Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

L.G.L.T. Meertens

An Abstracto reader
prepared for IFIP WG 2.1

Computer Science/Department of Algorithmics & Architecture Note CS-N8702 April

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

COVER LETTER

"The thing can be done," said the Butcher, "I think
  The thing must be done, I am sure.
The thing shall be done! Bring me paper and ink,
  The best there is time to procure."

The purpose of this reader is to provide WG 2.1 members with a convenient overview of my papers devoted to the Abstracto theme.

Not only have I included the papers that employ a notation that has facetiously been called "SQUIGOL", but also two earlier papers that, although set in a quite different framework, shed some more light on the purpose of the whole project, and also show something of the scientific trek I have made before I came to the point where I am now. Between the papers I have inserted some comments made in hindsight. I did not include any of the (sizable) stack of presentations prepared for WG 2.1 meetings that only exist in the form of overhead sheets, without connecting text.

For the WG 2.1 working documents I have tried to adapt the usage of symbols to my current practice. The papers that have been published I have left untouched.

The order in which the papers appear here is the chronological order in which they were written. A better reading order may be:

-- "Remarks on Abstracto";

-- "Abstracto 84", Sections 1 - 4;

-- "Algorithmics", Sections 1 - 3;

-- "Two exercises";

-- "A Common Basis";

-- "Algorithmics", remaining Sections;

-- "Some more examples".

Amsterdam, April 1987 Lambert Meertens

Note CS-N8702
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

An Abstracto Reader

prepared for IFIP WG 2.1

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

CONTENTS

REMARKS ON ABSTRACTO

He had bought a large map representing the sea,
Without the least vestige of land:
And the crew were much pleased when they found it to be
A map they could all understand.

The first paper of this reader was prepared for the Oxford meeting of WG 2.1 in December 1977. The term "Abstracto" must have been introduced earlier in WG 2.1 circles, since there were two other presentations at the meeting with that word in their titles.

One reason that I find this paper interesting is that it already implicitly identifies the "five unmistakable marks" by which we may know the genuine article if we are to happen upon it during our explorations:

1. "Program transformation" is viewed as a mathematical activity of manipulating algorithmic expressions, and a "transformation" as nothing but the application of a theorem.

2. Abstracto is an open language, not developed with the aim of being able to use a mechanical system.

3. No sharp distinction is made between an "algorithm" and a "problem specification", and executability (called "implementability" here) is not required.

4. The real issue is the development of high-level concepts and notations.

5. There is already a clear emphasis on "iterators", mirrored by the current emphasis on homomorphisms, and even a genuine reduction (the "OPT ... TPO" construction).

It is also interesting that a "textbook for an advanced course on algorithmics" is mentioned, an application area that has been giving guidance to my thoughts until this day.

Also interesting to me is to look back at these first attempts to give some concrete form to Abstracto. Reading this now gives me a feeling of compassion for these authors who are pathetically groping around in pitch darkness. It makes me wonder what I will think if I read my latest writings ten years from now.

A final point of interest is the appearance of this Bird character, who is quoted with agreement several times (and again in the next few papers). It is clear that I must have recognized a kindred spirit in him, but little did I suspect our fruitful future collaboration. I was to meet Richard only four years later, at the Nijmegen meeting in 1981.

ALGOL Bull. 42 (1978), 56-63.

REMARKS ON ABSTRACTO*

Leo Geurts
Lambert Meertens
Mathematisch Centrum, Amsterdam

1. ABSTRACTO LIVES

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

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

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

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

---

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

without the deleterious burden of irrelevant detail.

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

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

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

2. THE LANGUAGE OF MATHEMATICS

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

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

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

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

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

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

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

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

3. IN SEARCH OF ABSTRACTO 84

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

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

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

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

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

is solved by

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

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

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

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

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

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

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

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

(corresponding to $p \Rightarrow q;\ q \Rightarrow r$), and the proof rule

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

(corresponding to IF $p1 \rightarrow p1 \Rightarrow q1$ [] $p2 \rightarrow p2 \Rightarrow q2$ FI). By turning a derivation of $p \Rightarrow q$ upside down, a solution is constructed. Unfortunately, there is no suitable rule for a solution of the form

$$\text{DO } b \rightarrow p \wedge b \Rightarrow p \text{ OD}.$$

(The rule

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

does not express termination and allows the derivation of $p \Rightarrow p \wedge \neg b$ for arbitrary p and b.)

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

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

may be expressed more clearly by

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

one might wish to express

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

as

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

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

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

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

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

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

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

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

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

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

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

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

4. GLIMPSES OF ABSTRACTO 84

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

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

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

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

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

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

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

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

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

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

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

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

records, as in

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

(disjoint) unions, as in

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

PASCAL scalars, as in

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

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

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

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

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

5. A POSSIBLE PITFALL

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

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

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

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

REFERENCES

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

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

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

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

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

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

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

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

ABSTRACTO 84

> The last of the crew needs especial remark,
> Though he looked an incredible dunce:
> He had just one idea--but, that one being "Snark,"
> The good Bellman engaged him at once.

The following paper was written during a stay at New York University in 1979. It is not a paper I am particularly fond of, but it represents a necessary stage I had to go through in my quest. Also, the objectives of Abstracto are formulated here more clearly (in Sections 3 and 4) than in the first paper.

One specific reason why I do not like the paper is that the technical part is full of bugs. I have appended a list of errata, just in case.

Proc. of the 1979 Annual Conf., 33-39,
ACM, Detroit (1979)

ABSTRACTO 84: THE NEXT GENERATION

Lambert Meertens
Mathematical Centre
Amsterdam

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

1. THE ABSTRACTO PROJECT

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

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

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

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

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

2. ABSTRACTO AS A PIDGIN

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

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

Viewed in this perspective, the Abstracto effort is aimed at speeding up evolution by proposing and using suitable notations for important algorithmic concepts. Of course, it will be possible

(and maybe desirable) to take a snapshot of Abstracto at regular intervals, to clean up the picture and to present it as, say, Abstracto 84. But this will not stop Abstracto from evolving on.

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

3. ABSTRACTO AND TRANSFORMATIONAL PROGRAMMING

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

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

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

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

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

4. DESIGNING ABSTRACTO 84

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

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

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

```
z:=1 ; x:=2 ; z:=z·x
```

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

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

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

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

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

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

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

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

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

$$*(b \rightarrow S)$$

is the same as that of

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

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

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

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

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

$$z := iz$$

using $z = x + iy$.

5. ABSTRACT ALGORITHMIC EXPRESSIONS AND REFINEMENT

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

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

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

able". By that act, EA grows. Thus, the experience gained by using Abstracto may serve as a guideline for the development of programming languages.

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

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

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

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

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

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

for all p and q, if {p}S′{q}, then {p}S{q}.

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

for all q, wp(S,q) implies wp(S′,q).

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

$S \leq S$;
if $S \leq S'$ and $S' \leq S''$, then $S \leq S''$.

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

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

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

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

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

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

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

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

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

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

for all q, wp′(S,q) implies wp′(S′,q).

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

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

Take for wp* any predicate transformer satisfying

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

Any wp* thus defined satisfies

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

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

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

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

It may appear that cr also has to satisfy

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

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

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

This is indeed the case, as is easily verified.

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

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

then it is straightforward to verify that

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

is acceptable as definition. Similarly, one can take

$$cr(b_1 \to S_1 \;[]\; b_2 \to S_2) = (b_1 \supset cr(S_1)) \;\&\; (b_2 \supset cr(S_2))$$

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

6. EXAMPLES OF ABSTRACT ALGORITHMIC EXPRESSIONS

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

$$A[v:=u]$$

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

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

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

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

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

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

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

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

This has many presumably unintended solutions, like

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

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

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

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

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

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

An example of a problem expression is

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

This could be realized by the concrete expression

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

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

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

Clearly, we may take

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

and

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

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

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

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

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

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

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

$$S|v:p,$$

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

$$y:=v \mid v:\ x\geq 0 \supset (v^2=x \ \&\ v\geq 0).$$

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

$$wp(S|v:p,q) = (\exists v:\ p)\ \&\ (\forall v:\ p \supset wp(S,q)).$$

We may take

$$cr(S|v:p) = \exists v:\ p$$

and

$$wp^*(S|v:p,q) = \forall v:\ p \supset wp(S,q).$$

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

(e) $v:=[p\Rightarrow q] =^* v:=v' \mid v':\ p \supset q[v:=v']$, where v′ is a list of fresh variables of the same length as v;
(f) $v:=[p \Rightarrow p\&\neg b] =^*$
$*(b \rightarrow v:=v' \mid v':$
$p\&b \supset p[v:=v']\ \&\ 0(v') < 0(v))$,
where v′ is again a list of fresh variables of the proper length, and 0 is a mapping from objects of the type of v to the elements of some well-ordered set (e.g., the ordinals), which may be chosen freely;
(g) If p′ implies p, then $S|v:p \leq^* S|v,v':p'$, where v′ is a (possibly empty) list of fresh variables;
(h) $S|v:p \leq^* S[v:=u]\,|v':p[v:=u]$, where u is a list of units of the proper length and v′ is a list of variables that are either fresh or an element of v, sufficiently large to bind all variables of v that remain present after the step;
(i) $S|\varepsilon:\underline{true} \leq^* S$ (where ε stands for the empty list).

Rules (e) and (f) allow the elimination of problem expressions. If the variant definition hinted at above is adopted, we would only have refinement in one direction. Rule (f) is probably the most powerful one in practice. It corresponds to rules in other proof systems that cover the WHILE loop. The mapping 0 ensures termination. It can be shown that mapping to the natural numbers (the initial segment of the ordinals) gives the same power, but at the cost of introducing mappings that are sometimes much more complicated than necessary (cf. Boom[3]). In (g) we find another application of the rule of consequence. It might have been combined with (h); for the sake of simplicity, this has not been done. Rule (h) is also quite powerful. By application of this rule one may arrive at (i), where the bound expression is eliminated. One has to go through this rule once for each abstract expression introduced.

