (or "imperative") one, is not as deep as supporters/opponents of one or the other style would make it appear. A much deeper difference is the distinction between viewing an algorithmic expression, be it denoted as a function definition or as a while program, as an *operational* prescription for an automaton, or as an *abstract* specification determining a relationship between input and output. The price paid for taking the latter viewpoint is that this abstraction may make it hard to express some transformations that derive their relevance from performance characteristics of certain types of architecture. Such a transformation makes sense only if we commit ourselves to a decision on how the abstract specification is mapped to a process on a machine—although in due time several natural "canonical" mappings for various architectures may emerge. Moreover, if the inverse mapping is not defined, a low-level transformation may lack a high-level counterpart. (This problem occurs in high-level programming languages as well: try to express in Pascal, say, the low-level optimization that the storage for a global array variable that will no longer be referenced can be used for other purposes.) Since computing resources will always remain scarce—relative to our unsatiable need for processing—this is not a minor inconvenience. Some consolation can be found in the thought that many of these transformations are well understood and can be automated relatively well (e.g., recursion elimination; tabulation techniques; low-level data structure choice), possibly sustained by "implementation hints" added to the program text.

The main ingredients of our language will be "objects", (monadic, or unary) "functions", and (dyadic, or binary) "operations". Functions always take an object as argument, and return an object. Operations are written in infix notation, and may take an object, a function or an operation as left operand and an object as right operand. They return an object. Function application is (notationally) not treated as an operation (although, from a mathematical point of view, it is one, of course). It is simply denoted by juxtaposition, usually leaving some white space for legibility or to delineate the boundary between the lexical units involved. So, if f is a function and x is an object, fx stands for the application of f to x. If g is then applied to fx, this may be denoted by gfx. Function composition, usually written in mathematics in the form $g \circ f$, is *also* denoted by juxtaposition, without intervening operation.

This makes expressions such as $h\,g\,f$ and $g\,f\,x$ ambiguous. But semantically, there is no ambiguity: the expressions specify the same, since $(h\,g)f$ denotes the same function as $h(g\,f)$, and $(g\,f)x$ the same object as $g(f\,x)$. (The reader should note that these identities are algebraic, and about the simplest ones possible.) In fact, the wish to omit as many parentheses as possible without depending on priority rules motivated this unconventional convention. In particular, it removes the somewhat annoying disparity between an identity expressed on the object level, as in

$$f(g(x)) = g'(f(x)),$$

and its expression as functional identity, as in

$$f \circ g = g' \circ f.$$

A drawback is that this convention does not indicate how to denote the application of a functional (higher-order function) to a function argument; in the general case, a function may be so generic that it might both be composed with and be applied to another function. An example is the identity function; in that particular case, the distinction is semantically unimportant, but for other functions it is not. So some operation will be needed to denote function application in the general case. (Actually, it turns out possible to denote function application with the operations provided in the sequel, but only in a clumsy way.)

If $\times$ is an operation, then $x \times y$ denotes the application of $\times$ to x and y. In general, parentheses are needed to distinguish, e.g., $f(x \times y)$ from $(f x) \times y$. The interpretation of $f x \times y$ in the absence of parentheses is $f(x \times y)$. In a formula $x \times y \times z$, the absence of parentheses implies, likewise, the interpretation $x \times (y \times z)$. This convention is similar to the right-to-left parsing convention of APL.

Note. In derivations, chains may occur like $e_1 = e_2 = \cdots$. The connective signs ("$=$" etc.) in these chains are meta-signs, and are not to be confused with operations (in particular, the *operation* $=$, which takes two operands and delivers a truth value). They will always give precedence to the operations in the expressions e_i.

A further reduction of the number of parentheses is made possible by the following convention. An expression of the form "$\alpha\,;\,\beta$" stands for "$(\alpha)\,\beta$". The—purely syntactic—operator ";" takes lower precedence than the semantic operations. If several ";"s occur, they group from left to right: "$\alpha\,;\,\beta\,;\,\gamma$" stands for "$((\alpha)\,\beta)\,\gamma$".

An important convention is the following: If $\times$ is some operation, and x is an acceptable left operand for $\times$, then the notation "$x\times$" stands for the *function* $\lambda y\colon x \times y$. Note that $x \times y$ is now syntactically, but not semantically, ambiguous, since $(x\times)\,y$ denotes the same object as $x \times y$. In the notation $f x \times$ the meaning is always $f(x\times)$, so it denotes a functional composition. If the meaning $(f x)\times$ is intended, parentheses are required (or, equivalently, the notation $f x\,;\times$ can be used). This convention makes it also possible to define the meaning of an operation $\times$ in the following form:

Let x be ... . Then $x\times$ denotes the function F_x .

The meaning of $x\times y$ is then that of $F_x\ y$.

Now, for example, $1+\sqrt{}$ is defined: its meaning is $1+;\sqrt{} = \lambda y\colon 1+y;\circ\sqrt{} = \lambda x\colon 1+\sqrt{x}$.

Finally, if $\times$ is an operation that takes two objects as operands, and f and g are functions, then $f\times g$ stands for the function $\lambda x\colon (fx;\times g\,x)$.

The aim of these conventions is only to increase the usability of the formal language. The proof is therefore in the practical use. It will take time, and the experience of a variety of practitioners of Algorithmics, to find the most helpful notational conventions. Note that the current mathematical practice of using the sign "+" for addition and juxtaposition for multiplication, and to give multiplication precedence, has taken its time to become universally accepted—after the general idea of using an algebraic notation was already commonly accepted. Also, if the language is as open as the language of Mathematics, it is possible to adopt other conventions locally when this is more helpful in dealing with the problem at hand.

To define functions and operations concisely, we use, in addition to lambda forms, the convention of BURSTALL and DARLINGTON [6]. For example, the following lines define the Fibonacci function:

$$Fib\ 0 \Leftarrow 0;$$
$$Fib\ 1 \Leftarrow 1;$$
$$Fib\ n+2 \Leftarrow Fib\ n;+Fib\ n+1.$$

The variables on the left-hand side of " $\Leftarrow$ " are dummy variables for which values are to be substituted such that the left-hand side matches the actual function application; then the right-hand side, after applying the same substitutions, is equal to the function application and may replace it in a formula. This step is known as "Unfold"; the reverse operation as "Fold". A canonical evaluation can be defined by systematically unfolding, thus providing an operational semantics. BURSTALL and DARLINGTON show that an amazingly large number of transformations can be expressed as a sequence of Unfold/Fold steps. As long as $\Leftarrow$ is interpreted as equality, this is generally safe. If $\Leftarrow$ is interpreted in terms of the canonical evaluation, then a Fold step may introduce non-termination where it was not present.

5. STRUCTURES

In giving an algorithmic description, we are generally not only concerned with elementary values, like numbers and characters. These are combined into larger objects with a certain structure. For example, in some application we may want to compute on polynomials, represented as a sequence of coefficients, or with a file of debtors. The usual algorithmic approach to such aggregate structures has grown from the aim of obtaining an efficient mapping to the architecture of concrete computational automata. For the purposes of Algorithmics, we need a more algebraic approach. The domain of data on which a program operates usually has some algebraic structure. This fact

underlies the work in the field of algebraic data types. However, since the *motivation* there is not to obtain a simple algebra, but to achieve representation abstraction, the types as specified by way of example in the papers in this field are not usually algebraically (in the *al-jebr* sense) manageable. If they are, as for example the type of natural numbers, or the type of McCarthy's *S*-expressions, the structure of algorithms operating on objects of these types tend to reflect the structure of the objects. In algebraic terms, the function relating the input to the output is a homomorphism. This observation underlies the work by VON HENKE[13]. (The work by JACKSON[15]—best known outside of *Academia*—can be viewed as based on the same idea, although the term "homomorphism" is not used there.)

Let us start with algebraic structures that are about as simple as possible. Using the notation of MCCARTHY[17], we have

$$S_D = D \oplus S_D \times S_D .$$

This defines a domain of "D-structures", each of which is either an element of the (given) domain D (e.g., numbers, or sequences of characters), or is composed of two other D-structures. To practitioners of computer science, it is virtually impossible to think of these structures, McCarthy's "*S*-expressions", without a mental picture of an implementation with car and cdr fields from which arrows emerge. To mathematicians, however, this domain is simply a free groupoid, about the poorest (i.e., in algebraic laws) possible algebra, and computer-scientists will have a hard time explaining to them how arrows enter (or emerge from) their mental picture.

We need some notation for constructing such structures. We construct a D-structure by using the function "$\hat{}$" and the operation "$+$". If x is an element of D, then $\hat{}x$ will stand for the corresponding element of S_D. The monadic function $\hat{}$ is, of course, an injection. It is a semantically rather uninteresting function, and it could be left unwritten in many cases without ambiguity. As a compromise, the application of $\hat{}$ to x is written as $\hat{x}$ if this is typographically reasonable. If s and t are D-structures, then $s+t$ denotes the D-structure composed of s and t. The set S_D consists then of all structures that can be built from D by a finite number of applications of $\hat{}$ and $+$. (It is also useful to allow an infinite number of applications; this possibility will be ignored here to keep the treatment simple.)

The diligent reader will have noticed an important difference between the structures defined now, and the *S*-expressions as used for LISP. The value **nil** is missing. We can introduce it by writing (using "0" instead of "**nil**"):

$$S_D = D \oplus \{0\} \oplus S_D \times S_D .$$

Algebraically, however, this makes little difference; the domain obtained is isomorphic with $S_{D \oplus \{0\}}$, i.e., the one obtained by the previous construction if D is first augmented with an element 0. It becomes more interesting if we impose an algebraic law: $s+0 = 0+s = s$. This gives about the poorest-but-one possible algebra. Now we have a more dramatic deviation from the *S*-expressions, for it is certainly not the case that, e.g., cons(s, **nil**) $= s$.

The previous law is known as the *identity* law, and an element 0 satisfying this law is called an "identity (element)". Note that an identity can always be *added*, but that there is at most one identity in a groupoid.

We can go further and consider structures on which other algebraic laws are imposed. Of particular interest are the laws of *associativity*: $s+(t+u) = (s+t)+u$; of *commutativity*: $s+t = t+s$; and finally of *idempotency*: $s+s = s$. The interesting thing now is that the structures obtained correspond to familiar data structures: we get, successively, *sequences*, *bags*,[1] and *sets*. For sets, $\hat{}$ is the function $\lambda x\colon \{x\}$ and $+$ is the set union $\cup$. The identity law gives us the empty sequence, bag or set. This relationship between familiar algebraic laws and familiar data structures has been pointed out by BOOM[5]. Sequences correspond to what are known in algebra as monoids (or semigroups if there is no identity).

The usual way of characterizing sequences algebraically uses an operation "append (or prepend) an element". The choice between using "append" and "prepend" as the primitive operation introduces an asymmetry. The introduction of sequences by imposing associativity is quite symmetric. This way of introduction gives a uniform approach, exhibiting the essential and deep similarity between binary labelled trees (the S-expressions), sequences, bags and sets. This can be used to express laws that apply to all these kinds of structures. To stress the similarity, $+$ will be used in all cases; a disadvantage is that the type has then (at least in some cases) to be clear from the context. The notation S_D will likewise be used for all domains of such structures, and not be reserved for the free S-expressions.

To prove laws, we can use the following lemma:

INDUCTION LEMMA. *Let f and g be two functions defined on S_D, satisfying, for all $x \in D$ and s and $t \in S_D$:*
(i) $f0 = g0$,
(ii) $f\hat{x} = g\hat{x}$, *and*
(iii) $fs+t = gs+t$, *using the induction hypothesis that* $fs = gs$ *and* $ft = gt$.
Then $f = g$.

PROOF. By induction on the complexity of the function argument.

If S_D has no identity, then part (i) can of course be omitted. It is sometimes easier, in particular for sequences, to replace (ii) and (iii) together by $fs+\hat{x} = gs+\hat{x}$, which gives the traditional induction on the length. The advantage of the lemma as stated here is that it allows many laws to be proved independently of the algebraic richness of S_D.

To express interesting laws we first need some general operations, that also play an important role in Backus's FP. The notation used here for "applied-to-all" has been taken from [4]; the APL notation is used for "inserted-in".

1. Bags (or *multi-sets*), underrepresented in mathematics, are ubiquitous in computer science. They differ from sequences in that the elements have no order, and from sets in that an element can occur more than once.

Applied-to-all. Let f be a function in $D_1 \to D_2$. Then $f*$ stands for the function in $S_{D_1} \to S_{D_2}$ satisfying
(i) $f*0 = 0$,
(ii) $f*\hat{x} = \hat{}fx$, and
(iii) $f*s+t = f*s; +f*t$.

So f is applied to each "member" (elementary component) of its argument, and the result is a structure of the function values obtained. For example, if s is the set of numbers 0 through 9, then $1+*s$ is the set 1 through 10. For $f*$ to be well defined, it is required that $+$ on S_{D_2} have at least the same algebraic richness as its counterpart on S_{D_1}: if $+$ on S_{D_1} is associative, then so is $+$ on S_{D_2}, and so on. If S_{D_1} has no identity, we can simply omit part (i) from the definition. A similar remark can be made in most cases in the sequel: the laws are presented for structures with identity, but can easily be amended to cover identity-less structures.