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

7. AN EXAMPLE

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

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

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

This problem can be specified by the abstract expression

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

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

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

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

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

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

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

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

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

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

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

Using (g), this may again be refined to

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

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

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

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

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

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

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

Acknowledgements

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

REFERENCES

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

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

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

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

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

ERRATA to ABSTRACTO 84

(L stands for left and R for right column.)

p.36L: In

> for all p and q, if {p}S′{q}, then {p}S{q}

swap "S′" and "S".

p.36R: It should be mentioned that $\leq'$ does not necessarily preserve monotonicity. (Counterexample: let the set of abstract expressions be the closure under ";" of {Up, Dn, Sk}, with the concrete expressions being those not containing Dn. The semantics are given by

$$\begin{aligned} wp(Up, q) &= q[\![i:=i+1]\!], \\ wp(Dn, q) &= q[\![i:=i-1]\!], \\ wp(Sk, q) &= q. \end{aligned}$$

Then we find

$$\begin{aligned} wp'(Up, q) &= wp(Up, q), \\ wp'(Dn, q) &= TRUE, \\ wp'(Sk, q) &= wp(Sk, q). \end{aligned}$$

So Up $\leq'$ Dn. Now take f(S) = (S; Up). Then $wp'(f(Up), q) = q[\![i:=i+2]\!]$ and $wp'(f(Dn), q) = q$. So $f(Up) \not\leq' f(Dn)$.)

It should also have been pointed out that wp′ does not fully satisfy Dijkstra's healthiness criteria, since the Law of the Excluded Miracle does not hold--on purpose--for "unconcretizable" expressions (like Dn above).

p.37L: The statement (near the top) "wp*(S,q) implies wp′(S,q)" is wrong. (Counterexample: let the set of abstract expressions be the closure under ";" of {Ab, Ho}, and let the concrete expressions be those not containing Ab. The semantics are given by

$$\begin{aligned} wp(Ab, q) &= FALSE, \\ wp(Ho, q) &= FALSE. \end{aligned}$$

Then we find (since Ab $\leq$ Ho)

$$\begin{aligned} wp'(Ab, q) &= FALSE, \\ wp'(Ho, q) &= FALSE. \end{aligned}$$

Take cr(Ab) = FALSE, cr(Ho) = TRUE, and

$$\begin{aligned} wp^*(Ab, q) &= q, \\ wp^*(Ho, q) &= FALSE. \end{aligned}$$

Then wp*(Ab, TRUE) does not imply wp'(Ab, TRUE).)

Idem: At the end of Section 5, the "and so on" is too optimistic; the choice for cr(*(b→S)) is not obvious (the predicate must hold whenever *(b→S) would terminate).

p.38L: At the bottom of the previous page, the "Clearly we may take..." is by itself correct, but the choice for wp* here is not compatible with (d) and (e) later on. The property (d) can be possibly be saved by taking

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

but "=*" in (e) cannot hold then.

p.38R: The remark concerning O: "It can be shown that mapping to the natural numbers ..." is wrong. Just consider computing in the domain of ordinals.

Idem: In the third line from the bottom, delete "of Lemma 1".

p.39L: At about the middle, read "(h)" instead of "(d) of Lemma 2".

"What's the good of Mercator's North Poles and Equators,
Tropics, Zones, and Meridian Lines?"
So the Bellman would cry: and the crew would reply
"They are merely conventional signs!"

Using the framework sketched in the previous paper, I did most of the examples from the problem sets prepared for the Brussels meeting of WG 2.1 in December 1979 and the meeting in Wheeling, West Virginia, in August 1980.

On the whole, I was reasonably successful, but I nevertheless abandoned the approach. The reasons for that are set forth in a note I wrote to myself at about that time, and that I cannot resist quoting in length. After mentioning some technical problems I ran across, I continue:

> Most of these shortcomings can be overcome in an ad-hoc manner, and this might possibly lead to insight how to extend the framework. However, in using my approach it has gradually dawned upon me that there is a much more basic shortcoming, not so much of a technical, but more of a methodological character.
>
> I tried to apply my "method" to the set of Brussels examples, and found that it did, on the whole of it, reasonably well (admittedly using some ad-hoc tricks). In fact, it turned out much more applicable than I had expected. Paradoxically, this very success made me suspicious. My attempts to understand the situation led to a strong dissatisfaction with the basic approach. Although I still feel that the framework has an area of applicability in which it is valuable, and further investigations in this direction might be worthwile, I started thinking in a completely different direction. To explain the dissatisfaction, I have to go into some detail. An expression (think of it as "statement") in the 1979 framework is either basic, or composed of other expressions. The composition methods are "sequencing" (expressed, conventionally, with a semicolon operator), "conditional choice" and "iteration". The basic expressions are assignment and two "abstract" expressions: the "problem expression" and the "bound expression". For following the argument, it is sufficient to know that the abstract expressions allow a great deal of indeterminacy. The framework contains a number of transformations, allowing to refine expressions of a given pattern, subject to some conditions, by other expressions. The game was to start with a problem expression and to transform it into a "concrete" expression, i.e., not containing abstract dictions. The verification of a derivation consists of (apart from some routine pattern matching) proving a number of mathematical propositions. It was not hard to see that given an initial problem expression P and an intended concrete expression C, a derivation in the calculus would be possible if and only if the proposition

"P may be refined to C" was true. (I did not attempt to prove this formally, and this "completeness claim" may need some technical refinements.)

I observed, however, an unmistakable pattern in the derivations. The transformations steps could be categorized as belonging roughly to one of the following two types:
(i) bound variable substitution ("VS") and
(ii) imposing control ("IC"), i.e., introducing one of the three composition methods.
The VS steps may lead to the disappearance of bound variables altogether, after which an abstract expression may be replaced by a concrete expression. Their applicability reflects a mathematical theorem. The IC steps may introduce bound variables and useful invariants, but typically reflect no deep mathematical truth. Now, in a typical derivation, an IC step introduces bound variables to be eliminated and invariants to be exploited in subsequent VS steps. They lay the ground for applying the theorems we need. But in fact, they anticipate the VS steps. If I would not know *in advance* which theorems I would be going to use further on in the derivation, these steps would only lead to blind alleys. Not only does the derivation exhibit the theorems used, but conversely, given the theorems, the derivation is as good as determined. So applying the transformation rules is only possible after one knows which theorems to use, and is then further a chore. Two substantially different implementations have derivations that diverge from their very starts. Instead, I had hoped for situations where initial transformations would lead to semi-abstract expressions that might still be refined in radically different directions, or ways to transform radically different semi-abstract formulations into each other.

What now? A cursory examination has convinced me that the framework is, in fact, largely irrelevant: finding the theorems to be applied is the key to the development. I had not realized this before and plan to examine this more thoroughly to get a better understanding of what is going on, and in particular the framework-irrelevance aspect.

If the Abstracto dream is to come true, I need a radical departure from the line of thinking I have followed until now. I have no idea how to proceed. Really, the key "transformations" are the mathematical theorems and not the boring blind-pattern-match manipulations that I looked upon until now as being "the" transformations. This should provide some clue to the direction of research. Mimic mathematics not only in form, but also in substance. Or is it all a pipedream?

The reason why all of this was not immediately clear to me is probably that I am--next to being fond of doing formal manipulations with preferably as few symbols as possible--a very experienced and prolific programmer. For virtually all examples it was immediately obvious to me how to "do" them in concrete program form, so that I knew all the time where I was heading. One notable exception was the problem of the longest upsequence, and there it was painfully apparent that whatever guidance my

framework provided in the "discovery" process was shallow or even trivial. At some point in the "imperative" development of this problem it is clear enough what you are looking for--an extension of the obvious invariant that is still efficiently maintainable--and I found that all reasoning that leads to its construction had to take place completely outside the framework proper.

I then indeed started to examine the notion of transformations as being applied theorems in the most general setting I could devise that could be tied in with the notion of algorithm, which I called "pre-algorithmic systems". The first results of this examination had already been presented in Wheeling. Further investigation convinced me that all known transformations, and probably also all transformations yet to be discovered, could be expressed as transformations of such systems. This also included the mapping of imperative to applicative programs and vice versa, and in fact even program execution. Also, the methodological problem of having to see the theorems to be applied in advance did not appear, or at least not in a comparable severity. (Whatever the approach, some foresight may always be helpful.)

Why, then, did I not pursue this framework? The reason for that is that it was decidedly at the wrong level. My pre-algorithmic systems were to algorithms as Turing Machines are to abstract machines in general, or as first-order predicate calculus to mathematics. My motive in studying these systems was only to gain more understanding. For a similar reason, I did not want to embrace the applicative style, in spite of the apparent success of Burstall and Darlington's unfold/fold method. Consider that there is a standard way of "compiling" imperative source programs, however spaghetti-like their structure, to applicative object code, and that the best way to compile FOR and WHILE statements is by first expanding them by way of a source-to-source transformation in GOTO form. Thus, the whole advantage of having a notational embodiment for a higher-level concept than GOTO is lost: applicative programming considered harmful.

Then came the Nijmegen meeting, in May 1981, at which Richard Bird entered the stage and presented a paper entitled "Some notational suggestions for transformational programming". It used an applicative (functional) style, but the objections I had did not apply. There were notations for high-level concepts, and just the kind of manipulations, at the right level, that you would want to see. This sure looked like the warranted genuine article. Maybe a baby Snark, but still definitely worth investigating. My main worry was the scope of applicability. Would I find that I needed more and more primitive functions and corresponding rules as I did more examples? So I started doing some problems this way. First I found that I had indeed to invent new functions and laws all the time, which was disappointing. I put it down for some time, but took it up again while I was visiting NYU in '82/'83, since it still looked like the most promising avenue of research. Then I suddenly realized that there was a pattern in the new functions and laws. Investigating this led to a whole lot of other discoveries (the applicability to "generic" structures; the relationship to fictitious values), and I was very excited about this.