Inserted-in. Let $\times$ be an operation in $D \times D \to D$. Then $\times/$ stands for the function in $S_D \to D$ satisfying
(i) if $\times$ has an identity e (so that $e \times x = x \times e = x$), then $\times/0 = e$,
(ii) $\times/\hat{x} = x$, and
(iii) $\times/s+t = \times/s; \times\times/t$.

So if $\times$ stands for the conventional multiplication operation, $\prod_{x \in S} x$ is a more familiar notation for $\times/s$. However, inserting an operator $\times$ in a structure s is only meaningful if $\times$ has at least the same algebraic richness as the operation $+$ used to construct the structure. This means that if $\times$ is multiplication, then the notation $\times/s$ is not allowed if s is a set, for (in general) $x \times x \neq x$. Otherwise, we would obtain contradictions like $2 = \times/\hat{2} = \times/\hat{2}+\hat{2} = \times/\hat{2}; \times\times/\hat{2} = 2\times 2 = 4$. (Alternatively, we could define the insertion as an indeterminate expression, depending on the choice of representatives from the congruence classes induced by the laws of $+$.)

The classes of functions $f*$ and $\times/$ are special cases of the homomorphisms definable on S_D. By combining them in the form $\times/f*$, all such homomorphisms can be expressed. This can be stated in the form of another lemma:

HOMOMORPHISM LEMMA. *Let the function $g \in S_D \to D'$ be a homomorphism, i.e., let there exist a function $f \in D \to D'$ and an operation $\times \in D' \times D' \to D'$ with identity $\times/0$, satisfying, for all $x \in D$ and s and $t \in S_D$:*
(i) $g\,0 = \times/0$,
(ii) $g\,\hat{x} = fx$,
(iii) $g\,s+t = g\,s; \times g\,t$.
Then $g = \times/f*$.

PROOF. By the induction lemma. For part (i), we have $g\,0 = \times/0 = \times/f*0$. For part (ii), $g\,\hat{x} = fx = \times/\hat{}fx = \times/f*\hat{x}$. For part (iii), by the induction hypothesis $g\,s = \times/f*s$ and $g\,t = \times/f*t$. Then $g\,s+t = g\,s; \times g\,t = \times/f*s; \times\times/f*t = \times/f*s+t$.

Note that this gives an algebraic formulation of the "Divide and Rule" paradigm. For part (iii) tells us that to rule a structure s that is not atomic (i.e., to compute $g\,s$), we can divide s in two parts, rule these, and combine the results appropriately.

The operations $*$ and $/$ give rise to three important new laws.

LAW 1. *Let* $f \in D_2 \to D_3$ *and* $g \in D_1 \to D_2$. *Then* $(fg)* = f* g*$.

LAW 2. *Let* $f \in D \to D'$, $\times \in D \times D \to D$ *and* $\times' \in D' \times D' \to D'$ *satisfy* $fx \times y = fx; \times' fy$ *and* $f \times / 0 = \times' / 0$.
Then $f \times / = \times' / f*$.

LAW 3. *Let* $\times \in D \times D \to D$ *and let* $+$ *operate on* S_D.
Then $\times / + / = \times / \times / *$ *(where these functions operate on* S_{S_D}*).*

PROOF. The proof (by induction) of law 1 is straightforward. Law 2 is an application of the homomorphism lemma, by taking $f \times /$ for g and $\tilde{\times}'$ for $\times$. Law 3 is an application of the same lemma, with $\times /$ for f and $\times / + /$ for g.

Each of these laws corresponds to a whole set of program transformations. Since the law $g*x + y = g*x; +g*y$ holds, and $g* + /0 = + /0$ (since 0 is the identity of $+$, we have $+/0 = 0$), we can apply law 2, with $g*$ for f and $+$ for both $\times$ and $\times'$, to obtain

COROLLARY. *Let* $g* \in S_D \to S_{D'}$. *Then* $g* + / = + / g**$.

The importance of the corollary is that it has no condition to be verified, in contrast to the complex applicability condition of the law from which it was derived.

This game can be continued on more complicated algebras. The simple cases dealt with above, however, already give rise to a surprisingly fruitful range of identities. For example, the identity mentioned in Section 2, which in functional form reads $\uparrow_{age} / + / mr* = \uparrow_{age} / (\uparrow_{age} / mr)*$, in which mr is used as a function, is derived as follows

$$\begin{aligned} \uparrow_{age} / + / mr* &= \uparrow_{age} / \uparrow_{age} / * mr* \quad \text{(by law 3, using } \uparrow_{age} \text{ for } \times\text{).} sp - .1v \\ &= \uparrow_{age} / (\uparrow_{age} / mr)* \quad \text{(by law 1).} \end{aligned}$$

This identity applies then to trees, sequences, bags and sets. Indeed, the transformation $A \Rightarrow B$ is valid, irrespective of whether the inhabitants are registered in orderly ledgers, or in bags. It is possible that $\uparrow_{age} /$ is not meaningful on the structures considered, but then both sides of the identity are meaningless.

A particular type of structure is obtained by taking the point domain $\{\iota\}$, containing one single element ι. Assume $+$ is at least commutative, and define $1 = \hat{\iota}$. Then each member of $S_{\{\iota\}}$, except 0, can be written in the form $1 + \ldots + 1$. In this particular case, associativity implies commutativity, since the 1s are indistinguishable. (This is not true if we allow infinite structures.) If identity, associativity and commutativity are the only laws for $+$, so that, e.g., $1 + 1 \neq 1$, then $S_{\{\iota\}} = \mathbb{N}$, the natural numbers, and $+$ has the conven-

tional meaning of addition. If idempotency holds too, we obtain a set with two elements, 0 and 1, which will be identified with "false" and "true", respectively. The meaning of $+$ on this domain is that of $\vee$, the "logical or" operation.

6. Fictitious values

Since antiquity mathematicians have been confronted with equations that, although not inconsistent, were nevertheless "impossible". A simple example is the equation $s+8 = 5$. If a shepherd adds eight sheep to his flock, it is impossible that the result is that the flock contains five sheep. And yet, discovered the mathematicians, it is possible to practise an internally consistent mathematics with fictitious quantities such as "3 short". In this way the notion of "number" has been extended from natural to, successively, integral, rational, algebraic, real and complex numbers. Today we are so familiar with all this that it is hard to realize what triumph of intellect the invention must have been to denote "nothing", something "non-existent", with a symbol like "0". Why has mathematics gone the way of accepting "fictitious values" on an equal footing? The answer must be that for mathematical practice the simplicity of the algebraic laws prevailed over semantic doubts about the necessary extensions of the notion of "value". Nowadays, we feel no qualms in stating that the set of primes that are also squares is empty, rather than that such a set is "impossible". Only one century ago, this was not so easy. The well-known mathematician C. L. Dodgson—well-known for other than his mathematical writings—advocated that universal quantification over such an "impossible" set would stand for a contradiction. Nobody could have worded the arguments better than he, but nothing has stopped mathematics from going the way of algebraic simplicity, in spite of all "common sense", leading to the currently universally accepted interpretation, which is just the reverse. So now we have

$$(\forall x \in S: p(x)) \supset (\forall x \in S': p(x)) \text{ for all } p \text{ iff } S' \subset S.$$

The Carrollean definition would have required, instead of "iff $S' \subset S$", the much more complicated "iff $S = \emptyset \vee S' \neq \emptyset \wedge S' \subset S$". Yet it is important to realize that all this is a matter of convenience, and not of mathematical necessity. If, for example, we define $<$ between sets over an ordered domain by

$$S < T \text{ iff } \forall s \in S: \forall t \in T: s < t,$$

then under the present interpretation $<$ is not transitive, whereas it would have been so, had nineteenth-century "common sense" prevailed. So the advantages of the current convention are not unequivocal.

The problem that arises in the oldest-inhabitant problem treated in Section 2 if some municipality is without inhabitants, can be solved by introducing the fictitious value "Nobody". In more mathematical terms, the domain of inhabitants forms a semi-lattice (disregarding inhabitants of equal age), and, as is well known, it is always possible to add some bottom element to it. If we

denote the operation of the semi-lattice by "$\uparrow_{age}$", then the oldest inhabitant of a set s of inhabitants is given by $\uparrow_{age}/s$, and so this "Nobody" is $\uparrow_{age}/0$. If Nobody is next compared to somebody, somebody will be chosen, since $s\uparrow_{age}\uparrow_{age}/0 = s$. This explains why "$\Rightarrow$" could be replaced by "$=$". In general, if some operation $\times$ has no identity in its domain, we can extend the domain by adding $\times/0$ as its identity. The properties of $\times/0$ are completely determined by the relevant algebraic laws. In particular, we see that it is an identity of $\times$ from $x \times \times/0 = \times/\hat{x}; \times\times/0 = \times/\hat{x}+0 = \times/\hat{x} = x$. Such a fictitious value can drastically simplify an algorithmic description; for that reason, it is not uncommon to find the notation ∞ in algorithms described in "pidgin ALGOL". The important insight is that such a domain extension is, in general, consistent. Inconsistencies can arise through additional laws, or through interference between laws involving several operations in a domain. To give an example of the possible pitfalls, let the operation $\ll$ be defined by

$$x \ll y \Leftarrow x .$$

This operation is associative, since $(x \ll y) \ll z = x \ll (y \ll z)$. The function $\ll/$ selects the first element of a sequence (or the leftmost element of a tree). Now consider $\ll/0$, where 0 is the empty sequence. Then $\ll/0; \ll x = x$, since $\ll/0$ is the identity of $\ll$. But from the definition of $\ll$, we have $\ll/0; \ll x = \ll/0$. So $x = \ll/0$ for arbitrary x. The problem arises since the law $x \ll y = x$ has already assigned a value to a formula containing the newly introduced identity. In fact, each element is a so-called right-identity of $\ll$; if a semi-group contains both a left- and a right-identity, then it is well known that they must coincide. If, for algorithmic purposes, a fictitious element $\ll/0$ is desirable, we must choose between two possibilities to retain consistency: either restrict the law $x \ll y = x$ to $x \neq \ll/0$, or use $\ll/0$ as a right-identity only (in which case the law $\ll/s+t = \ll/s; \ll \ll/t$ requires, of course, the restriction $s \neq 0$). Which solution is best depends on the context.

For the applicability of the methods of "transformational programming" and especially of "programming by stepwise refinement", it is important that algorithmic descriptions allow a certain amount of "indeterminacy". We may then find descriptions like "Let x be an element of s". The correctness of the algorithm does not depend on the element chosen, and so permits arbitrary choice. This type of "arbitrariness" should not be confused with the intended chaotic arbitrariness of pseudo-random generators. It only indicates a freedom that is left in realizing the algorithm, and which can be used, e.g., to achieve a simplification through a judicious choice of x. Now what if $s = 0$, the empty structure? The usual approach is then that the meaning of "Let x be an element of s" is "undefined", an entity that is loved by semanticists but best avoided by programmers. Let us use the symbol $\square$ to denote an unspecified choice: the operation of making an arbitrary choice between two values. So $x \square y$ is a specification that is satisfied by any solution for x, but also by any solution for y. The expression $1 \square 2$ may yield 1, but may as well yield 2 (but not 3). The operation $\square$ is associative: $(x \square y) \square z$ is equivalent to $x \square (y \square z)$. It

is also commutative and idempotent. So $\square/s$ stands for an "arbitrary" choice from the structure s. Choosing from an empty structure can now be described with the formula $\square/0$. But no choice is possible, so what is the meaning of this formula? The answer is: "Nothing". A more learned answer is that $\square/0$ represents the unsatisfiable specification. In essence, the question is as unanswerable as the question what it means to take the square root of -1. The meaning of $\square/0$ is given by the algebraic laws it satisfies; beyond that, it has no inherent meaning, any more than ∞, $\sqrt{-1}$, $\sqrt{2}$, $\frac{1}{2}$ or, for that matter, -3 have one. So, in particular, its meaning is that it satisfies $x \square \square/0 = x$. In words, if we may choose "freely" between x and Nothing, then we must choose x.

An important identity for $\square$ is

$$fx \square y = fx; \square fy.$$

This corresponds to what is known in Formal Semantics as the "monotonicity" of f. We know then, from law 2 of Section 5, that $f\square/ = \square/f*$. A prerequisite for general applicability of this law here, is, however, that the function be "strict", i.e., that the identity $f\square/0 = \square/0$ be satisfied as well. (In Formal Semantics, a function f is called "(error-)strict" or "bottom preserving" if $f(x)$ is "undefined" (or "the error value") whenever x is. The pseudo-value $\square/0$ can serve here, more or less, as a denotation of an "error value".) Many other identities require that the functions involved be strict. That a function is indeed strict will sometimes follow from its definition. In other cases, such as for the constant function $0\ll$, it does not; if strictness is not necessary, we have to specify what we want. It is, of course, possible to take strictness of functions as an immutable characteristic of the framework. But this is undesirable. In particular, if $\square/0$ is an identity of the operation $\square$, this gives simpler algebraic laws. Since then $x \square \square/0 = x$, the function $x\square$ cannot be strict for satisfiable x, and so the identity $x \square \square/s = \square/x\square * s$ requires the restriction $s \neq 0$. A reasonable convention appears to be that a function f is only strict if the algebraic identities assign no other meaning to $f\square/0$, or, of course, if strictness is explicitly specified. Then $\hat{}$, $+$, and all functions of the forms $f*$ and $\times/$, are strict. Moreover, $=$ must be strict, to prevent pathological paradoxes as would be created by $fx \Leftarrow$ if $fx = \square/0$ then x else $\square/0$.

We can now define the asymmetric relation $\Rightarrow$ in terms of $=$ and $\square$, for $p \Rightarrow q$ has the same meaning as $p = p \square q$. A consequence is that $p \Rightarrow \square/0$ for each p; for that reason programmers are well advised not to interpret "$\Rightarrow$" too literally as "may be replaced by": otherwise, "Nothing" would remain of programming.

7. Abstract algorithmic expressions

The expressions we have encountered until now are algorithms, in the sense that we could construct an automaton that accepts such expressions and—provided that the value of all variables is known—produces a result in a finite amount of time. The first mathematical formulae were, likewise, computational prescriptions. When we now manipulate formulae, it is the exception

rather than the rule that we are concerned with the efficiency of evaluating the formula; whether we replace x^2-y^2 by $(x+y)(x-y)$, or prefer the replacement in the opposite direction, depends on the context. Likewise, we must abandon our fixation on efficiency if algorithmics is to enjoy a fruitful development. In general, developing an efficient algorithm will require that we first understand the problem, and for this we need simple algorithmic expressions; but to simplify an expression we have to shed our old habits. In mathematics, a formula like $\limsup_{n\to\infty} a_n^{1/n}$ shows that the thought of a constructive prescription has been abandoned. For algorithmics, it is similarly useful not to cling to the idea that every algorithmic expression must be interpretable by an automaton. An interesting step, that has not yet been explored, is to extend the notion of "structure" to structures whose finite constructibility is not guaranteed, or is even provably impossible. So, for example, the function *infrep* defined by

$$infrep\ x \Leftarrow \hat{x} + infrep\ x$$

would define an infinite structure of x's.

For the time being, the primary purpose is to allow algorithmic expressions that serve purely as specifications. An example of a possible specification is, in natural language, "a counterexample to Fermat's Last Theorem". Even though we do not know, at the time of writing, how to construct one, we can (in theory) recognize one if it exists. But even the uncertainty about the existence of a counterexample does not make the specification vague; it has a precise and well-understood meaning. Allowing such "unexecutable" specifications to be expressed in the language of algorithmics makes it possible to keep the complete trajectory, from the initial (formal) specification to the final algorithm, in one unified framework. Many transformational derivations start with an expression that is theoretically executable, but not in practice; in particular, they tend to take the form of "British Museum" algorithms, in which a finite but exceedingly large search space is examined. An advantage is that one may hope to run this initial "specification" for a very small example. A disadvantage is that it is not always trivial to give an expression for the proper search space; the requirement that it be finite may increase the distance from the true specification. Also, it is not unthinkable that this step might introduce an error (some relevant case not included in the search space); particularly so since it precedes the formal development. It turns out that we can use one particular "unexecutable" expression to denote a "sufficiently large" search space. It will be denoted by "$\mathbb{U}$", and its meaning is, informally, the "universe" of all possible objects that are meaningful, i.e., of the right type, in the given context. The trick is that the notation $P:s$, where P is a predicate, stands for the collection of elements of s that satisfy P. A more traditional notation is $\{x \in s \mid P(x)\}$; however, ":" works also on structures other than sets. The meaning of $\{x \in \mathbb{U} \mid P(x)\}$ is then understood to be the same as that of the common notation $\{x \mid P(x)\}$. So, if C is a predicate testing for the property of being a counterexample to Fermat's famous claim, then $C:\mathbb{U}$ specifies *all* counterexamples, and $\square/C:\mathbb{U}$ specifies *a* counterexample.

8. Semantics for algorithmic expressions

How important it is to have a formal semantics for algorithmic expressions depends on the degree to which we want to place confidence in the meaningfulness of purely formal manipulations. My feeling is that in the current stage, a requirement that each proposed construction be accompanied by a formal definition of its meaning, so that each transformation could be formally justified, would be stifling. After all, great progress had been made in, e.g., Analysis, before Cauchy developed a firm foundation, and the paradoxes involved in summing divergent series have not led to disaster. Well-known examples where theory followed the application are Heaviside's "Operational Calculus" and Dirac's δ-notation. In due time, if the approach to Algorithmics investigated here proves its worth, possible paradoxes can be resolved by introducing higher-level concepts similar to, e.g., uniform convergence, to tighten the conditions of some theorems.

Still, some form of semantics would help to reason about aspects of proposed constructions. It is well known that we need extremely sophisticated mathematical constructions to define denotational semantics for expressions involving unbounded indeterminacy, and the desire also to allow infinite objects in the domain of discourse will hardly simplify matters. This seems to defeat the original motivation for defining semantics in a denotational way, namely to define meanings in clearer terms (i.e., better amenable to formal reasoning) than possible under the usual operational approach. In our case, the situation is even worse. For the intention is that the algorithmic expressions serve equally well as *specifications*. But specifications requiring an inordinate mathematical ability to understand them in the first place, are pretty useless. An operational semantic definition is, of course, out of the question (but see the next Section). A possible approach is the following.

Let $\mathfrak{E}$ stand for the set of algorithmic expressions. It is assumed that, next to the usual well-formedness criteria, other aspects, such as typability, are prerequisites for acceptability as an expression of $\mathfrak{E}$. To simplify the treatment, we assume that $\mathfrak{E}$ is recursive, and that $\mathfrak{E}$ contains a recursive subset $\mathfrak{V}$ of expressions that are identified with "values" (e.g., "2", or "$\lambda x : x+1$"). Intuitively, we can interpret an expression e of $\mathfrak{E}$ as "specifying" one, or more, or possibly no, elements of $\mathfrak{V}$. Define $\mathfrak{B}(e)$ to be the set $\{v \in \mathfrak{V} \mid e \text{ "specifies" } v\}$. Alternatively, we can interpret e as a "task" to find or construct some element of $\mathfrak{V}$. That task might have several solutions, or be impossible. Define $e \Rightarrow e'$ to mean: the task e can be solved by solving the task e'. The relation $\Rightarrow$ is a subset of $\mathfrak{E} \times \mathfrak{E}$. We can think of $\Rightarrow$ as "may be transformed to". The relation $\Rightarrow$ is reflexive and transitive (which may be ensured by taking the reflexive and transitive closure of some initial relation). Under the interpretation of an expression e as specifying elements of $\mathfrak{V}$, we would certainly expect e to specify a given $v \in \mathfrak{V}$ whenever $e \Rightarrow v$. On the other hand, if $v \in \mathfrak{B}(e)$ has been established, then v is a solution of the task e, so we have $e \Rightarrow v$. It follows that $\mathfrak{B}(e) = \{v \in \mathfrak{V} \mid e \Rightarrow v\}$. This gives a characterization of $\mathfrak{B}$ in terms of $\Rightarrow$. If we define the relation $\equiv \subset \mathfrak{E} \times \mathfrak{E}$ by $e \equiv e'$ iff $e \Rightarrow e'$ and $e' \Rightarrow e$, then $\equiv$ is an equivalence relation. We can, in the usual

way, step from $\mathcal{E}$ (and $\mathcal{V}$) to the equivalence classes induced by $\equiv$ in these sets. For convenience, the classes may still be denoted by some representative; but where formerly we had to write $e \equiv e'$, now we have $e = e'$.

When may a task e be replaced by a task e'? A requirement is certainly that any solution to e' be a solution to the original task e. So $e \Rightarrow e'$ requires $\mathcal{B}(e') \subset \mathcal{B}(e)$. We take this as the characterization of $\Rightarrow$ in terms of $\mathcal{B}$, replacing "requires" by "iff". This has some consequences. Call an expression f "flat" if $\mathcal{B}(f)$ is the empty set. An example of a flat expression is $\square/0$ (assuming that we do not admit this pseudo-value in the distinguished company of the proper values). Then we find, for any e, $e \Rightarrow \square/0$. But $\square/0$ can hardly be considered a reasonable replacement for e, unless e happens to be flat too. So, possibly, a more reasonable characterization of $\Rightarrow$ in terms of $\mathcal{B}$ might additionally require the "preservation of definedness", meaning that a non-flat expression may not be replaced by a flat one. This gives rise to rules that are more complicated, which is a reason for rejecting this approach. Instead, it is better to accept the validity of $e \Rightarrow \square/0$, with the consequence that the meaning of $\Rightarrow$ does not correspond exactly to the intuitive notion of "may (as a task) be replaced by". The preservation of definedness has then to be proved separately for derivations involving $\Rightarrow$. It is generally easier to do this once than to check it for each individual derivation step.

There is another important difference between the usual formal treatment of the refinement relation between algorithms (see, e.g., MEERTENS[19]), and the relation $\Rightarrow$. For, in the usual treatment, one has $\square/0 \Rightarrow e$ for any e. This is unacceptable here, since we would then find that each $e = \square/0$. See, however, the notion of "total variant" of a function defined below.

If we start with some definition of $\mathcal{B}$, next derive $\Rightarrow$ from that definition, and use $\Rightarrow$ then to find $\mathcal{B}$, this will be the original function we started with. If, however, we start with some definition of $\Rightarrow$, use that to define $\mathcal{B}$ and use this function to determine $\Rightarrow$, the latter relation may be larger than the original one. Next to transitivity and reflexivity, a "complete" relation $\Rightarrow$ satisfies a stronger closure property:

$$\text{If } \{v \in \mathcal{V} \mid e' \Rightarrow v\} \subset \{v \in \mathcal{V} \mid e \Rightarrow v\}, \text{ then } e \Rightarrow e'.$$

In this way, a relation $\Rightarrow$ can be specified by giving an initial subset, in the form of rules like

$$e_1 \,\square\, e_2 \Rightarrow e_i, \; i = 1, 2.$$

But this still does not give the full story. A pleasant property of expression-forming constructions is *monotonicity*: if $C[e]$ stands for an expression containing e as a *constituent* sub-expression, and $e \Rightarrow e'$, then we want to be able to conclude that $C[e] \Rightarrow C[e']$. This property is postulated for all constructions admitted to our language (and so $\mathcal{B}$ is excluded).

It is necessary to give a meta-rule for $\Rightarrow$ on functions, since equality of functions is not in general decidable. (The notion of "function" includes here our binary operations.) A reasonable rule appears to be:

META-RULE FOR $\Rightarrow$ ON FUNCTIONS.
Let f and $f' \in D \to \mathbb{V}$ (where $D \subset \mathbb{V}$), and let $fv \Rightarrow f'v$ for all $v \in D \cup \{[]/0\}$. Then $f \Rightarrow f'$.

This rule makes a choice between several possibilities for defining $\Rightarrow$ on functions. The possibility chosen seems to be the more manageable rule. If functionals (higher-order functions) can operate on functions involving indeterminacy, the meta-rule must be used with caution. For assuming the reasonable identity $f [] g; x = fx; [] gx$, we are led to conclude that $f [] g = \lambda x: (fx; [] gx)$. Now take $f = \text{id}$ $(= \lambda x: x)$, $g = 3 \ll (= \lambda x: 3)$, and let $h = \lambda x: x [] 3$. Then $h = f [] g$. But if $F = \lambda\phi: (\phi 1; +\phi 2)$, then we find $Ff [] g = Ff; [] Fg = 1+2; [] 3+3 = 3 [] 6$, whereas $Fh = h1; +h2 = 1 [] 3; +2 [] 3 = 3 [] 4 [] 5 [] 6$.

The converse rule "If $f \Rightarrow f'$, then $fv \Rightarrow f'v$" results if the monotonicity postulate is applied to function application. A consequence is that if f is a partial function, but f' is total (i.e., never yields $[]/0$), then $f \Rightarrow f'$ cannot hold. However, it is often desirable to turn partial functions into total ones. For example, a problem specification may prescribe that error messages be given if certain conditions are not met. It may then be preferable to treat these error messages initially as "instances" of $[]/0$. Call f' a "variant" of f if $fv \Rightarrow f'v \neq []/0$ whenever fv is not flat. A useful curiosity is that if f is "determinate" (see below), then $f' \Rightarrow f$. This is also a sufficient condition to show that a determinate function f' is a variant of f. A "total variant", finally, is a variant that is a total function.

We also need rules for function applications. Unfortunately, the simple rule

$$(\lambda x: C[x])e = C[e]$$

is not enough. One counter-example is found by considering $f1 [] 2$, where $f = \lambda x: x - x$. Mechanical textual substitution gives $1 [] 2; -1 [] 2 = -1; [] 0 [] 1$, which, together with the above meta-rule, would lead to the conclusion that function application is not monotonic (or, worse, that $0 \Rightarrow 1$). Another problem is given by taking $h []/0$, where $h = \lambda x: x [] 3$ is—for the moment—taken to be a strict function. Textual substitution results in $[]/0; [] 3 = 3$, which is inconsistent with the identity characterizing strictness, namely $h []/0 = []/0$. Therefore, the rule for function application needs the condition that the expression for the argument is "determinate" (see below) and non-flat if the function is specified to be strict. This corresponds, roughly, to what is known as "call-by-value" semantics. Note, however, that it is not required to *evaluate* the argument; all that is needed is that we exhibit certain properties, for which some sufficiency conditions can even be given in terms of syntactic criteria. If the function definition does not involve more than a single occurrence of the argument, then indeterminacy of the argument is no problem. The reason that functions are non-strict by default should now be apparent: this choice simplifies the applicability condition of the rule. Note that for strict functions it is always safe to use the rule in the "Fold" direction, namely $C[e] \Rightarrow (\lambda x: C[x])e$.