Reprinted from: Mathematics and Computer Science. Proc. CWI Symp. November 1983. CWI Monographs Vol. 1 (J.W. de Bakker, M. Hazewinkel and J.K. Lenstra, eds.). North-Holland, 1986.

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

$$\uparrow_{age}/+/mr*dm \;=\; \uparrow_{age}/(\uparrow_{age}/mr)*dm\,.$$

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, $f\,x$ stands for the application of f to x. If g is then applied to $f\,x$, this may be denoted by $g\,f\,x$. 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 "^" 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) $g0 = \times/0$,
(ii) $g\hat{x} = fx$,
(iii) $gs+t = gs; \times gt$.
Then $g = \times/f*$.

PROOF. By the induction lemma. For part (i), we have $g0 = \times/0 = \times/f*0$. For part (ii), $g\hat{x} = fx = \times/\hat{}fx = \times/f*\hat{x}$. For part (iii), by the induction hypothesis $gs = \times/f*s$ and $gt = \times/f*t$. Then $gs+t = gs; \times gt = \times/f*s; \times \times/f*t = \times/f*s+t$.

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 $\times'$ 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

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

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 $1 + \cdots + 1$. 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 $[]$ to denote an unspecified choice: the operation of making an arbitrary choice between two values. So $x [] y$ is a specification that is satisfied by any solution for x, but also by any solution for y. The expression $1[]2$ may yield 1, but may as well yield 2 (but not 3). The operation $[]$ is associative: $(x [] y) [] z$ is equivalent to $x [] (y [] z)$. It

is also commutative and idempotent. So $[]/s$ stands for an "arbitrary" choice from the structure s. Choosing from an empty structure can now be described with the formula $[]/0$. But no choice is possible, so what is the meaning of this formula? The answer is: "Nothing". A more learned answer is that $[]/0$ 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 $[]/0$ 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 $x[][]/0 = x$. In words, if we may choose "freely" between x and Nothing, then we must choose x.

An important identity for $[]$ is

$$fx[]y = fx;[]fy.$$

This corresponds to what is known in Formal Semantics as the "monotonicity" of f. We know then, from law 2 of Section 5, that $f[]/ = []/f*$. A prerequisite for general applicability of this law here, is, however, that the function be "strict", i.e., that the identity $f[]/0 = []/0$ 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 $[]/0$ 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 $[]/0$ is an identity of the operation $[]$, this gives simpler algebraic laws. Since then $x[][]/0 = x$, the function $x[]$ cannot be strict for satisfiable x, and so the identity $x[][]/s = []/x[]*s$ 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 $f[]/0$, 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 $fx = []/0$ then x else $[]/0$.

We can now define the asymmetric relation $\Rightarrow$ in terms of $=$ and $[]$, for $p \Rightarrow q$ has the same meaning as $p = p[]q$. A consequence is that $p \Rightarrow []/0$ 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 $[]/C:\mathbb{U}$ 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 $\{v \in \mathfrak{V} \mid e$ "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 $\mathfrak{E}$ (and $\mathfrak{V}$) 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 $\mathfrak{B}(e') \subset \mathfrak{B}(e)$. We take this as the characterization of $\Rightarrow$ in terms of $\mathfrak{B}$, replacing "requires" by "iff". This has some consequences. Call an expression f "flat" if $\mathfrak{B}(f)$ is the empty set. An example of a flat expression is $[]/0$ (assuming that we do not admit this pseudo-value in the distinguished company of the proper values). Then we find, for any e, $e \Rightarrow []/0$. But $[]/0$ 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 $\mathfrak{B}$ 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 $e \Rightarrow []/0$, 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 $[]/0 \Rightarrow e$ for any e. This is unacceptable here, since we would then find that each $e = []/0$. See, however, the notion of "total variant" of a function defined below.

If we start with some definition of $\mathfrak{B}$, next derive $\Rightarrow$ from that definition, and use $\Rightarrow$ then to find $\mathfrak{B}$, this will be the original function we started with. If, however, we start with some definition of $\Rightarrow$, use that to define $\mathfrak{B}$ 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:

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

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

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

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 $\mathfrak{B}$ 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 $f = \mathrm{id}$ ($= \lambda x: x$), $g = 3\ll$ ($= \lambda x: 3$), 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 $\lambda x: x[]3$ 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:

$$\mathrm{abs}\ 2\times e = \mathrm{if}\ (2\times e) < 0\ \mathrm{then}\ -(2\times e)\ \mathrm{else}\ (2\times e) =$$
$$\mathrm{if}\ e < 0\ \mathrm{then}\ 2\times -e\ \mathrm{else}\ 2\times e = 2\times \mathrm{if}\ e < 0\ \mathrm{then}\ -e\ \mathrm{else}\ e =$$
$$2\times \mathrm{abs}\ e.$$

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:

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

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 $\mathfrak{E}$, 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 $\mathfrak{E}$ is sufficiently expressive—to find some pair $\langle e, v\rangle \in \mathfrak{E} \times \mathfrak{V}$ 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 $\mathfrak{E}$.

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 $\uparrow_{age}/+/mr*dm$ 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

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

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:

> $fp\rightarrow = p\rightarrow f$, provided that f 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

> $s \leqslant t$ 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, $len\,\hat{w} = -1; + +/(1+\#)*\hat{w} = -1; +(1+\#)w = \#w$. 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:

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

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 $c = \hat{}*r$. Then (0) is, of course, satisfied, but what about (3a)? Since $len\,\hat{} = \#$, we find

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

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

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

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

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

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

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

In conclusion,

$$fr \neq \square/0 \text{ if and only if } \uparrow/\,\#*r \leqslant \text{MAX}.$$

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, $\uparrow/len*\hat{0} = \uparrow/\hat{}\,len\,0 = \uparrow/\hat{0} = 0$, 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

$$ws\,c = ws\,c'{+}\hat{l} = +/ws_1{*}c' = +/(\text{MAX}-len){*}c' =$$
$$+/(\text{MAX}-(-1)+{+}/(1+\#){*}){*}c' =$$
$$+/(\text{MAX}+1;-{+}/(1+\#){*}){*}c' =$$
$$\text{MAX}+1;\times\#c';-{+}/{+}/(1+\#){**}c' =$$
$$\text{MAX}+1;\times\#c';-{+}/(1+\#){*}{+}/c'.$$

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

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

so that we have

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

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

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

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, $len\,l < \text{MAX}+1$, the quantity $\#c'$ prevails over $len\,l$ in the comparison. This leads us to consider the simpler function

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

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 $f\,r{+}\hat{w}$ 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

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

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

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

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

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

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

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

So

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

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

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

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

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

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

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

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 $fr \neq []/0$, then $ff\, r \neq []/0$. The standard technique of recursion elimination gives the obvious iterative "eager" algorithm. Note also that $fr = []/0$ implies $ff\, r = []/0$. This is a consequence of $f \Rightarrow ff$, since then $[]/0 \Rightarrow fr \Rightarrow ff\, r \Rightarrow []/0$. 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.

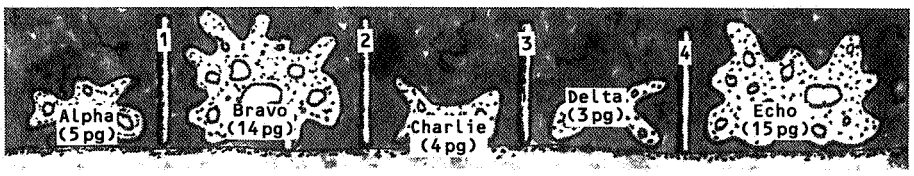


FIGURE 3. Five amoebae lined up before the tournament (magnification: 500 ×)

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

$$wd = \times/ wd_0 * .$$

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 $s = +/\hat{}*t$, in which the inserted operation $+$ introduces associativity. Then

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

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 $dt_1 = (w_1+w_2)(w_1w_2+w_3)$ and $dt_2 = (w_2+w_3)(w_1+w_2w_3)$. 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

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

$$t\, \hat{w}_1 + s' + \hat{w}_n \Leftarrow (w_1 \leqslant m_R ; \rightarrow \hat{w}_1 + t\, R) \; [] \; (w_n \leqslant m_L ; \rightarrow t\, L ; + \hat{w}_n),$$
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.

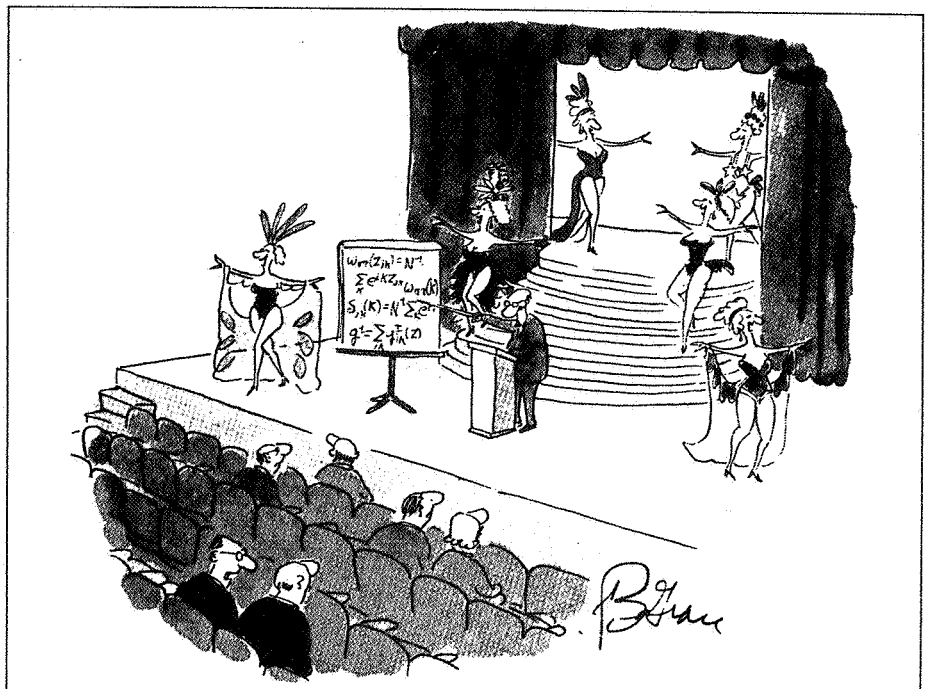
13. Conclusion

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.