An expression e is determinate if, for any two values v_1 and v_2 such that $e \Rightarrow v_1$ and $e \Rightarrow v_2$, we have $v_1 = v_2$. It seems reasonable to require all values to be determinate, which implies that $\Rightarrow$ and $=$ coincide on $\mathcal{V}$. All values are, by definition, non-flat. The function-application rule could then be stated by restricting the argument to values (as was already done for the meta-rule), with the advantage that the notions of "determinacy" and "flatness" need not be used. A problem arises, however, if we want to define $\mathcal{B}(h)$, where h is as above (but not strict). Since h is obviously indeterminate (we have both $h \Rightarrow \mathrm{id}$ and $h \Rightarrow 3\ll$), we do not want to allow $\lambda x\colon x \,\square\, 3$ as element of $\mathcal{V}$. No enumerable collection of determinate lambda forms, however, can capture the meaning of h. This is related to the problem mentioned above for equality of functions.

A function definition may contain several occurrences of the argument, as in

$$\mathrm{abs}\ x \Leftarrow \mathrm{if}\ x < 0\ \mathrm{then}\ -x\ \mathrm{else}\ x\,.$$

Suppose we want to show the equality

$$\mathrm{abs}\ 2\times e = 2\times \mathrm{abs}\ e\,.$$

This is easily proved by the Unfold/Fold method:

$$\begin{aligned}&\mathrm{abs}\ 2\times e = \mathrm{if}\ (2\times e) < 0\ \mathrm{then}\ -(2\times e)\ \mathrm{else}\ (2\times e) = \\ &\mathrm{if}\ e < 0\ \mathrm{then}\ 2\times -e\ \mathrm{else}\ 2\times e = 2\times \mathrm{if}\ e < 0\ \mathrm{then}\ -e\ \mathrm{else}\ e = \\ &2\times \mathrm{abs}\ e\,.\end{aligned}$$

Unfortunately, the condition for the function-application rule is not satisfied if e is indeterminate. And yet, it is easy to see that in this particular case no harm is done. This insight can be generalized to the following meta-rule:

META-RULE FOR INDETERMINATE UNFOLD/FOLD.
Let $C[e]$ and $C'[e]$ be expressions containing e as a constituent expression, and let e occur at most once in $C'[e]$.
If there is a derivation $C[e] \Rightarrow C'[e]$ for determinate *e, and e is* uninterpreted *in that derivation, then $C[e] \Rightarrow C'[e]$ is also valid for indeterminate expressions e.*

This allows one to use, e.g., $e - e \Rightarrow 0$ or $1 \cdot e = e$, the latter by applying the meta-rule in both directions. This meta-rule is a corollary of the rules given above, as the following derivation shows:

$$C[e] \Rightarrow (\lambda x\colon C[x])\,e \Rightarrow (\lambda x\colon C'[x])\,e \Rightarrow C'[e]\,.$$

The middle step is an application of the meta-rule for $\Rightarrow$ on functions, together with the monotonicity property.

9. EXECUTABLE EXPRESSIONS

In going from specification to implementation, we can stop the development when we have an expression that has an obvious translation in terms of a program (i.e., it belongs to the "solution space"). If that translation is so obvious, then we can wonder if it could not be delegated to a machine. If that is possible at all (and it is certainly possible for some subset of the language $\mathcal{E}$ of

algorithmic expressions), then we effectively have a machine for executing some expressions. This would eliminate an uninteresting step that might easily introduce clerical errors. It also opens the possibility of having the machine apply certain optimizations that are hard to express without spoiling the clarity of the expressions, but that are nevertheless obvious (e.g., replacing recursion by iteration, or eliminating redundant computations).

In the current stage of this work, a serious effort to define an "executable subset" of the algorithmic expressions is still out of the question. We may wonder, however, what properties we would require of a hypothetical machine for executing expressions. Let $\mathfrak{E}$, $\mathfrak{V}$ and $\Rightarrow$ be as in the previous section. A possible approach is that the machine tries to mimic $\Rightarrow$, going through a sequence $e_1 \Rightarrow e_2 \Rightarrow \cdots$, hopefully ending up in a member of $\mathfrak{V}$. To the machine, the forms it operates on are states, rather than expressions. It is realistic to assume that the machine may have to attach some bookkeeping information to the expressions. To simplify the discussion, this possibility will be ignored. Obviously, we may not assume that the machine is capable of accepting all expressions of $\mathfrak{E}$ as states.

Let $\mathfrak{P}$ be a subset of $\mathfrak{E}$, standing for the "executable" expressions, i.e., the expressions that the machine is designed to cope with. (The letter $\mathfrak{P}$ has been chosen here because to us these expressions are programs for the machine.) We assume that $\mathfrak{P}$ and $\mathfrak{P} \cap \mathfrak{V}$ are recursive sets. Now we define $p \rightarrow p'$ to mean: if the machine is in the state p, it can, possibly, switch next to the state p'. So $\rightarrow$ is a subset of $\mathfrak{P} \times \mathfrak{P}$. There is no reason to require that the machine be deterministic, but it makes sense to assume that $\rightarrow$ is at least recursively enumerable. There must be some halting condition for the machine. A simple criterion is to have the machine halt if its state is a value, i.e., a member of $\mathfrak{V}$. This is then the output. For the sake of simplicity, we require all values to be "dead-end states", where p is a dead-end state if no state is reachable via $\rightarrow$ from p. Now we have two requirements:

> *Soundness.* Let $\rightarrow^*$ stand for the transitive and reflexive closure of $\rightarrow$. Then, for all $p \in \mathfrak{P}$ and $v \in \mathfrak{V}$, if $p \rightarrow^* v$, then $p \Rightarrow v$.
>
> *Preservation of Definedness.* Let p be an arbitrary non-flat member of $\mathfrak{P}$ (where the non-flatness is with respect to $\mathfrak{E}$). Then (a) if $p \rightarrow^* p'$, and p' is a dead-end state, then it is a value; and (b) there does not exist an infinite sequence of states $p_0, p_1, \cdots$ such that $p = p_0 \rightarrow p_1 \rightarrow \cdots$.

The first requirement is simply that the machine produce no wrong answers. The second one requires that *if* the program p, viewed as an expression, specifies a result (some value), then the machine will output a value when started in state p. Part (a) prohibits the machine from reaching a dead end without producing output (which, if it can be detected, can be interpreted as abortion of the program), whereas part (b) forbids infinite loops. It is, of course, in general undecidable whether the machine will halt if started in a given state p, so the proof would depend heavily on properties of $\Rightarrow$, such as monotonicity, and possibly of $\mathfrak{P}$.

A relation $\rightarrow$ satisfying the requirements for soundness and for preservation of definedness, may be called an "operational semantics" for $\mathcal{P}$. Note that different machines may correspond to different executable subsets of $\mathcal{E}$, and even that two machines operating on the same set $\mathcal{P}$ may differ in their operational semantics. So there is no such thing as *the* subset of executable expressions. In fact, let $\mathcal{P}$ be *any* executable subset, with operational semantics $\rightarrow$. Then it is always possible—provided that $\mathcal{E}$ is sufficiently expressive—to find some pair $\langle e, v\rangle \in \mathcal{E} \times \mathcal{V}$ such that $e \notin \mathcal{P}$ and $e \Rightarrow v$. Then $\mathcal{P} \cup \{e, v\}$ is also an executable subset, with operational semantics $\rightarrow \cup \{\langle e, v\rangle\}$. So there do not even exist maximal executable subsets of $\mathcal{E}$.

The "canonical evaluation" of programs in the style of BURSTALL and DARLINGTON [6] is one prime candidate for being an operational semantics. Some expressions have obvious translations into an imperative style, like $\uparrow_{age}/+/mr \cdot dm$ into the program of figure 2a of Section 2. $\mathcal{P}$ could be restricted to such programs, which could then be "compiled" into "pidgin ALGOL". Yet another possibility is translation into FP.

A problematic aspect is the evaluation of expressions such as $x \,[]\, y$. It is easy to imagine a machine that would always go to a state $x' \,[]\, y$ if $x \rightarrow x'$ for some x'. Note, however, that the machine is forced, by virtue of the requirement of preservation of definedness, to try the other choice if the preferred choice leads to a dead end without output. This corresponds, in a limited sense, to what is sometimes called "angelic nondeterminism". Operationally, however, no "nondeterminism" need be involved in this. But the same is also required if the first choice may lead to an infinite loop. Fortunately, the machine need not decide beforehand if this undecidable contingency will arise; it is sufficient if the evaluations of the alternatives are "dovetailed" (interleaved) in a fair way, i.e., not excluding some alternative indefinitely. In the context of a recursive function definition, this provides "automatic backtracking", where $[]/0$ takes the role of "Fail". To give a stronger example, consider

$$f x \Leftarrow \text{if } x = 0 \text{ then } f0 \,[]\, 1 \text{ else } 1.$$

It is then guaranteed that $f0 = 1$, since $f0 \Rightarrow f0 \,[]\, 1 \Rightarrow f1 \Rightarrow 1$, and no other value than 1 could be a possible outcome. Although this may not be the most pleasant thing to implement, neither is it prohibitively difficult or expensive, and certainly not if occurrences of $[]$ in "executable code" are the exception rather than the rule. It will often be possible to exhibit the non-flatness of expressions by a static analysis. If x is known to be non-flat, then the step $x \,[]\, y \rightarrow x$ is allowed.

10. SOME MORE BASIC OPERATIONS

If x and y denote two objects, $\langle x, y\rangle$ denotes an object that is a pair consisting of those two objects. The functions π_1 and π_2 allow the retrieval of the components from the pair, so, e.g., $\pi_2\langle x, y\rangle = y$. If $x \in D_1$ and $y \in D_2$, the pair $\langle x, y\rangle \in D_1 \times D_2$. If orderings are defined on the component domains, then the product domain is assumed to be ordered lexicographically, unless a different order is specified.

We have already encountered the operation $\ll$, which selects its left operand: $x \ll y = x$. An important application is that $x \ll$ denotes the constant function $\lambda y\colon x$. The operation $\gg$ selects its right operand (and so $x \gg$ is, for each x, the identity function id).

If x is a determinate object (meaning that no choice of the type $[]$ is involved), then $P?x$, where P is a predicate (i.e., a function returning a truth value), stands for $x \ll *Px$. This formulation has probably no immediately obvious meaning to the reader. Remember that "false" and "true" are identified with 0 and $1 = \hat{\iota}$, respectively. So, if Px is false, $P?x = x \ll *0 = 0$. If Px is true, $P?x = x \ll *1 = x \ll *\hat{\iota} = \hat{}x \ll \iota = \hat{x}$. We see now that $P?x$ means "if Px then $\hat{x}$ else 0". The operation ? is mainly (but not only) useful as auxiliary operation to define other operations. An important application is in the definition of a "filter": a function to "extract" all members of a structure satisfying a given property. The function $+/P?*$ returns the structure of all P-satisfying members of its argument. For example, if Px holds, but Py does not, we obtain $+/P?*\hat{x}+\hat{y} = +/(\hat{}P?x; +\hat{}P?y) = +/\hat{x}+\hat{0} = +/\hat{x}; ++/\hat{0} = \hat{x}+0 = \hat{x}$. It is important enough to merit a shorter notation; for this, we use $P:$, which we have already encountered. For example, the filter $x=:$ extracts all elements equal to x. We can then define

$$x\in \Leftarrow 0\neq x=:$$

to test for membership of x.

Some laws that use : are:

$$P:+/ \;=\; +/P:*;$$

$$x=:\mathbb{U} \;=\; \hat{x};$$

$P:f* = f*(Pf):$, provided that f is determinate;

$P:Q: = P \wedge Q;:$ (remember that $P \wedge Q; x = Px; \wedge Qx$).

The proof of the first, least obvious, law, is $P:+/ = +/P?*+/ = +/+/P?** = +/P:*$, in which the middle step is an application of the corollary of Section 5. The second law cannot be proved from previous laws, since no previous law involves $\mathbb{U}$; instead, it can be viewed as a (partial?) characterization of $\mathbb{U}$. The derivation of the third law is left as an exercise to the interested reader. (Hint: use the meta-rule for $\Rightarrow$ on functions from Section 8 to show first that $fx; \ll = fx \ll$, and next that $P?f = f*(Pf)?$.) The last law is most easily proved by proving it first for determinate predicates P and Q (by considering all possibilities of assigning truth values to Px and Qx), and then using the last meta-rule of Section 8.

An example of the use of these laws is given by

$$x\in P:\mathbb{U} \;=\; 0\neq x=:P:\mathbb{U} \;=\; 0\neq P:x=:\mathbb{U} \;=\; 0\neq P:\hat{x} \;=\; 0\neq P?x \;=\; Px.$$

Another important property connected with : needs some terminology. Call an operation $\times \in D\times D\to D$ "selective" if $[] \Rightarrow \times$, i.e., for all x and $y \in D$,

$x \,[]\, y \Rightarrow x \times y$. Examples of selective operations are $[]$ itself, $\ll$, $\gg$, and $\downarrow_f$ and $\uparrow_f$, to be defined below. The property is then:

> If $\times$ is selective and $\times/P{:}s \Rightarrow x \neq []/0$ for some structure s, then $Px \Rightarrow 1$.

The crucial step in the proof is $[]/P{:}s \Rightarrow \times/P{:}s$.

Another useful application of ? is in the definition of $\rightarrow$, where the predicate $p \rightarrow$ is defined by $[]/p \ll ?$, in which p is a proposition, i.e., an expression whose value belongs to the domain of truth values. (Since the operation ? requires a predicate as first operand, the operation $\ll$ is used to turn the proposition p into a predicate.) Then $p \rightarrow x; [] q \rightarrow y$ specifies, indeterminately, x or y, but x is only specified if p can be satisfied, and y if q can be. For example, assume that p holds and q does not. Then we find $p \rightarrow x; [] q \rightarrow y = []/p \ll ?x; [][]/q \ll ?y = []/\hat{x}; [][]/0 = x[][]/0 = x$. So the combination of $\rightarrow$ with $[]$ gives "guarded expressions", whose meaning is not primitive but is obtained by composing the meanings of the individual operations. Note that $0[]1; \rightarrow x = x$, since $0[]1; \rightarrow x = 0 \rightarrow x; []1 \rightarrow x$.

An important law for $\rightarrow$ is:

$$f\,p \rightarrow \; = p \rightarrow f, \text{ provided that } f \text{ is strict.}$$

Since $p \rightarrow$ is obviously strict, we have $p \rightarrow q \rightarrow \; = q \rightarrow p \rightarrow \; (= p \wedge q; \rightarrow)$.

If x and y are elements of a semi-lattice with greatest lower bounds, then $x \downarrow y$ stands for the greatest lower bound of x and y. The expression $\downarrow/0$ stands then for the top of the semi-lattice. If it has no top already, it can be extended with one in a consistency-preserving way. It is often profitable to identify $\downarrow/0$ with $[]/0$. The operation $\uparrow$ is defined similarly. Although it is likewise often useful to define $\uparrow/0 = []/0$ if the (semi-)lattice has no bottom, it is generally unsafe to use this device for both $\downarrow$ and $\uparrow$ if they can appear mixed in a formula.

On structures, we can define a default partial ordering

$$s \leqslant t \text{ iff } 0[]1; \ll : t \Rightarrow s.$$

So $s \leqslant t$ if s can be obtained by omitting some (possibly none) of the members of t. For sequences, $\leqslant$ corresponds then to "is a (possibly non-contiguous) subsequence of". For sets, natural numbers, and truth values, we find as meanings, respectively, "$\subset$", the traditional "$\leqslant$", and implication. Structures for which the construction operation $+$ is associative and commutative form now a lattice, and $\downarrow$ gives, e.g., "$\cap$" for sets and "$\wedge$" for truth values. The operation $\uparrow$ is then defined as well. Note that $\uparrow/0 = 0$, since 0 is an identity of the operation $\uparrow$.

The operation $<_f$, where f is a determinate function, is defined by

$$x <_f y \Leftarrow fx; <fy,$$

and $=_f$, $>_f$, etc., are defined similarly.

The operation $\downarrow_f$, for a determinate function f whose range is a domain with a total ordering, is defined by

$$x\downarrow_f y \Leftarrow (x \leqslant_f y\,;\, \rightarrow x)\,[]\,(y \leqslant_f x\,;\, \rightarrow y).$$

An identity relating $\downarrow_f$ to $\downarrow$ is $f\downarrow_f/ = \downarrow/f*$. The operation $\uparrow_f$ is defined similarly. It is again often helpful to define $\downarrow_f/0 = []/0$ or $\uparrow_f/0 = []/0$, with the same *caveat* for mixed use.

Finally, we need a function # to count the number of elements of a structure. This can be done by mapping each element to ι, so $\#\hat{x}+\hat{y} = \hat{\iota}+\hat{\iota} = 1+1 = 2$. So we can define # as $\iota \ll *$. There is a surprise, though: on sets (and more generally, on all structures with idempotency) this # refuses to count properly. The problem is that #, as defined, is a homomorphism. But the number-of-elements function on sets is not. That "number of elements" cannot be defined as a homomorphism on sets follows from the breakdown of the law $\# +/ = +/\#*$ (an application of the corollary of Section 5) for sets; in particular, $\#s$; $+\#s$ for a non-empty set s differs from $\#s+s = \#s$. The function $\iota \ll *$ is only defined on sets as a mapping to the *set* $S_{(\iota)}$, which is the domain of truth values, and it tests then for non-emptiness.

11. First example: A Text-Formatter

The following problem specification, copied from Bauer *et al.* [2], is a reformulation (under the heading "Text editor") of the original specification (under the heading "Line editing problem") given in Naur [22].

> "A text, i.e. a non-empty sequence of words separated by blanks (BL) or new line characters (NL), is to be re-structured according to the following rules:
> (1) every two words are separated by exactly one BL or NL;
> (2) the first word is preceded by NL; the last character is neither BL nor NL;
> (3) each line is at most MAX characters long (not counting NL); within this range, it contains as many words as possible.
> The input line is required to start with NL; further, no word must contain more than MAX characters."

As a first step, we aim at more abstraction. This can be done by assuming that a type "word" is already given, and that the function #, applied to a word, will give its length (some natural number). Then the input can be viewed as a single "line", i.e., a sequence of words, whereas the output is a sequence of lines. This abstract view makes requirements (1) and (2), the clarification "(not counting NL)" of (3) and the first part of the last sentence irrelevant, since they deal with the concrete representation of sequences of lines in terms of some character code. More important is that it guarantees that the algorithmic development will work for different representations. (If more concreteness is nevertheless required, it is still advantageous to split the problem into a more algorithmic part, and the treatment of the concrete representation. For the latter, mappings from the types "sequence of words" and "sequence of lines" to the type "sequence of (character or 'BL' or 'NL')" have to be defined, and the abstract algorithm obtained has to be transformed

to work on this new concrete representation. Techniques for effecting a change of representation are given in BURSTALL and DARLINGTON[6] and MEERTENS[18]. Hopefully, it will be possible in some future to leave such low-level transformations to an automated system.)

Next we have to make the natural-language specification more precise. The meaning of "A text ... is to be re-structured" is best expressed as a requirement on the relationship between the input and the output:

(0) the output, "unstructured", is the original input.

Furthermore, requirement (3) is best split into two parts:

(3a) each line of the output is at most of length MAX;
(3b) each line of the output contains as many words as is possible within the constraints imposed by (0) and (3a).

An observation can now be made: the specification is symmetric with respect to the directions left-to-right and right-to-left. More precisely, let *rev* be a function that takes a sequence as argument and returns the reverse sequence as result. Then we have:

> If a function f "solves" (0), (3a) and (3b) (i.e., for each acceptable input line i, $f\ i$ is acceptable output), then so does $rev * rev\ f\ rev$ $(= rev\ rev * f\ rev)$.

From (3b) we can derive the following requirement:

> No line of the output starts with a word that would have fit at the end of the previous line.

For, otherwise, that line contains fewer words than possible. Expressed very informally, this means: lines are "eager" to accommodate words as long as there is enough room. Because of the symmetry, a solution must then also satisfy the mirror-image "reluctant" requirement:

> No line of the output ends with a word that would have fit at the start of the following line.

But it is not hard to give input for which the "eager" and the "reluctant" requirements are, together, impossible to satisfy. An example, if MAX = 13, is the input "`Impossible.to.satisfy.in.both.ways!`". The unique "eager" solution is then

```
Impossible.to
satisfy.in...
both.ways!...
```

The "reluctant" solution is different:

```
Impossible...
to.satisfy...
in.both.ways!
```

Something is wrong. The "reluctant" approach tends to leave as much white space on the first line as possible. This is, by application of real-world knowledge, typographically undesirable. The "eager" approach, in contrast, leaves the last line unfilled. This is, if not typographically desirable, then at least neutral. This suggests to us replacing (3b) by:

> (3b′) each line *but the last, if any,* of the output contains as many words as is possible within the constraints imposed by (0) and (3a).

However, this still does not solve the "eager" *vs.* "reluctant" problem: just add a 13-character "word" (e.g., "`Exasperating!`") to the end of the example input given above. The problem with the specification seems to reflect our conditioning to think in terms of left-to-right. Whereas (0) and (3a) are "boundary conditions", (3b) is an "objective", namely, "Do not waste more space than necessary"; more precisely:

> (3b″) minimize the total white space on the output, not counting the last line.

This approach was suggested to me by Robert Dewar. There is still a tiny problem left: if the last line is completely filled, then another empty line may be added without penalty in terms of the white-space objective. So a second objective, subordinate to the previous one, is to minimize the number of lines of the output.

Now we are ready to start giving a formal treatment of the problem. This will be done in an unusually detailed way, comparable to the minuteness of the steps in $S = 1S = (\frac{1}{2}\cdot 2)S = \frac{1}{2}(2S) = \frac{1}{2}(S+S)$. We use the letter r for the input ("raw"), and c for the output ("cooked"). The proposition that the input/output constraints are satisfied, is denoted by $r{\sim}c$. If, furthermore, *obj* denotes the objective function, then the problem is to determine, for given input r,

$$f\,r \Leftarrow \downarrow_{obj}/\,r{\sim} : \mathbb{U}.$$

In words: take any *obj*-minimizing object c such that $r{\sim}c$. We put $\downarrow_{obj}/0 = []/0$. We must define $\sim$ and *obj*. If *len* is a function giving the length of a single line, then $\sim$, expressing that the two constraints (0) and (3a) are satisfied, can be defined as:

$$r{\sim}c \Leftarrow +/c = r;\ \wedge\uparrow/len*c \leqslant \text{MAX}.$$

The *len* of a line is the sum of the lengths of its words, plus 1 for each space between a pair of words. A simple way to obtain this result, is to add 1 to the length of each word before summing, and to subtract 1 from the sum. For an empty line, we have to define its length separately:

$$len\ 0 \Leftarrow 0;$$

$$len\ l{+}\hat{w} \Leftarrow -1;\ +\ +/(1+\#)*l{+}\hat{w}.$$

For a line consisting of a single word, we have, of course, $len\,\hat{w} = -1;\ +\ +/(1+\#)\cdot\hat{w} = -1;\ +(1+\#)w = \#w$. The objective function is defined by

$$obj\,c \Leftarrow \langle ws\,c,\ \#c\rangle,$$

where the "white-space" function *ws* gives the white space on its argument (not counting the last line). The white space left on a single line is given by the function $ws_1 = \text{MAX} - len$. This quantity has to be summed over all lines but the last. This gives us the definition:

$$ws\,c'+\hat{l} \Leftarrow +/ws_1\cdot c'.$$

To make the function total, we also define

$$ws\,0 \Leftarrow 0.$$

We turn now first to the question whether it is possible to satisfy the constraints, not bothering about the objective. One extreme approach to satisfy (0) is to have a one-line page, or $c = \hat{r}$. This is likely to violate constraint (3a). Since the white space does not matter, we can try the other extreme: use a separate line for each word. This would give us $c = \hat{}\cdot r$. Then (0) is, of course, satisfied, but what about (3a)? Since $len\,\hat{} = \#$, we find

$$\uparrow/\,len\cdot c = \uparrow/\,len\cdot\hat{}\cdot r = \uparrow/\,(len\,\hat{})\cdot r = \uparrow/\,\#\cdot r.$$

So, if $\uparrow/\,\#\cdot r \leqslant \text{MAX}$, i.e., each word on the input is at most MAX long, we have $r\sim\hat{}\cdot r$, so the problem posed is solvable. Next, we show that this condition is not only sufficient, but also necessary. If $l \neq 0$,

$$len\,l = -1;\ +\ +/(1+\#)\cdot l \geqslant -1;\ +\uparrow/(1+\#)\cdot l = -1;\ +\ 1+\uparrow/\#\cdot l = \uparrow/\#\cdot l.$$

In the given context, $\uparrow/0 = 0$, since line lengths are natural numbers. Then, if $l = 0$, $len\,l = 0 = \uparrow/\,\#\cdot l$, so no condition $l \neq 0$ is necessary for the inequality $len\,l \geqslant \uparrow/\,\#\cdot l$. Now we have

$$\uparrow/\,len\cdot c \geqslant \uparrow/\uparrow/\,\#\cdot\cdot c = \uparrow/\,\#\cdot +/c.$$

If $r\sim c$ is satisfied, $+/c = r$ and $\uparrow/len\cdot c \leqslant \text{MAX}$, so

$$\uparrow/\,\#\cdot r = \uparrow/\,\#\cdot +/c \leqslant \uparrow/len\cdot c \leqslant \text{MAX}.$$

In conclusion,

$$f\,r \neq []/0 \text{ if and only if } \uparrow/\,\#\cdot r \leqslant \text{MAX}.$$