SOME MORE EXAMPLES OF ALGORITHMIC DEVELOPMENTS

"The method employed I would gladly explain,
While I had it so clear in my head,
If I had but the time and you had but the brain--
But much yet remains to be said."

You can perhaps imagine my disappointment when I heard from Richard that he had dropped this whole approach because he found it was generally ununderstandable to audiences. Subsequent presentations of the Algorithmics paper at WG 2.1 meetings strongly suggested the same to me: I have presented basically the same talk three times at three consecutive meetings (that is, once per meeting), mainly with the effect of drawing blank stares or questions like if I thought "ordinary programmers" would ever be able to understand this. Still, I stubbornly refused to believe that this was due to something else than lack of familiarity, and, of course, my way of presentation, which tends to be a bit dense. After all, most of it is not harder than much of high-school mathematics.

So I plodded on undaunted, and continued to present examples. This paper was presented at the meeting in Pont-à-Mousson, held in September 1984. I have "modernized" the notation (mainly by not using the generic ^ and + for structures, but [·] and ++ for sequences, and so on), and inserted many more parentheses than are strictly needed given my conventions.

With these changes it still does not make for easy reading. Next to the fact that the information density is of course a lot higher than in most other styles, I think that this is mainly so because the development as given is just as it occurred to me (which is also the case for the Algorithmics paper). The development is a mixture of parts that are specific to the problem at hand, and parts of a much wider applicability but where some necessary theory is developed as it were on the fly, and then not even in a general form but tailored to the specific problem. (Note, in particular, the similarity between the development of the "smallest upravel" here and the text-formatter problem in the Algorithmics paper.) As a consequence, the structure of the derivation is obscured. It is possible to give a much clearer exposition if a modicum of theory is developed first. Richard Bird has done exactly that for some of these kind of problems, and by applying the notions and theorems from his "Theory of Lists" my developments, or at least substantial parts, can be dramatically simplified, while making the proof obligations much clearer. Even without this, the use of the "directed reduce" notation would already have helped to structure the presentation (in fact, for both problems).

Other parts where some standard theory is waiting to be developed are the constructive inversion of certain types of functions and the linearizing of call trees, as in the "longest common subsequence" problem, by collecting arguments from different calls together. A much nicer way than that used here is to recognize the fact that if the basic recursion pat-

tern of some function f is

```
f x = ⊕ / f * children x
```

(in which the "leaves" are omitted for the sake of simplicity), then we can express f as h∘call_tree, where

```
call_tree x = call_tree * children x
```

and h is a homomorphism satisfying

```
h t = ⊕ / h * t .
```

If ⊕ is now, for example, associative and idempotent, then we may perform rotations in the tree and cut away some duplicate branches. The advantage is that we do not have to think in terms of dynamic structures, but are on the familiar ground of homomorphisms on data structures.

IFIP WG2.1 Working paper ADP-7, 1984.

Some more examples of algorithmic developments

Lambert Meertens
Centrum voor Wiskunde en Informatica
Amsterdam

0. Preliminary

This paper is not self-contained. The notations and concepts used are explained in reference [3]. It has been prepared as a working paper for WG2.1.

A new addition is the B function. It is the functional inverse of the function $[]/$ and expresses on the language level the meta-level breadth function $\mathfrak{B}$. In [3] it was stated that this function could not be admitted to the language, since this destroys monotonicity. However, B is a useful acquisition, and since refinement steps $e \Rightarrow e'$ are rare, it is better to allow one exception to monotonicity. Refinement of expressions involving B is possible, as long as it does not happen inside an argument of B. It turns out possible to give a simple calculus for juggling with, and in particular, eliminating B. The major rule is of course

$$\mathsf{B}e_1 [] e_2 = \mathsf{B}e_1 ; \cup \mathsf{B}e_2 .$$

If e is determinate, $\mathsf{B}e = \{e\}$. I have not yet gotten around to produce a readable write-up of this calculus. Rules of the B-calculus will nevertheless be freely applied. Usually their justification will be intuitively obvious. (But some seemingly obvious transformations are unsound, as I have noticed, so beware.)

1. Longest common subsequence

A *subsequence* of a given sequence is a sequence that can be obtained by deleting any number of elements from the original sequence. The remaining elements need not be contiguous in the original sequence. So the sequence

```
s c a r e c r o w
```

has a subsequence

```
a r     r o w.
```

The problem is: given two sequences s and t, to determine the longest common subsequence of s and t. So, if s = "`scarecrow`" and t = "`tarrytown`", the answer is "`arrow`". Note that there may be different common subsequences of the maximum length, as between "`bathtub`" and "`perturbate`", namely "`bat`" and "`tub`". In such a case, any of these is an equally acceptable result. So the result to be determined is in general indeterminate.

There is an obvious dynamic-programming solution that will run in time $\Theta(\#s \cdot \#t)$. This is also the worst-case running time of the best known methods, but on the average "practical" case they may do much better. The problem has been treated by (among others) Hunt & McIllroy [1], who wrote a widely used program for finding a minimal set of differences between two files—which is an equivalent problem. A linear-space algorithm was given by Hirschberg [2]. The purpose of treating

this example is not to find better solutions, but simply to examine how well the method can deal with it. The problem of determining the longest upsequence is a special, simpler case of the current problem.

Let $s{\downarrow}t$ denote the longest common subsequence of s and t. (The operation $\downarrow$ is not defined on sequences in [3], so we can freely use this symbol. However, the choice of this symbol is not just a whim, for $\downarrow$ on two sets returns the largest common subset.) Then

$$s{\downarrow}t = \uparrow_{\#}/s \mathbin{/\!\!\wedge\!\backslash} t,$$

where $s \mathbin{/\!\!\wedge\!\backslash} t$ stands for the set of *all* common subsequences of s and t. The property of being a subsequence of s is expressed by the predicate $s{\geqslant}$, so we have

$$s \mathbin{/\!\!\wedge\!\backslash} t = t{\geqslant}\triangleleft s{\geqslant}\triangleleft \mathbb{U}.$$

Then

$$[] \mathbin{/\!\!\wedge\!\backslash} t = t{\geqslant}\triangleleft []{\geqslant}\triangleleft \mathbb{U} = t{\geqslant}\triangleleft \{[]\} = \{[]\};$$

$$([x]{+\!\!+}s) \mathbin{/\!\!\wedge\!\backslash} t = t{\geqslant}\triangleleft ([x]{+\!\!+}s){\geqslant}\triangleleft \mathbb{U} = t{\geqslant}\triangleleft ((s{\geqslant}\triangleleft \mathbb{U}) \cup ([x]{+\!\!+})*s{\geqslant}\triangleleft \mathbb{U}) =$$
$$(t{\geqslant}\triangleleft s{\geqslant}\triangleleft \mathbb{U}) \cup t{\geqslant}\triangleleft ([x]{+\!\!+})*s{\geqslant}\triangleleft \mathbb{U} = (s \mathbin{/\!\!\wedge\!\backslash} t) \cup t{\geqslant}\triangleleft ([x]{+\!\!+})*s{\geqslant}\triangleleft \mathbb{U}.$$

Remark. Whether we "recurse" through s or t, or through the beginning or end of an argument, is of lesser importance, because of the symmetry of the problem. A choice is made, however, in not using "$(s_1 {+\!\!+} s_2) \mathbin{/\!\!\wedge\!\backslash} t$" for the development. This would give a more complicated expression, since we then have to consider subsequences of two parts of t simultaneously. The price is that we hereby lose the possible interpretation that we are determining the largest common sub*tree*.