To "synthesize" *f*, we must derive some properties of $\sim$ and *obj*. In the first place, empty lines can be deleted from the output without violating the constraints. For

$$+/c_1+\hat{0}+c_2 = (+/c_1)+(+/\hat{0})+(+/c_2) = (+/c_1)+0+(+/c_2) = (+/c_1)+(+/c_2) = +/c_1+c_2.$$

Also, $\uparrow/len\cdot\hat{0} = \uparrow/\hat{}len\,0 = \uparrow/\hat{0} = 0$, so

$$\uparrow/len*c_1+\hat{0}+c_2 = (\uparrow/len*c_1)\uparrow(\uparrow/len*\hat{0})\uparrow(\uparrow/len*c_2) =$$
$$(\uparrow/len*c_1)\uparrow 0\uparrow(\uparrow/len*c_2) = (\uparrow/len*c_1)\uparrow(\uparrow/len*c_2) =$$
$$\uparrow/len*c_1+c_2.$$

Combining these two gives

$$r\sim c_1+c_2 \text{ if and only if } r\sim c_1+\hat{0}+c_2.$$

Next, we show that empty lines are always disadvantageous in terms of the objective. To show this, we have to distinguish several cases, because of the form of the definition of *ws*. First, we treat the case where the empty line considered is not the last line. Since

$$+/ws_1*\hat{0} = +/\hat{}ws_1\ 0 = ws_1\ 0 = \text{MAX}-len\ 0 = \text{MAX},$$

we have

$$ws\ c_1+\hat{0}+c_2+\hat{l} = +/ws_1*c_1;+\text{MAX}++/ws_1*c_2 \geqslant$$
$$+/ws_1*c_1;++/ws_1*c_2 = +/ws_1*c_1+c_2 = ws\ c_1+c_2+\hat{l}.$$

If the empty line *is* the last, but not the only one, we find

$$ws\ c_1+\hat{l}+\hat{0} = +/ws_1*c_1+\hat{l} = +/ws_1*c_1;++/ws_1*\hat{l} \geqslant$$
$$+/ws_1*c_1 = ws\ c_1+\hat{l}.$$

Finally, if the whole document consists of just one empty line,

$$ws\ \hat{0} = ws\ 0+\hat{0} = +/ws_1*0 = +/0 = 0 = ws\ 0.$$

So in all cases

$$ws\ c_1+\hat{0}+c_2 \geqslant ws\ c_1+c_2.$$

Since

$$\#\ c_1+\hat{0}+c_2 = (\#c_1)+(\#\hat{0})+(\#c_2) = (\#c_1)+1+(\#c_2) >$$
$$(\#c_1)+(\#c_2) = \#\ c_1+c_2,$$

we have

$$obj\ c_1+\hat{0}+c_2 > obj\ c_1+c_2.$$

We may conclude that it is never helpful to consider output containing empty lines. This can be expressed formally by inserting a filter that sifts out pages with empty lines, e.g., by replacing $\mathbb{U}$ in the definition of f by $0\notin:\mathbb{U}$. On the set of pages without empty lines, *obj* has the same ordering as *ws*, so we can replace $\downarrow_{obj}$ in the definition of f by $\downarrow_{ws}$. We can now also use for the *len* function the uniform definition

$$len\ l \Leftarrow -1;\ +\ +/(1+\#)*l,$$

since we know that the function is not applied to an argument 0. This allows us to do some elementary mathematics. If $c \neq 0$, we can put $c = c'+\hat{l}$, so

$$
\begin{aligned}
&ws\,c = ws\,c' +\!\!\hat{}\, \hat{l} = +/ws_1 * c' = +/(\mathrm{MAX} - len) * c' = \\
&+/(\mathrm{MAX} - (-1) + +/(1+\#)*) * c' = \\
&+/(\mathrm{MAX} + 1; - +/(1+\#)*) * c' = \\
&\mathrm{MAX} + 1; \times \#c'; - +/+/(1+\#)** c' = \\
&\mathrm{MAX} + 1; \times \#c'; - +/(1+\#) * +/c'.
\end{aligned}
$$

If, furthermore, $r \sim c$, then $r = +/c$, so

$$
\begin{aligned}
&len\,r = len\,+/c = len\,+/c' +\!\!\hat{}\, \hat{l} = -1; + +/(1+\#) * +/c' +\!\!\hat{}\, \hat{l} = \\
&+/(1+\#) * +/c'; + (-1) + +/(1+\#) * l = \\
&+/(1+\#) * +/c'; + len\,l,
\end{aligned}
$$

so that we have

$$+/(1+\#) * +/c' = len\,r; -len\,l.$$

Combining these two gives us: if $r \sim c$ and $c = c' +\!\!\hat{}\, \hat{l}$,

$$ws\,c = \mathrm{MAX} + 1; \times \#c'; - (len\,r; - len\,l).$$

In using this formula to compare the outcome of *ws* on two different non-empty pages that both meet the constraints, we can replace the part "$-(len\,r; -len\,l)$" by "$+len\,l$", since r, and therefore $len\,r$, is fixed. Since then, moreover, $len\,l < \mathrm{MAX} + 1$, the quantity $\#c'$ prevails over $len\,l$ in the comparison. This leads us to consider the simpler function

$$lpos\,c' +\!\!\hat{}\, \hat{l} \Leftarrow \langle \#c', len\,l \rangle.$$

On non-empty pages, the ordering of *ws* is that of *lpos*. If we also define

$$lpos\,0 \Leftarrow \langle 0, 0 \rangle,$$

we may even drop the restriction to non-empty pages.

If we combine the above findings, we obtain the following definition for f:

$$f\,r \Leftarrow \downarrow_{lpos} / r \sim : 0 \notin : \mathbb{U}.$$

This formulation makes it possible to find solutions of $f\,r +\!\!\hat{}\, \hat{w}$ in terms of solutions of $f\,r$. The effect, as we will see, is that of following the "eager" strategy. We may thereby lose some other, equally optimal, solutions. Expressed in words, the crucial idea is the following. Suppose c is the result of formatting a given input text r. We can "truncate" c by "erasing" the last word on its last line, and the last line itself if it then becomes empty. Then the two data c-truncated and w, together with the knowledge that c-truncated was obtained by erasing w from an optimal solution c, suffice to reconstruct c uniquely. (It is assumed that the value of MAX is known.) Moreover, c-truncated is then an acceptable way of formatting r-truncated, and although it need not be an optimal solution, there is no harm done by replacing it by an optimal one. It follows then that an optimal solution for r (since we know it to exist) can be formed from an optimal solution for r-truncated. This will now be shown more formally. We define

$$Trnc\, c'+\hat{}\, l+\hat{w} \Leftarrow (l \neq 0; \rightarrow c'+\hat{l}) \,[]\, (l = 0; \rightarrow c');$$
$$Trnc\, r'+\hat{w} \Leftarrow r'.$$

(Note that the function *Trnc* is "overloaded" here: the two definitions operate on arguments from different domains.) So suppose $r \sim c$, and among all possible solutions the *lpos* of c is minimal. Suppose, moreover, $c \neq 0$, so $r = +/c \neq 0$ (remember that empty lines are excluded), and we can put

$$c = c'+\hat{}\, l+\hat{w}_1;$$
$$r = r'+\hat{w}_2.$$

From $r \sim c$ we have $r'+\hat{w}_2 = +/c'+\hat{}\, l+\hat{w}_1 = +/c'; +l+\hat{w}_1$, so $r' = +/c'; +l$ and $w_1 = w_2$. (Note that we used the knowledge that $+\hat{}$ is injective here. The conclusion would be unwarranted if $+$ were commutative or idempotent.) We can now drop the subscripts on w. Let $c_T = Trnc\, c$. Then

$$c_T = Trnc\, c'+\hat{}\, l+\hat{w} = (l \neq 0; \rightarrow c'+\hat{l}) \,[]\, (l = 0; \rightarrow c'),$$

in which c' and l are still to be determined. We see that c' and l satisfy

$$(l \neq 0; \rightarrow c_T = c'+\hat{l}) \,[]\, (l = 0; \rightarrow c_T = c').$$

If $c_T = 0$, the first alternative cannot apply (since $c'+\hat{l} \neq 0$), so then $l = 0$. Otherwise, we can put $c_T = c'_T+\hat{l}_T$, and so

$$(c_T \neq 0; \wedge l \neq 0; \rightarrow \langle c', l\rangle = \langle c'_T, l_T\rangle) \,[]\, (l = 0; \rightarrow \langle c', l\rangle = \langle c_T, 0\rangle),$$

or

$$\langle c', l\rangle = (c_T \neq 0; \wedge l \neq 0; \rightarrow \langle c'_T, l_T\rangle) \,[]\, (l = 0; \rightarrow \langle c_T, 0\rangle).$$

The conditions on l have now lost their significance, since they are satisfied by both possible choices. If we put

$$c_1 = c'_T+\hat{}\, l_T+\hat{w}, \quad c_2 = c_T+\hat{w},$$

we find that $c = c'+\hat{}\, l+\hat{w}$ has to satisfy

$$c = (c_T \neq 0; \rightarrow c_1) \,[]\, c_2.$$

Since c has to satisfy $\uparrow/len*c \leqslant$ MAX, the first choice is open only if, moreover, $len\, l_T+\hat{w} \leqslant$ MAX, and the second one if $len\, \hat{w} = \#w \leqslant$ MAX. The remaining indeterminacy has to be resolved using the minimality of *lpos c*. If both choices are still open, c_1 has to be chosen, since

$$lpos\, c_1 = \langle \#c'_T, len\, l_T+\hat{w}\rangle < \langle 1+\#c'_T, len\, \hat{w}\rangle =$$
$$\langle \#c_T, len\, \hat{w}\rangle = lpos\, c_2.$$

The choice is now determinate, and $c = c_T \# w$, where $\#$ is defined by

$$0 \# w \Leftarrow \#w; \leqslant \text{MAX}; \rightarrow \hat{w};$$
$$c'_T+\hat{l}_T; \# w \Leftarrow (len\, l_T+\hat{w}; \leqslant \text{MAX}; \rightarrow c_1) \,[]$$
$$(len\, l_T+\hat{w}; > \text{MAX}; \wedge (\#w; \leqslant \text{MAX}); \rightarrow c_2).$$

It has to be verified next that $Trnc\ r \sim Trnc\ c$. In the first place,

$$
\begin{aligned}
&+/Trnc\ c = +/Trnc\ c' \hat{+} l \hat{+} \hat{w} = \\
&+/(l \neq 0 ; \rightarrow c' \hat{+} \hat{l}) \,[]\, (l = 0 ; \rightarrow c') = \\
&(l \neq 0 ; \rightarrow +/c' \hat{+} \hat{l}) \,[]\, (l = 0 ; \rightarrow +/c') = \\
&(l \neq 0 ; \rightarrow (+/c') + l) \,[]\, (l = 0 ; \rightarrow +/c') = +/c' ; +l = r' = \\
&Trnc\ r' \hat{+} \hat{w} = Trnc\ r .
\end{aligned}
$$

It is intuitively obvious that erasing words cannot increase line lengths, so that $\uparrow/len * c \leqslant \text{MAX}$ implies $\uparrow/len * Trnc\ c \leqslant \text{MAX}$. However, we will derive this also formally, just to show how this is done. We reinstate—temporarily—$len\ 0 = 0$. Then

$$
\begin{aligned}
&\uparrow/len * c' \hat{+} \hat{0} = \uparrow/(len * c') \hat{+} len\ 0 = \uparrow/(len * c') \hat{+} \hat{0} = \\
&\uparrow/len * c' ; \uparrow\uparrow/\hat{0} = \uparrow/len * c' ; \uparrow 0 = \uparrow/len * c' ; \uparrow\uparrow/0 = \uparrow/len * c' .
\end{aligned}
$$

So

$$
\begin{aligned}
&\uparrow/len * c = \uparrow/len * c' \hat{+} l \hat{+} \hat{w} = \uparrow/(len * c') \hat{+} len\ l \hat{+} \hat{w} = \\
&\uparrow/len * c' ; \uparrow len\ l \hat{+} \hat{w} \geqslant \uparrow/len * c' ; \uparrow len\ l = \\
&\uparrow/(len * c' ; \hat{+} len\ l) = \uparrow/len * c' \hat{+} \hat{l} = \\
&(l \neq 0 ; \rightarrow \uparrow/len * c' \hat{+} \hat{l}) \,[]\, (l = 0 ; \rightarrow \uparrow/len * c' \hat{+} \hat{0}) = \\
&(l \neq 0 ; \rightarrow \uparrow/len * c' \hat{+} \hat{l}) \,[]\, (l = 0 ; \rightarrow \uparrow/len * c') = \\
&\uparrow/len * (l \neq 0 ; \rightarrow c' \hat{+} \hat{l}) \,[]\, (l = 0 ; \rightarrow c') = \uparrow/len * Trnc\ c .
\end{aligned}
$$

We have now $Trnc\ r \sim Trnc\ c$.