If we look at the second term of the last form, a sequence of the form $[x]{+\!\!+}s_1$ can only be a subsequence of t if t can be written as $t_0 {+\!\!+} [x] {+\!\!+} t_1$, where s_1 is a subsequence of t_1, and so, if s_1 is a subsequence of s, $[x]{+\!\!+}s_1$ is an element of $([x]{+\!\!+})*s \mathbin{/\!\!\wedge\!\backslash} t_1$. If $Post_x t$ stands for the set of all such tails t_1 of t following an occurrence of x, we have

$$t{\geqslant}\triangleleft ([x]{+\!\!+})*s{\geqslant}\triangleleft \mathbb{U} = ([x]{+\!\!+})^* \cup /(s \mathbin{/\!\!\wedge\!\backslash})*Post_x t.$$

Then

$$([x]{+\!\!+}s){\downarrow}t = \uparrow_{\#}/(s \mathbin{/\!\!\wedge\!\backslash} t) \cup ([x]{+\!\!+})^* \cup /(s \mathbin{/\!\!\wedge\!\backslash})*Post_x t =$$
$$(\uparrow_{\#}/s \mathbin{/\!\!\wedge\!\backslash} t) \uparrow_{\#} \uparrow_{\#}/([x]{+\!\!+})^* \cup /(s \mathbin{/\!\!\wedge\!\backslash})*Post_x t = (s{\downarrow}t) \uparrow_{\#} ([x]{+\!\!+})*\uparrow_{\#}/(s{\downarrow})*Post_x t.$$

Now $\downarrow$ is weakly monotonic in its arguments, so

$$\uparrow_{\#}/(s{\downarrow})*Post_x t = s{\downarrow}\uparrow_{\#}/Post_x t.$$

Here, $\uparrow_{\#}/Post_x t$ is the longest element of the tails of t following an occurrence of x, so it is simply the tail of t following the first occurrence of x, if any, in t, and the (fictitious) value $[]/[]$ if no x occurs in t, where we put

$$r {+\!\!+} s{\downarrow}[]/[] = []/[] = \uparrow_{\#}/[] \text{ for all } r \text{ and } s.$$

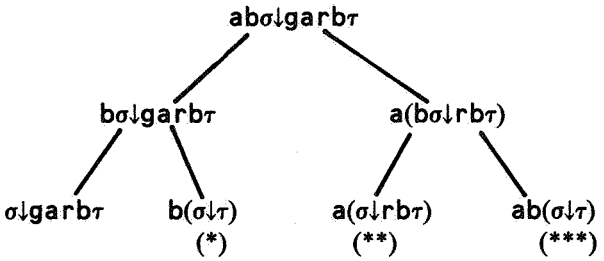
If we denote that longest tail by $post_x t$, we have

$$[]{\downarrow}t = [];$$

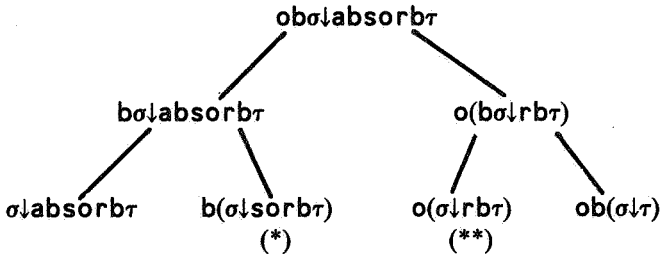
$$([x]{+\!\!+}s){\downarrow}t = (s{\downarrow}t) \uparrow_{\#} ([x]{+\!\!+}s{\downarrow}post_x t).$$

If we replace here "$=$" by "$\Leftarrow$", we have an effective ("executable") specification, provided $post_x$ is effectively defined. Some practical remarks: the two arguments of $\downarrow$ are always, respectively, an initial part of the original sequence s and a final part of the original sequence t. In a practical implementation, s and t could be globally available and the arguments passed could just be indices of s and t. The function $Post_x$ can be partially precomputed, by storing a table mapping indices of s to sets of indices of t. With the canonical evaluation scheme, however, this is still inefficient: we find an exponential number of nodes in the computation tree. We do much better if we recognize the fact

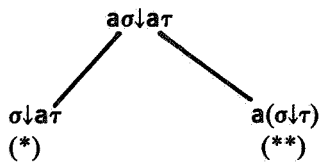
that many sub-applications of ↓ will have the same arguments, and do not recompute these, e.g. by using a table to store previous results. With this simple change we have (a slight improvement on) the obvious dynamic-programming method. To do still better in a "practical" case, we have to look how the applications of ↓ unfold into a tree. For this we use a brief notation, in which, e.g., "**a(σ↓rbτ)**" stands for "$[a]{+\!\!+}(\sigma{\downarrow}([r,b]{+\!\!+}\tau))$". Then we have, e.g.,



Here (***) is superior to (*), and (**) is at least as good. This can be used to prune the tree by snipping (*) off. Another example is shown by



In this case, (*) is at least as good as (**). Finally, in



(**) is at least as good as (*). This is intuitively obvious: in two sequences that start with identical elements, we may pair these off. Curiously, this case is the hardest to prove formally.

In the general case, the nodes of the tree have the pattern (still using the brief notation) $\gamma_i(s{\downarrow}\tau_i)$, where all nodes on the same depth have the same value for s. Non-competitive nodes can be discarded. To express this optimization, we must replace the recursion pattern by one carrying a *collection* of "candidates", being pairs (γ, τ). We define

$$s \propto C \Leftarrow \uparrow_{\#}/f_s{*}C, \text{ where}$$

$$f_s(\gamma,\tau) \Leftarrow \gamma{+\!\!+}s{\downarrow}\tau.$$

If we can compute $\propto$, we can also compute ↓, using

$$s \propto \{([\,],t)\} = \uparrow_{\#}/f_s{*}\{([\,],t)\} = f_s([\,],t) = s{\downarrow}t.$$

Since

$$f_{[\,]}(\gamma,\tau) = \gamma{+\!\!+}[\,]{\downarrow}\tau = \gamma = \pi_1(\gamma,\tau),$$

we have

$$[\,] \propto C = \uparrow_{\#}/f_{[\,]}{*}C = \uparrow_{\#}/\pi_1{*}C.$$

For $f_{[x]+\!\!+s}$ we find

$$f_{[x]+\!\!+s}(\gamma,\tau) = \gamma+\!\!+([x]+\!\!+s;\downarrow\tau) = \gamma+\!\!+(s\downarrow\tau;\uparrow_{\#}[x]+\!\!+s\downarrow post_x\tau) =$$
$$\gamma+\!\!+s\downarrow\tau;\uparrow_{\#}\gamma+\!\!+[x]+\!\!+s\downarrow post_x\tau = f_s(\gamma,\tau);\uparrow_{\#} f_s(\gamma+\!\!+[x], post_x\tau) = \uparrow_{\#}/f_s*succ_x(\gamma,\tau),$$

where

$$succ_x(\gamma,\tau) \Leftarrow \{(\gamma,\tau),(\gamma+\!\!+[x], post_x\tau)\}$$

gives the two possible ways in which an application of $\downarrow$ can develop. We must define then

$$f_s(\gamma, \square/[]) = \square/[] = \uparrow_{\#}/[],$$

and admit $\square/[]$ in structures. This can be circumvented by replacing "$\{(\gamma+\!\!+[x], post_x\tau)\}$" in the definition of $succ_x$ by "$\mathsf{B}(x\in\tau\rightarrow(\gamma+\!\!+[x], post_x\tau))$". (Since $p\rightarrow x$ stands for "if p then x else $\square/\{\}$", $\mathsf{B}(p\rightarrow x)$, for determinate p and x, stands for "if p then $\{x\}$ else $\{\}$"). Then

$$([x]+\!\!+s)\propto C = \uparrow_{\#}/f_{[x]+\!\!+s}*C = \uparrow_{\#}/(\uparrow_{\#}/f_s*succ_x)*C = \uparrow_{\#}/f_s* \cup/succ_x*C =$$
$$s\propto \cup/succ_x*C.$$

We have now an effective definition of $\propto$, not depending on $\downarrow$.

This was only a standard exercise in replacing a recursion pattern that spreads out into a tree by a linear-branch pattern, in which, instead, the argument spreads out. The non-competitive candidates have to be weeded out still. In particular, we would like to define a function *weed*, such that

$$s\propto C \Rightarrow s\propto weed\, C.$$

We can look for a definition of the form

$$weed\,\{\} \Leftarrow \{\};$$

$$weed\,(C\cup\{c\}) \Leftarrow (weed\, C)\oplus c.$$

The necessary conditions can be derived by starting to prove (by induction) that C may be replaced by *weed C*. We write, for brevity, W for *weed C*. So, assume $s\propto C\Rightarrow s\propto W$. Then

$$s\propto(C\cup\{c\}) = (s\propto C)\uparrow_{\#} f_s c \Rightarrow (s\propto W)\uparrow_{\#} f_s c = s\propto(W\cup\{c\}).$$

So $\oplus$ must be such that

$$s\propto(W\cup\{c\})\Rightarrow s\propto(W\oplus c).$$

We can safely start to define

$$\{\}\oplus c \Leftarrow \{c\}.$$

The next step is to try something like

$$(W\cup\{w\})\oplus c \Leftarrow W\cup cc(w,c),$$

in which cc gives the competitive candidates between w and c. Now we choose to represent W as a sequence, and then $succ_x$ must return a sequence too. (Representing W as a set will give a correct, but probably inefficient, final program.) The correctness criterion for cc is

$$s\propto(W+\!\!+[w,c]) = (s\propto W)\uparrow_{\#}(s\propto[w,c])\Rightarrow s\propto(W+\!\!+cc(w,c)) = (s\propto W)\uparrow_{\#}(s\propto cc(w,c)).$$

So a sufficient condition is

$$s\propto[w,c]\Rightarrow s\propto cc(w,c).$$

Now

$$s\propto[w,c] = \uparrow_{\#}/f_s*[w,c] = f_s\uparrow_{\#f_s}/[w,c] = f_s(w\uparrow_{\#f_s}c).$$

Similarly

$$s\propto cc(w,c) = f_s\uparrow_{\#f_s}/cc(w,c).$$

So it is sufficient if

$$w\uparrow_{\#f_s} c \Rightarrow \uparrow_{\#f_s} / cc(w, c).$$

An immediate solution is given by

$$cc(w, c) = [w\uparrow_{\#f_s} c],$$

but this is not too helpful, since the definition of f_s involves $\downarrow$, which is what we are trying to define in a computationally better way. Each method that always replaces w and c together by just one candidate must take s into account and will require substantial effort. If we do not want to consider the particular current value of s, the correctness criterion for cc is

$$w\uparrow_{\#f_s} c \Rightarrow \uparrow_{\#f_s} / cc(w, c) \text{ for all } s.$$

This can be rephrased as:

$cc(w, c) \leqslant [w, c]$;

if, for some s, $w >_{\#f_s} c$, then $w \in cc(w, c)$;

if, for some s, $w <_{\#f_s} c$, then $c \in cc(w, c)$.