Finally, it must be shown that replacing $Trnc\ c$ in $c = Trnc\ c ; \# w$ by an arbitrary realization of $f\ Trnc\ r$ does no harm to the minimality of $lpos\ c$. (The verification that the result still satisfies $r \sim c$ is straightforward and is omitted here.) If $Trnc\ r = 0$, there is no choice but taking $c = 0 \# w$. Otherwise, putting $c = c_T \# w = c'_T \hat{+} \hat{l}_T ; \# w$, we have

$$
\begin{aligned}
&lpos\ c = lpos\ c'_T \hat{+} \hat{l}_T ; \# w = \\
&lpos\ (len\ l_T \hat{+} \hat{w} ; \leqslant \text{MAX} ; \rightarrow c_1) \,[]\, (len\ l_T \hat{+} \hat{w} ; > \text{MAX} ; \rightarrow c_2) = \\
&(len\ l_T \hat{+} \hat{w} ; \leqslant \text{MAX} ; \rightarrow lpos\ c_1) \,[]\, (len\ l_T \hat{+} \hat{w} ; > \text{MAX} ; \rightarrow lpos\ c_2) .
\end{aligned}
$$

If we define

$$\langle m, n \rangle = lpos\ c_T ,$$

we find $\#c'_T = m$ and $len\ l_T = n$. Then

$$len\ l_T \hat{+} \hat{w} = len\ l_T ; +1 + \#w = n + 1 + \#w,$$

and so

$$
\begin{aligned}
&lpos\ c_1 = \langle \#c'_T, len\ l_T \hat{+} \hat{w} \rangle = \langle m, n + 1 + \#w \rangle ; \\
&lpos\ c_2 = \langle \#c_T, len\ \hat{w} \rangle = \langle \#c'_T \hat{+} \hat{l}_T, len\ \hat{w} \rangle = \\
&\langle \#c'_T ; +1, len\ \hat{w} \rangle = \langle m + 1, \#w \rangle .
\end{aligned}
$$

We can now simplify the expression for $lpos\ c$ to

$$
\begin{aligned}
&(n + 1 + \#w ; \leqslant \text{MAX} ; \rightarrow \langle m, n + 1 + \#w \rangle) \,[] \\
&\qquad\qquad (n + 1 + \#w ; > \text{MAX} ; \rightarrow \langle m + 1, \#w \rangle).
\end{aligned}
$$

This expression is non-strictly monotonic in $\langle m, n\rangle = lpos\, c_T$, so taking c_T to be a realization of $f\,Trnc\,r$, which minimizes *lpos*, guarantees that *lpos* c is minimized too. Summing up, we have

$$f0 = 0;$$

$$fr +\!\!+ \hat{w} = Trnc\, fr +\!\!+ \hat{w};\ +\!\!+ w \Rightarrow f\,Trnc\, r +\!\!+ \hat{w};\ +\!\!+ w = fr;\ +\!\!+ w.$$

After these lengthy preparations (but remember that most of the derivations were aimed at exhibiting obvious facts), we can now formulate an "implementation" of f:

$$ff\,0 \Leftarrow 0;$$

$$ff\,r +\!\!+ \hat{w} \Leftarrow ff\,r;\ +\!\!+ w.$$

This function satisfies $f \Rightarrow ff$ and it preserves the definedness of f; i.e., if $fr \neq \square/0$, then $ff\,r \neq \square/0$. The standard technique of recursion elimination gives the obvious iterative "eager" algorithm. Note also that $fr = \square/0$ implies $ff\,r = \square/0$. This is a consequence of $f \Rightarrow ff$, since then $\square/0 \Rightarrow fr \Rightarrow ff\,r \Rightarrow \square/0$. It is easy to define a total variant of ff by making $+\!\!+$ total, e.g. by removing the conditions "$\#w; \leqslant$ MAX" from its definition.

Some final remarks to this example: The length of the derivation is mainly due to the small steps taken, but also to some degree to the presentation, which emphasized the algorithmic analysis and synthesis. If one were to "guess" the definition of ff, then the verification is somewhat shorter. Note, in particular, that the need to handle $\mathbb{U}$ did not arise.

The final development phase was an example of "Formal Differentiation" (or "Finite Differencing") (PAIGE[23], PAIGE and KOENIG[24]). This term stands for a widely applicable technique for improving algorithms. It is of special interest here because it is often especially fit to the improvement of high-level algorithms that have been (semi-)automatically synthesized. The essential idea is that of "incremental" computation. Let x' be the result of applying a "small" variation to x. For many functions f, it is more efficient to compute the value of fx' from the result of fx and the variation, than to compute it afresh. It can be seen that this is a special case of the "Divide and Rule" paradigm. If x is the result of sequentially making small variations, then fx can also be computed sequentially. A challenging problem, not addressed here, is to develop general algebraic techniques for *deriving* expressions for "formal derivatives". For a not very general but interesting algebraic technique, see SHARIR[26].

The eager strategy (also known as "greedy" strategy) is a special case of formal differentiation in the context of optimization problems. A higher-level derivation would have run, schematically: (i) show that f satisfies the conditions of some "eagerness" theorem; (ii) apply the theorem to give ff as implementation. There appears to be a relationship with matroid theory here (KORTE and LOVÁSZ[16]). It remains to be investigated if this can be expressed conveniently in the framework pursued here. If so, it would be a good example of the "higher-level" theorems aimed at. A different choice for

the objective function (e.g., minimize the sum of the squares of the white space on each line) would have invalidated its applicability. Still, an important gain in efficiency is possible for many other objective functions (e.g., for the least-squares objective), namely by applying the technique of dynamic programming. An algebraic approach to this technique can be found in CUNINGHAME-GREEN[7], and a specific application of this approach in an algorithmic development in MEERTENS and VAN VLIET[20].

12. SECOND EXAMPLE: THE AMOEBA FIGHT SHOW

The following problem is of interest because it is the first problem that I tried to tackle algebraically without already knowing a reasonable algorithm for it—or seeing one immediately. It was passed on to me by Richard Bird. Its origin is, as far as I know, a qualifying exam question from CMU. Since I do not know the original formulation of the problem, it is given here in a setting of my own devising.

What with the rising prices of poultry, a certain showman has modernized his *Amazing Life-and-Death Rooster Fight Show*, and replaced his run of prize-fighting cocks by a barrel of cannibalistic amoebae. As is well known, amoebae have an engrossing way of tackling an opponent: it is simply swallowed, hide and hair.[1] It follows from the Law of Conservation of Mass that the weight of the winner then increases by that of the loser. Each show stages a tournament between n amoebae (where n is some positive natural number), consisting of a sequence of $n-1$ duels (two amoebae staged against each other). At the end of the tournament, all that remains is the final victor (although it encompasses, in some sense, all losers). The showman wishes to maximize the throughput of his enterprise by minimizing the time taken by one show. The time needed for a single duel, he has found experimentally, is proportional to the weight of the lighter contestant (about one minute for each picogram). At the start of a show, the amoebae are lined up in a microscopic furrow. Each two adjacent fighters are kept apart by a removable partition. (This set-up has been chosen thus because of limitations in the state of the art of micro-manipulation. For similar reasons, the initial arrangement cannot be controlled.) Each time a partition is removed, the two amoebae now confronting each other engage in a life-and-death duel.

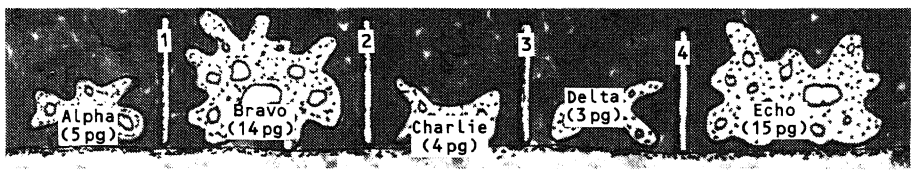


FIGURE 3. Five amoebae lined up before the tournament (magnification: 500 ×)

1. For amoebae, this terminology is not entirely appropriate. The hapless victim is, in fact, engulfed by the attacker's bulging around and completely enveloping it, *membrane* and *pseudopodia*.

The showman thinks the best strategy is to have, each time, the lightest amoeba fight against its heaviest neighbour. His assistant suspects that it is better to choose the pair whose weight difference is largest. In the situation sketched in figure 3, these two strategies give rise to the same sequence of duels. First, the showman removes partition 4, and Delta and Echo fight. After 3 minutes, Echo has consumed Delta. Next, partition 3 is lifted, and Charlie enters the arena against Echo. The unequal battle takes 4 more minutes. Echo weighs now, after having feasted on Delta and Charlie, $15+3+4 = 22$ picograms. The next step is the removal of partition 1. It takes Bravo 5 minutes to gobble up Alpha. When the last partition is taken away, the battle of the champions starts. In spite of Bravo's putting up a heroic resistance, pseudopod after pseudopod wraps around its body, and after 19 exciting minutes the last visible part disappears into Echo's innards. The whole tournament has taken $3+4+5+19 = 31$ minutes. Unaware of the fact that a different sequence of duels would have required less than half an hour, the showman and his assistant start clearing the house for the next show.

Let us see if we can do better. The process of amoeba fusion in a tournament creates a tree structure on top of the original sequence of amoebae. For the example, that tree is $\hat{A}+\hat{B}; +\hat{C}+\hat{D}+\hat{E}$, where A stands for Alpha, etc. Each node corresponds to a sub-tournament. Since the structure of the tree gives sufficient information to determine the tournament, even if the elements are not amoebae, it is simplest to work directly with the sequence of the *weights* of the amoebae. Let $w\,t$, for a given tournament tree t, stand for the final weight of the champion of t, $d\,t$ for its duration, and $wd\,t$ for the pair $\langle w\,t,\ d\,t\rangle$. For the trivial case of a one-amoeba "tournament" we have

$$wd\,\hat{w} = wd_0\,w \Leftarrow \langle w,\ 0\rangle .$$

Then we find

$$wd\,t_L+t_R = wd\,t_L; \times wd\,t_R ,$$

where the operation $\times$ is given by

$$\langle w_L ,\ d_L\rangle \times \langle w_R ,\ d_R\rangle \Leftarrow \langle w_L+w_R ,\ d_L+d_R+w_L\downarrow w_R\rangle .$$

(The operation $\times$ is commutative, but, of course, not associative.) So, by the homomorphism lemma, we can express wd by

$$wd = \times/\,wd_0\,\ast .$$

The function d can be re-defined as $\pi_2\ wd$. If $T\,s$ is the set of all possible tournament trees that can be put on top of an initial configuration s, the problem can be specified as: Determine $\downarrow_d/\ T\,s$. The property characterizing a member t of $T\,s$ is $s = +/\hat{}\,\ast t$, in which the inserted operation $+$ introduces associativity. Then

$$T\,s \Leftarrow (s = +/\hat{}\,\ast)\colon \mathbb{U} .$$

It would be possible, of course, to develop an algorithm for determining T, after which we would have an algorithm for the whole problem. But

computing Ts for large values of $\#s$ is very inefficient; the number of binary trees with n endpoints is of the order $\Omega(4^n n^{-3/2})$. It will turn out, moreover, that we do not need an explicit construction of Ts in the derivation. It is also obvious that dynamic programming gives us a polynomial algorithm. In such cases it is generally easy to transform an algorithm for a function of the form $\downarrow\!/ f*$ $(= f\downarrow_f\!/)$ to an algorithm for $\downarrow_f\!/$. Therefore, we concentrate first on simplifying $\downarrow\!/ d* T$.

Let us first try some simple cases. In minimization problems such as the present one, it often pays off to switch to a seemingly more conventional algebraic notation that exploits the algebraic properties of the two operations $\downarrow$ and $+$ (CUNINGHAME-GREEN [7]). For not only are both associative and commutative, but together they are also distributive: $x+y\downarrow z = x+y;\downarrow x+z$. If we denote the operation $+$ the way a *multiplicative* operator is usually written in mathematical formulae, namely by juxtaposition of its operands (so we write "xy" instead of "$x+y$"), and we use then the—now free—symbol "$+$" to denote the operation $\downarrow$, then the distributive property referred to above is written as $x(y+z) = xy+xz$, in which "multiplication" takes precedence over "addition". This is purely a notational convention, but the advantage is that we can apply our experience in handling and simplifying formulae of this kind. Unconventional identities, however, are $x0 = 0x = x$ (since the *meaning* is still addition) and $x+0 = 0+x = 0$ (in which it is assumed that all numbers involved are non-negative; a property preserved by the two operations). So we have, in particular, $x+xy = x0+xy = x(0+y) = x0 = x$: a term cancels other terms of which it is a factor. The special case $x+x = x$ of the identity $x+xy = x$ expresses the fact (which we knew already, of course) that the operation $+$ is idempotent. The expression for $\times$ in this new notation becomes now:

$$\langle w_L, d_L\rangle \times \langle w_R, d_R\rangle = \langle w_L w_R, d_L d_R(w_L+w_R)\rangle .$$