Put $w = (\gamma_w, \tau_w)$ and $c = (\gamma_c, \tau_c)$. If $\gamma_w \leqslant_{\#} \gamma_c$ and $\tau_w \leqslant \tau_c$, then, because of the weak monotonicity of $\downarrow$,

$$w \leqslant_{\#f_s} c \text{ for all } s,$$

so w need not be included then. In the reverse case, c need not be included (but we must include at least one of the two, since we do not want to loose definedness). Moreover, if, for some ξ, $\gamma_w + \!\!+ \xi = \gamma_c$ and $\tau_w = \xi + \!\!+ \tau_c$, then w can be discarded. This can be seen as follows. Every realization of $s\downarrow(\xi + \!\!+ \tau_c)$ can be written as a realization of the form $(s_1\downarrow\xi) + \!\!+ (s_2\downarrow\tau_c)$, where $s = s_1 + \!\!+ s_2$. Then

$$f_s w = \gamma_w + \!\!+ s\downarrow(\xi + \!\!+ \tau_c) = \gamma_w + \!\!+ (s_1\downarrow\xi) + \!\!+ (s_2\downarrow\tau_c) \leqslant_{\#} \gamma_w + \!\!+ \xi + \!\!+ s_2\downarrow\tau_c \leqslant_{\#} \gamma_w + \!\!+ \xi + \!\!+ s\downarrow\tau_c = f_s c.$$

This last optimization is of interest if ξ consists of one element. It is then best expressed as a modification to the $succ_x$ function, namely by replacing "$\{(\gamma, \tau)\}$" in the definition of $succ_x$ by "$\mathsf{B}(x \neq first\ \tau \rightarrow (\gamma, \tau))$", in which it is understood that *first* [ ] is some fictitious value such that $x \neq first[\,]$ for all proper values x.

The efficiency improvement aimed at is certainly obtained, for the weeding ensures that the γ-components of W have strictly increasing lengths, so $\#W$ is bounded by $\#s$. In a practical imperative implementation, the computations of *weed* and of $+\!\!+/succ_x*$ can be merged. Such mergings can be obtained by a simple unfold/fold, but this is somewhat boring and yields no further simplification. Also, on non-Von architectures this is possibly no improvement anyway. To sum up the findings, we have

$$s\downarrow t \Rightarrow s \propto' [([\,], t)],$$

where

$[\,] \propto' W \Leftarrow \pi_1\ last\ W$;

$([x] + \!\!+ s) \propto' W \Leftarrow s \propto weed\,(+\!\!+/succ_x * W)$;

$succ_x(\gamma, \tau) \Leftarrow \mathsf{B}(x \neq first\ \tau \rightarrow (\gamma, \tau)); + \!\!+ \mathsf{B}(x \in \tau \rightarrow (\gamma + \!\!+ [x], post_x \tau))$;

$post_x([y] + \!\!+ \tau) \Leftarrow (x = y \rightarrow \tau [] x \neq y \rightarrow post_x \tau)$;

$weed[\,] \Leftarrow [\,]$;

$weed\,(W + \!\!+ [c]) \Leftarrow (weed\ W) \oplus c$;

$[\,]\oplus c \Leftarrow [c]$;

$(W +\!\!+ [w])\oplus c \Leftarrow W +\!\!+ cc(w, c)$;

$cc(w, c) \Leftarrow (b = [\,] \rightarrow [w]\!]c]\,]\!]\,b \neq [\,] \rightarrow b)$,
where $b = \mathsf{B}((\gamma_w >_{\#} \gamma_c;\ \vee \tau_w >_{\#} \tau_c) \rightarrow w]\!](\gamma_c >_{\#} \gamma_w;\ \vee \tau_c >_{\#} \tau_w) \rightarrow c)$.

I have not eliminated B here, mainly since this would mess up a clear specification, but note that it is supposed to return a sequence in this context. Note also a final optimization in the definition of $[\,] \propto' W$.

2. Smallest upravel

A *ravel* of a given sequence is a bag of subsequences that "shuffled" together can give the original sequence back. For example, the sequence "**accompany**" can be raveled into "**am**", "**copy**" and "**can**", as follows:

```
a c c o m p a n y

a       m
  c   o   p     y
    c       a n  .
```

A ravel is an *upravel* if all its elements are upsequences, i.e., strictly increasing. The above ravel is not an upravel, since "**can**" is not increasing. (For the examples, we have $\mathbf{a} < \mathbf{b} < \cdots$.) An upravel is given by "**an**", "**acm**" and "**copy**". Each sequence has of course at least one upravel, by turning each of its elements into a one-element sequence. Of all the possible upravels, we want to determine one with the least number of elements.

A possible application is in sorting sequences that are almost in order already, or that have been formed by catenating a small number of (almost) sorted sequences or by merging such sequences according to some irrelevant criterion. In such a case, one can sort a sequence by merging the elements of an upravel. The condition "strictly increasing" should then of course be replaced by "weakly increasing". This makes only a marginal difference.

To proceed with a formal development, we first define what it means to shuffle *two* sequences s and t:

$[\,] \bowtie t \Leftarrow t$;

$s \bowtie [\,] \Leftarrow s$;

$[x] +\!\!+ s;\ \bowtie [y] +\!\!+ t \Leftarrow [x] +\!\!+ (s \bowtie ([y] +\!\!+ t))\ ]\!]\ [y] +\!\!+ (([x] +\!\!+ s) \bowtie t)$.

So the first element of a shuffle of s and t (unless s and t are both empty) is the first element of at least one of s and t, and the rest of the shuffle is then a shuffle of s and t but with that first element removed from one of the two. The operation $\bowtie$ is obviously commutative, and (somewhat less obvious) associative. A shuffle of a whole bag of sequences u is then given by $\bowtie/u$. If we want to have *all* possible shuffles, we can use the B function. I just give the results of applying the B-elimination calculus:

shuffles $u \Leftarrow \mathsf{B} \bowtie/u = \dot{\bowtie}/\mathsf{B}*u = \dot{\bowtie}/\{\cdot\}*u$.

$S \dot{\bowtie} T \Leftarrow \cup/(\pi_1 \bowtie_{\mathsf{B}} \pi_2) S \times T$;

$[\,] \bowtie_{\mathsf{B}} t \Leftarrow \mathsf{B}([\,] \bowtie t) = \mathsf{B}t = \{t\}$;

$s \bowtie_{\mathsf{B}} [\,] \Leftarrow \mathsf{B}(s \bowtie [\,]) = \mathsf{B}s = \{s\}$;

$$([x]{+\!\!+}s) \bowtie_{\mathsf{B}} ([y]{+\!\!+}t) \Leftarrow \mathsf{B}([x]{+\!\!+}s) \bowtie ([y]{+\!\!+}t) =$$
$$\mathsf{B}([x]{+\!\!+}(s \bowtie ([y]{+\!\!+}t)) \,[]\, [y]{+\!\!+}(([x]{+\!\!+}s) \bowtie t)) =$$
$$\mathsf{B}([x]{+\!\!+}(s \bowtie ([y]{+\!\!+}t))); \cup \mathsf{B}([y]{+\!\!+}(([x]{+\!\!+}s) \bowtie t)) =$$
$$([x]{+\!\!+})*\mathsf{B}(s \bowtie ([y]{+\!\!+}t)); \cup ([y]{+\!\!+})*\mathsf{B}(([x]{+\!\!+}s) \bowtie t) =$$
$$([x]{+\!\!+})*(s \bowtie_{\mathsf{B}} ([y]{+\!\!+}t)); \cup ([y]{+\!\!+})*(([x]{+\!\!+}s) \bowtie_{\mathsf{B}} t).$$

Here, $S \times T$ is the Cartesian product of S and T. Since $S \times \{x\} = (\mathrm{id}, x \ll)*S$,

$$S \dot{\bowtie} \{[]\} = \cup/(\pi_1 \bowtie_{\mathsf{B}} \pi_2)(\mathrm{id}, [] \ll)*S = \cup/(\mathrm{id} \bowtie_{\mathsf{B}} [] \ll)*S = \cup/\{\cdot\}*S = S,$$

so the singleton set $\{[]\}$, consisting of an empty sequence, is an identity of this operation $\dot{\bowtie}$, and we have $\bowtie/\langle\rangle = \{[]\}$.

A bag u is a ravel of a sequence s if $s \in \mathit{shuffles}\, u$. The property of being an upsequence is tested by

$$\mathit{up}\ [] \Leftarrow \mathsf{T};$$

$$\mathit{up}\ ([x]{+\!\!+}s) \Leftarrow (x < \mathit{first}\, s) \wedge \mathit{up}\ s.$$

Here we put $\mathit{first}\,[] = \downarrow/[]$, which is larger than all proper values. The smallest upravel is now

$$\mathit{su}\, s \Leftarrow \downarrow_{\#}/(\wedge/\mathit{up}*) \triangleleft (s \in \mathit{shuffles}) \triangleleft \mathbb{U}.$$

If u is an upravel of s, then so is $([]\neq)\triangleleft u$. Since, moreover, $([]\neq)\triangleleft u \leqslant_{\#} u$, we can insert a filter $([]\notin)\triangleleft$ in the definition of su in front of $\mathbb{U}$ to sift out ravels containing the empty sequence. An arbitrary upravel that does not contain the empty sequence is now given by

$$\mathit{upravel}\, s \Leftarrow []/(\wedge/\mathit{up}*) \triangleleft (s \in \mathit{shuffles}) \triangleleft ([]\notin) \triangleleft \mathbb{U},$$

A general paradigm for synthesizing an effective definition of su is the incremental strategy. Note the formal resemblance to the text-formatter problem. Some informal reasoning shows that a properly truncated smallest upravel of some sequence is a smallest upravel of the truncated sequence, so there is hope. Truncation is here the removal of the first element of a sequence, and proper truncation requires the removal of sequences of an upravel that become empty through truncation. However, a problem is that in general the smallest upravel of a given sequence cannot be formed by extending just any smallest upravel of the truncated sequence. For example, although "`at`" and "`e`" form a smallest upravel of "`ate`", it cannot be extended to form the (unique) smallest upravel "`ft`" with "`ae`" of "`fate`". Whether an upravel u can be extended with an element x without increasing its size, depends on the collection $\mathit{first}*u$ of the first elements of the sequences of u. The extension is possible if (and only if) $(x<)\triangleleft \mathit{first}*u$ is not empty. In constructing u, we do not want to "look ahead", i.e., take the value of x into consideration. (This is the essence of the incremental strategy.) So we can try to apply a selection among the possibly many upravels of the smallest possible size to keep $\mathit{first}*u$ in some sense as "large" as possible, so that the widest possible range of x's can be accommodated in a size-preserving extension. If this ordering would be total, we could just refine the operation $\downarrow_{\#}$ in the definition of su to take the new ordering as a subordinate criterion in the selection.