If the initial amoeba weight configuration is $\hat{w}_1$, the duration of the (trivial) tournament is, of course, 0. For a configuration $s = \hat{w}_1+\hat{w}_2$, the only member of Ts is $\hat{w}_1+\hat{w}_2$, and we find a duration of w_1+w_2. For a configuration $s = \hat{w}_1+\hat{w}_2+\hat{w}_3$, the set Ts contains two trees: $t_1 = (\hat{w}_1+\hat{w}_2)+\hat{w}_3$ and $t_2 = \hat{w}_1+(\hat{w}_2+\hat{w}_3)$. By computing $\pi_2 \times\!/ wd_0 *$ for t_1 and t_2, we find $d\,t_1 = (w_1+w_2)(w_1w_2+w_3)$ and $d\,t_2 = (w_2+w_3)(w_1+w_2w_3)$. So the shortest tournament takes time $(w_1+w_2)(w_1w_2+w_3)+(w_2+w_3)(w_1+w_2w_3)$. After distribution, we obtain the formula

$$w_1^2w_2+w_1w_3+w_1w_2^2+w_2w_3+w_1w_2+w_2^2w_3+w_1w_3+w_2w_3^2.$$

This simplifies to $w_1w_2+w_1w_3+w_2w_3$. We see a pattern emerging: the next formula should be $w_1w_2w_3+w_1w_2w_4+w_1w_3w_4+w_2w_3w_4$. The hypothesis is that we obtain, for a general configuration of n weights, the "sum" of all "products" of the members of each subset of size $n-1$ of the set of amoebae. First, we return to the notation using "$+$" for addition, and "$\downarrow$" for taking the minimum. An expression like $(w_1+w_2)\downarrow(w_1+w_3)\downarrow(w_2+w_3)$ can be rewritten thus:

$$(w_1+w_2)\downarrow(w_1+w_3)\downarrow(w_2+w_3) =$$
$$(w_1+w_2+w_3;-w_3)\downarrow(w_1+w_2+w_3;-w_2)\downarrow(w_1+w_2+w_3;-w_1) =$$
$$w_1+w_2+w_3;-w_1\uparrow w_2\uparrow w_3 .$$

In the general case, we expect to find

$$\downarrow/ d*Ts = +/s;-\uparrow/s .$$

A moment's reflection will show why this is a lower bound for the duration of any tournament on s. For in a tournament, each contestant but one is eaten, and its weight is then counted at least once. So the best possible is that each weight of the less fortunate contestants is counted exactly once, and that the one contestant not counted is as heavy as they come. The next question is if we can prove that this formula is correct (and not only a lower bound) for the general case. For this, we do not need the full-fledged expression for Ts, but only a simple property:

> The tree $t_L+t_R \in Ts$ if and only if there exist configurations s_L and s_R such that $s = s_L+s_R$, $t_L \in Ts_L$ and $t_R \in Ts_R$.

First we prove, by induction, that we have indeed a lower bound. Let $t = \downarrow_d/Ts = t_L+t_R$, and so (by the induction hypothesis) $d\,t_L \geqslant +/s_L;-m_L$ and $d\,t_R \geqslant +/s_R;-m_R$, where $s_i = +/\hat{}*t_i$ and $m_i = \uparrow/s_i$ for $i = L, R$. Then

$$d\,t = (+/s_L;-m_L)+(+/s_R;-m_R)+(+/s_L;\downarrow+/s_R) =$$
$$+/s;-m_L+m_R;+(+/s_L;\downarrow+/s_R) \geqslant$$
$$+/s;-m_L+m_R;+m_L\downarrow m_R = +/s;-m_L\uparrow m_R = +/s;-\uparrow/s.$$

Next, we must show that this lower bound is attainable (which is trivial for a single amoeba). The method is again by induction. Write $s = \hat{w}_1+s'+\hat{w}_n$. If we take for t_R a d-minimizing member of $Ts'+\hat{w}_n$, we find for $d\,\hat{w}_1+t_R$, by using the hypothesized formula for $d\,t_R$, the expression

$$w_1+(+/s'+\hat{w}_n;-\uparrow/s'+\hat{w}_n) = +/s;-\uparrow/s'+\hat{w}_n .$$

Similarly, taking $t_L = \downarrow_d/T\hat{w}_1+s'$, we find

$$d\,t_L+\hat{w}_n = (+/\hat{w}_1+s';-\uparrow/\hat{w}_1+s')+w_n = +/s;-\uparrow/\hat{w}_1+s'.$$

So

$$\downarrow/d*Ts \leqslant d\,\hat{w}_1+t_R;\downarrow d\,t_L+\hat{w}_n =$$
$$(+/s;-\uparrow/s'+\hat{w}_n)\downarrow(+/s;-\uparrow/\hat{w}_1+s') =$$
$$+/s;-(\uparrow/s'+\hat{w}_n;\uparrow\uparrow/\hat{w}_1+s') = +/s;-\uparrow/s.$$

The proof shows that it is possible to organize the tournament such that (a) an amoeba of (initially) maximum weight will emerge as champion and (b) the loser of each duel is putting up its first appearance (and so is not burdened by the weight of any fellow amoebae it has devoured). It follows immediately from (a) and (b) that each amoeba, except the one destined to be champion, enters the stage only against the future champion. Conversely, it is now

obvious that any tournament with this property is optimal. The step from here to a linear-time algorithm is simple, if not trivial. One possible algorithmic formulation is

$$\downarrow_d / \; T\, s \Rightarrow t\, s\,,$$

where t is defined recursively by

$$t\, \hat{w} \Leftarrow \hat{w}\,;$$

$$t\, \hat{w}_1 + s' + \hat{w}_n \Leftarrow (w_1 \leqslant m_R;\ \rightarrow \hat{w}_1 + t\, R)\ \square\ (w_n \leqslant m_L;\ \rightarrow t\, L;\ + \hat{w}_n),$$
$$\text{where } L = \hat{w}_1 + s',\ \ m_L = \uparrow / L,\ \ R = s' + \hat{w}_n,\ \ m_R = \uparrow / R.$$

The correctness follows directly from the preceding proof, since it has been shown that $d\, t\, s = \downarrow / d * T\, s$.

Our showman is probably more interested in a simple method that tells him when to lift which partition, than in determining a tree. It should be obvious that we can advise him to remove, each time, any partition keeping the heaviest amoeba apart from a neighbour. It is not hard to derive this formally from the given expression for t.

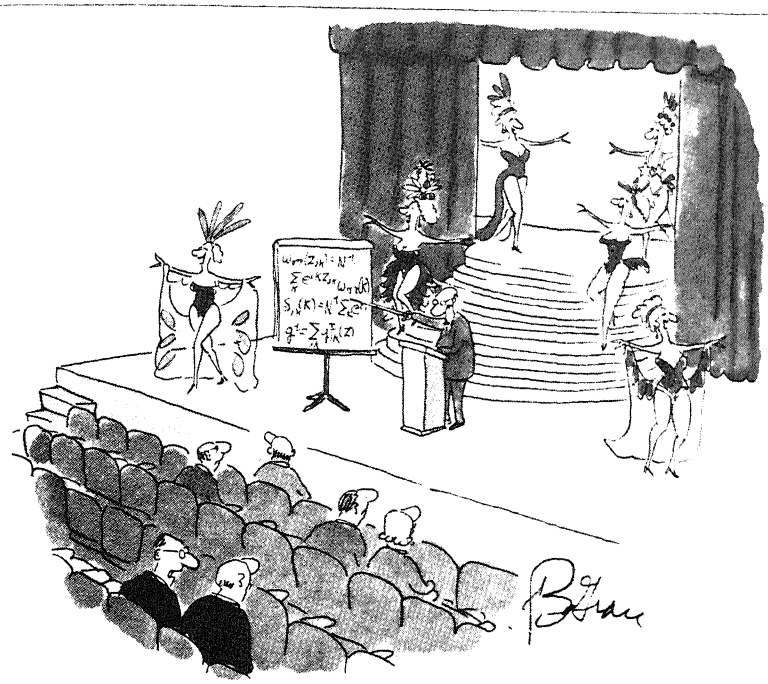
13. Conclusion

An attempt has been made here to convince the reader that the ideal of a discipline of "Algorithmics" can be realized. If the account was possibly unconvincing, then, I suspect, a major culprit is perhaps the shock of being exposed to a set of unfamiliar squiggles. In my first endeavours, exploring the suggestions of Bird [4], I found that the only way to proceed was to translate the formulae continually into familiar "operational" concepts. Now, after having played with these notations for some time, I find myself applying transformations without being conscious of an operational meaning. The reader is invited to try and undergo the same experience. A good starting point is to derive

$$\#P : +/ \;=\; +/ + / (\iota \ll * P\,?) * *.$$

This is a meaningful and useful transformation; the two formulae are readily translated into "pidgin Algol", and the resulting programs are each about 10 lines long.

Much work has to be done to develop the current set of concepts and notations beyond the initial attempts presented here. Important points are the discovery and formulation of "algebraic" versions of higher-level programming paradigms and strategies, and the development of techniques to assess something like the concrete "complexity" of an expression in the absence of an operational model in which time and space are meaningful notions. Other issues to be investigated are the introduction of infinite objects, of ways to express some form of concurrency, and of suitable notations for handling algebraically more complex structures than the ones dealt with here.



"Personally, I think he's trying to cover up a shaky theory."

ACKNOWLEDGEMENTS

The cartoon by Bud Grace, Copyright © 1984, B. Grace, is reprinted here by the kind permission of the artist. I am indebted to Steven Pemberton of CWI and to Norman Shulman of NYU for scrutinizing earlier versions and suggesting many improvements.

REFERENCES

1. J. BACKUS (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM 21*, 613–641.
2. F. L. BAUER *et al.* (1981). Programming in a wide-spectrum language: a collection of examples. *Science of Computer Programming 1*, 73–114.
3. R. S. BIRD (1977). Improving programs by the introduction of recursion. *Comm. ACM 20*, 151–155.
4. R. S. BIRD (1981). *Some Notational Suggestions for Transformational Programming.* WG 2.1 working paper NIJ-3 (unpublished).
5. H. J. BOOM (1981). *Further Thoughts on Abstracto.* WG 2.1 working paper ELC-9 (unpublished).
6. R. M. BURSTALL, J. DARLINGTON (1977). A transformation system for developing recursive programs. *J. ACM 24*, 44–67.
7. R. CUNINGHAME-GREEN (1979). *Minimax Algebra. Lecture Notes in Economics & Mathematical Systems 166*, Springer, Berlin, 1979.
8. E. W. DIJKSTRA (1968). A constructive approach to the problem of program correctness. *BIT 8*, 174–186.

9. E. W. DIJKSTRA (1971). Notes on structured programming. O.-J. DAHL, E. W. DIJKSTRA, C. A. R. HOARE. *Structured Programming,* Academic Press.
10. R. W. FLOYD (1967). Assigning meanings to programs. J. T. SCHWARTZ (ed.). *Proc. Symp. Appl. Math., Vol. 19, Mathematical Aspects of Comp. Science* 19–32, AMS, Providence, RI.
11. L. GEURTS, L. MEERTENS (1978). Remarks on Abstracto. *ALGOL Bull. 42,* 56–63.
12. J. GUTTAG, J. HORNING, J. WILLIAMS (1981). FP with data abstraction and strong typing. *Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture* 11–24, ACM.
13. F. W. VON HENKE (1976). An algebraic approach to data types, program verification, and program synthesis. *Proc. Math. Foundations of Comp. Science '76, Lecture Notes in Comp. Science 45,* 330–336, Springer, Berlin.
14. C. A. R. HOARE (1969). An axiomatic basis for programming language constructs. *Comm. ACM 12,* 576–580.
15. M. A. JACKSON (1975). *Principles of Program Design.* A.P.I.C. Studies in Data Processing 12, Academic Press.
16. B. KORTE, L. LOVÁSZ (1981). Mathematical structures underlying greedy algorithms. F. GÉCSEG (ed.). *Fundamentals of Computation Theory, Lecture Notes in Comp. Science 117,* 205–209, Springer, Berlin.
17. J. MCCARTHY (1963). A basis for a mathematical theory of computation. P. BRAFFORT, D. HIRSCHBERG (eds.). *Computer Programming and Formal Systems* 33–70, North-Holland.
18. L. MEERTENS (1977). From abstract variable to concrete representation. S. A. SCHUMAN (ed.). *New Directions in Algorithmic Languages 1976* 107–133, IRIA, Rocquencourt.
19. L. MEERTENS (1979). Abstracto 84: the next generation. *Proc. of the 1979 Annual Conf.* 33–39, ACM.
20. L. MEERTENS, J. C. VAN VLIET (1976). Repairing the parenthesis skeleton of ALGOL 68 programs: proof of correctness. G. E. HEDRICK (ed.). *Proc. of the 1975 Int. Conf. on ALGOL 68* 99–117, Oklahoma State University, Stillwater.
21. P. NAUR (1966). Proof of algorithms by general snapshots. *BIT 6,* 310–316.
22. P. NAUR (1969). Programming by action clusters. *BIT 9,* 250–258.
23. R. PAIGE (1981). *Formal Differentiation.* UMI Research Press.
24. R. PAIGE, S. KOENIG (1982). Finite differencing of computable expressions. *ACM Trans. on Programming Languages and Systems 4,* 402–454.
25. A. RALSTON, M. SHAW (1980). Curriculum '78—Is Computer Science really that unmathematical? *Comm. ACM 23,* 67–70.
26. M. SHARIR (1982). Some observations concerning formal differentiation of set theoretic expressions. *ACM Trans. on Programming Languages and Systems 4,* 196–225.
27. N. WIRTH (1971). Program development by stepwise refinement. *Comm. ACM 14,* 221–227.
28. N. WIRTH (1973). *Systematic Programming.* Prentice-Hall.