So we try

$$\mathit{su}\, s \Rightarrow \mathit{su}'\, s,$$

where $\mathit{su}'\, s$ is the function realizing the restricted selection process:

$$\mathit{su}'\, s \Leftarrow \downarrow_{(\#,\, \mathit{first}\cdot)}/(\wedge/\mathit{up}*) \triangleleft (s \in \mathit{shuffles}) \triangleleft ([]\notin) \triangleleft \mathbb{U},$$

in which the ordering on the codomain of $\mathit{first}*$ is still to be determined. For u to satisfy $[] \in \mathit{shuffles}\, u$, $[] = []/\mathit{shuffles}\, u$ must be satisfiable. (A property p is "satisfiable" if $p \Rightarrow 1$.) So

$$[] = []/\mathit{shuffles}\, u = []/\mathsf{B} \bowtie/u = \bowtie/u$$

must be satisfiable. From here on I will not repeat "must be satisfiable". It is easy to see from the definition of $\bowtie$ that $s \bowtie t = []$ if and only if s and t are both empty. So $\bowtie/u = []$ if and only

if $\wedge/([\,]=)*u$. If u satisfies $[\,]\notin u$ too, $\approx/u = [\,]$ if and only if $\wedge/(\mathsf{F}\ll)*u$, i.e., $u = \langle\,\rangle$. Since $\langle\,\rangle$ is, trivially, a bag of upsequences, we can define

$$su'\langle\,\rangle \Leftarrow [\,].$$

We can try to find an incremental development by putting

$$su'[x]+\!\!+s \Leftarrow x \oplus su'\,s\,,$$

in which $\oplus$ has to be determined. Let us first treat the simpler case as though the part "$\downarrow_{(\#,\,first\,\cdot)}/(\wedge/up*)\triangleleft$" above simply read "$[]/$". We want to have then

$$[]/([x]+\!\!+s\,;\in shuffles)\triangleleft([\,]\notin)\triangleleft\mathbb{U} = x \oplus_0 []/(s \in shuffles)\triangleleft([\,]\notin)\triangleleft\mathbb{U}.$$

Proceeding as above, we want to determine u' satisfying

$$[x]+\!\!+s = \approx/u',$$

where u such that $s = \approx/u$ and $[\,]\notin u$ is known. (This is the constructive counterpart of the usual inductive hypothesis, so we may dub it the "constructive hypothesis"). We can rewrite $\approx/$ in such a way that it becomes explicit which elements can go to the front, by "computing" *split* $\approx/$, where

$$split = (first, tail).$$

(The function *tail* is, of course, defined as $tail\,[x]+\!\!+s \Leftarrow s$.) We can push *split* inwards, provided we can find an operation $\approx_s$ such that

$$split\,s \approx t = split\,s;\ \approx_s split\,t\,.$$

We find (by unfolding $\approx$) for *non-empty* sequences:

$$(x, s)\approx_s(y, t) \Leftarrow (x, s \approx([y]+\!\!+t))[](y, ([x]+\!\!+s)\approx t).$$

Since *split* is ill-defined on empty sequences, we cannot hope to extend $\approx_s$ to accept empty operands. (Actually, this can be done, but this requires much ado about "Nothings" with magical properties.) Fortunately, $[\,]$ is the identity of $\approx$, so we can insert, without change of meaning, a filter to suppress empty sequences in u.

It is easy to see that

$$\pi_1 \approx_s/split* = first*.$$

If we define

$$Trnc_x\,u \Leftarrow \pi_2\,[]/(x = \pi_1)\triangleleft\mathsf{B}\approx_s/split*u\,,$$

we obtain (or should obtain; I must confess that I have not attempted to do this in detail for lack of time and interest) by unfold/fold:

$$shuffle\,u \Leftarrow shuf([\,]\neq)\triangleleft u\,;$$

$$shuf\,u \Leftarrow u = \langle\,\rangle \rightarrow [\,]\ []\ u \neq \langle\,\rangle \rightarrow f_u\,[]/first*u\,;$$

$$f_u\,x \Leftarrow x +\!\!+ shuffle\ Trnc_x\,u\,;$$

$$Trnc_x\,(\langle[x]\rangle \sqcup u) \Leftarrow u\,;$$

$$Trnc_x\,(\langle[x]+\!\!+v\rangle \sqcup u) \Leftarrow \langle v\rangle \sqcup u\,.$$

The last two line gives an *indeterminate* definition, since both left-hand sides match if $v = [\cdot]$. Moreover, in both lines the argument is a bag, and there may be several ways to bring a member starting with x in front. Now $\approx/u = shuffle\,u$. Now we see that u' must be such that $x \in first*u'$ and $Trnc_x\,u' = u$. So $u' = ext_x\,u$, where ext_x is the functional inverse of $Trnc_x$, so by simply "swapping" the argument with the right-hand side, we have

$$ext_x\,u'' \Leftarrow \langle[x]\rangle \sqcup u'';$$

$$ext_x(\langle v\rangle \sqcup u'') \Leftarrow \langle [x]{+\!\!+} v\rangle \sqcup u''.$$

This definition is again indeterminate, and this is essential. We see that $x \in first{*}u'$ is automatically satisfied. Also, if $[\,] \notin u$, then $[\,] \notin u'$. So we have now

$$x \oplus_0 u \Leftarrow ext_x\, u.$$

This would be fine for determining arbitrary ravels. To get an upravel, we define

$$x \oplus_1 u \Leftarrow ifup\,(x \oplus_0 u);$$

$$ifup\ u \Leftarrow \wedge/up{*}u \rightarrow u.$$

It is understood here that if there are several choices in matching an application to the left-hand side of an indeterminate definition, only choices should be made whose right-hand sides are not flat, if possible. The definition is sufficiently effective, since there is only a finite number of possible matchings in the definition of ext_x for any given actual argument, so an automaton could simply try these one by one until one is found that does not lead to a dead end. We could, if we wanted to, rewrite the definition in such a way that no dead end could be encountered in the canonical evaluation. Instead, we will only simplify matters somewhat. After some unfolding of $ifup\,(x \oplus_0 u)$, we run into the formulae:

$$\wedge/up{*}(\langle [x]\rangle \sqcup u'') \rightarrow \langle [x]\rangle \sqcup u'';$$

$$\wedge/up{*}(\langle [x]{+\!\!+} v\rangle \sqcup u'') \rightarrow \langle [x]{+\!\!+} v\rangle \sqcup u''.$$

By the constructive hypothesis, $\wedge/up{*}u''$ is satisfied for the first formula, so its guard is satisfied as well. For the second formula, we know that $\wedge/up{*}(\langle v\rangle \sqcup u'')$, so its guard can be simplified to $x < first\ v$. We redefine ext_x now as

$$ext_x\, u'' \Leftarrow \langle [x]\rangle \sqcup u'';$$

$$ext_x(\langle v\rangle \sqcup u'') \Leftarrow x < first\ v \rightarrow \langle [x]{+\!\!+} v\rangle \sqcup u''.$$

Then

$$x \oplus_1 u = ext_x u.$$

Now for the hard part. The u' to be determined has also to be $(\#, first{*})$-minimizing, where we may use the constructive hypothesis that u minimizes that same function. We start by defining a partial ordering on collections of first elements of the sequences of an upravel that captures as much as we can summon of the ability to accommodate prospective x's to be prepended. The lower in the ordering, the more accommodating. It is assumed here that $first{*}$, applied to a bag, yields another bag (for reasons that will become apparent), so the domain of the partial ordering to be defined consists of pairs of bags. We use the symbol $\preccurlyeq$ for the partial-ordering relation. Then we require:

$$\langle\rangle \preccurlyeq b;$$

$$(\langle [x]\rangle \sqcup b) \preccurlyeq (\langle [x]\rangle \sqcup c) \text{ if } b \preccurlyeq c;$$

$$\langle [x]\rangle \preccurlyeq \langle [y]\rangle \text{ if } x \geqslant y \text{ (sic)}.$$

Of the three properties that are needed for a relation to be a partial ordering, namely (i) $s \preccurlyeq s$, (ii) if $s \preccurlyeq t$ and $t \preccurlyeq s$, then $s = t$, and (iii) $\preccurlyeq$ is transitive, parts (i) and (ii) follow from the requirements. Part (iii) does not, so $\preccurlyeq$ is defined as the transitive closure of the initial relation satisfying the requirements. Note that if we can find a minimum in a set of values according to this partial ordering, then the function # will be minimized as well. So if we accomplish what we are trying to do, we can forget about # and simply take $\downarrow_{first}$.

Once we start comparing according to $\preccurlyeq$, we hope to be comparing values of the form $first{*}ext_x\, u$. It is then helpful to have a function $repl_x$ such that

$$first{*}ext_x\, u = repl_x\, first{*}u.$$

By the usual method (unfold/fold), we find a solution:

$$repl_x\, b \Leftarrow \langle [x] \rangle \sqcup b\,;$$

$$repl_x\, (\langle [y] \rangle \sqcup b) \Leftarrow x < y \rightarrow \langle [x] \rangle \sqcup b\,.$$

If $\preccurlyeq$ captures indeed accommodatingness, we must now find

If $repl_x\, b \Rightarrow b_1 \,[]\, b_2$, *where* b_1 *and* b_2 *are both determinate, then* $b_1 \preccurlyeq b_2$ *or* $b_2 \preccurlyeq b_1$.

The proof is left to the indefatigable reader. It follows that $\preccurlyeq$ is total, if restricted to pairs of operands from $\mathsf{B}\, repl_x\, b$, so the meaning of $\downarrow/\mathsf{B}\, repl_x\, b$ is defined, where it is understood that a minimal element is selected according to the $\preccurlyeq$-ordering. Furthermore, we have

If $b_1 \preccurlyeq b_2$, *then* $\downarrow/\mathsf{B}\, repl_x\, b_1 \preccurlyeq \downarrow/\mathsf{B}\, repl_x\, b_2$.

Again, I leave the burden of proof to the undefeatable peruser, to whom I might as well dedicate the paper. (Seriously, I am interested in a snappy proof; one in the style of that of the 4CT I can generate myself.)

If we put

$$\begin{aligned} H &= first{*}{*}(\wedge/up{*}) \triangleleft (s \in shuffles) \triangleleft ([\,] \notin) \triangleleft \mathbb{U} \\ &= first{*}{*}\mathsf{B}\, upravel\, s\,, \end{aligned}$$

the constructive hypothesis is

$$first{*}u = \downarrow/H\,.$$

We want to show that $u' = \downarrow_{first}./\mathsf{B}\, ext_x\, u$, where u is treated as fixed (determinate), is a $first{*}$-minimizing element of $\mathsf{B}\, upravel\, ([x] +\!\!+ s)$ then. For arbitrary fixed u'', we have

$$first{*}\downarrow_{first}./\mathsf{B}\, ext_x\, u'' = \downarrow/first{*}{*}\mathsf{B}\, ext_x\, u'' = \downarrow/\mathsf{B}\, first{*}ext_x\, u'' = \downarrow/\mathsf{B}\, repl_x first{*}u''.$$

So

$$\begin{aligned} &first{*}u' = first{*}\downarrow_{first}./\mathsf{B}\, ext_x\, u = \downarrow/\mathsf{B}\, repl_x first{*}u = \downarrow/\mathsf{B}\, repl_x \downarrow/H \preccurlyeq \\ &\downarrow/\mathsf{B}\, repl_x \,[]/first{*}{*}\mathsf{B}\, upravel\, s = \downarrow/\mathsf{B}\, repl_x \,[]/\mathsf{B}\, first{*}upravel\, s = \downarrow/\mathsf{B}\, repl_x first{*}upravel\, s = \\ &first{*}\downarrow_{first}./\mathsf{B}\, ext_x\, upravel\, s = first{*}\downarrow_{first}./\mathsf{B}\, upravel\, ([x] +\!\!+ s). \end{aligned}$$

This is what we wanted to show.

The bottom line is that we can define $\oplus$ in the definition of su' by

$$x \oplus u \Leftarrow \downarrow_{first}./\mathsf{B}\, ext_x\, u\,.$$

Remember that ext_x was defined by

$$ext_x\, u \Leftarrow \langle [x] \rangle \sqcup u\,;$$

$$ext_x\, (\langle v \rangle \sqcup u) \Leftarrow x < first\, v \rightarrow \langle [x] +\!\!+ v \rangle \sqcup u\,.$$

If $(x <) \triangleleft first{*}u = \langle\,\rangle$, then the second part does not apply, so $x \oplus u = \langle [x] \rangle \sqcup u$. Otherwise, the first part is not of interest, since it extends the size of the upravel and so certainly does not yield a $first{*}$-minimizing result. If there are several matchings to the second part, it is easy to see that one yielding a $first{*}$-minimizing outcome is the one with a minimal choice for y, the replacee in $repl_x$. This makes it possible to define $\oplus$ so that it directly realizes a $first{*}$-minimizing extension.

In conclusion, the "program" is

$$su\, s \Rightarrow su'\, s\,;$$

$$su'\, [\,] \Leftarrow \langle\,\rangle\,;$$

$$su'\, ([x] +\!\!+ s) \Leftarrow x \oplus su'\, s\,;$$

$x \oplus u \Leftarrow x < r \rightarrow \langle [x] \rangle \sqcup u \;[]\; x \geqslant r \rightarrow x \oplus_r u,$
where $r = \downarrow / first * u;$

$x \oplus_r (\langle [r] +\!\!+ v \rangle \sqcup u) \Leftarrow \langle [x, r] +\!\!+ v \rangle \sqcup u .$

REFERENCES

1. J. W. HUNT, M. D. MCILLROY, *An Algorithm for Differential File Comparison.* CS Report TR 41, Bell Laboratories, Murray Hill, NJ, 1976.
2. D. S. HIRSCHBERG, A linear space algorithm for computing maximal common subsequences. *Comm. ACM 18* (1975) 341–343.
3. L. MEERTENS, Algorithmics—Towards programming as a mathematical activity. *Mathematics and Computer Science,* CWI Monographs Vol. 1 (J. W. de Bakker, M. Hazewinkel and J. K. Lenstra, eds.), North-Holland, $[]/(1984 \leqslant) \triangleleft \mathbb{U}$.

A COMMON BASIS

> The Boots and the Broker were sharpening a spade--
> Each working the grindstone in turn:
> But the Beaver went on making lace, and displayed
> No interest in the concern.

Somehow or other Richard picked up interest in my "squiggles" again (really his, if he had not disowned them). It cannot have been the general acclaim they met with at my presentations that made him do so. Maybe it was the ease with which I kept pulling functions and operators to the left or pushing them to the right (while writing the formulas upside-down) over a beer, even after many beers, that convinced him of the continued value of this approach.

Anyway, at the Pont-à-Mousson meeting a task force was set up to try and agree on a common notation, in which Richard and I were joined by David Wile. Way back in 1973, David had worked on a closely related approach to language design, using finite and infinite sequences (streams) as the semantic basis. (I was aware of his work all the time; I can remember that I was disappointed, when visiting CMU in 1975, to find him no longer there.)

Due to the fact that David could "join" us only by electronic mail, whereas Richard and I met in person for altogether a fair amount of time, his influence on the report as delivered by the task force has probably been much less than it otherwise would have been.

The paper was discussed in a quite fitting context: on a boat (going down the Amsterdam-Rhine Canal, and passing Breukelen during the presentation; fortunately, the rudder and the bowsprite did not become entangled during the voyage). This was in April 1985.

Next to correcting typographical and other silly errors, I have also updated the notation for filter here from ":" to "◁", and given the directed reduces a real arrow, instead of only an arrowhead.

IFIP WG2.1 Working paper ARK-3, 1985.

A common basis for algorithmic specification and development

Richard Bird
Lambert Meertens
David Wile

1. Introduction

This report, prepared for members of Working Group WG 2.1, summarises the results of two meetings between Lambert Meertens (LM) and Richard Bird (RB) held in Amsterdam on Jan 5–11, 1985 and in Oxford on Feb 15–25, 1985. David Wile (DW) was kept informed of the progress of these discussions by electronic mail, but due to difficulties with that medium, not all his contributions were received in time to incorporate into this summary.

Our initial objective in holding these meetings was to try to agree on a common basis, formulate general concepts, and suggest concrete notations for a—as yet unborn—Science of Algorithmics based on transformational programming. Not surprisingly, this ambitious programme of work was not realised. Although there was mutual agreement about the general importance and philosophy of the approach, there were differences of attitude about notational style and many areas where no firm conclusions could be reached in the time available. Indeed, it was felt early on that a reformulated objective would be more appropriate, namely to present to the Working Group concrete issues for further discussion, elaboration and refinement. By focusing attention on certain key areas, from semantic foundations to specific notations, we hope to provide input to WG 2.1 to continue its work.

2. Background

There is a style of algorithmic specification and derivation with the following commonly observed characteristics: it is based on simple notions of function definition and application, the functions are homomorphisms on structures, and derivations proceed, in part, by using general algebraic identities. Several people are actively pursuing transformational developments in this style. They feel that a more traditional 'mathematical' style of manipulation is appropriate for many steps in the treatment of algorithmic problems by transformations. We do not know yet how widely applicable such a style is; in particular whether it is only really suitable for small problems of a highly algorithmic flavour. However, for these kinds of problems at least it is certainly a valuable tool. One can express intricate transformations succinctly and substantial transformations precisely.

In spite of the observed similarities of approach mentioned above, there are also differences. There are variations in the concepts used, differences in notation for identical concepts, and wide divergence in general syntactical conventions. Since familiarity with notation is essential for ready comprehension of any transformational treatment of a problem, this disparity is a severe hindrance, and means must be sought to alleviate it.

Naturally, before one can discuss specific notations, one has to agree on a common framework and what predefined concepts are chosen for emphasis. These two aspects are closely related and mutually dependent, but for the purposes of organisation we have divided the rest of the report into four sections: Framework, Specific concepts, Notation and Laws.

3. Framework

Broadly speaking, the syntactic framework we envisage is an equational language of expressions involving functions, primitive objects and structures of various kinds. Certain equations between expressions are to be regarded, in a suitable sense, as the definition of new objects, functions, or structures. Possibilities for a concrete syntax are discussed in Section 5. The semantic framework is more problematic. One major candidate is a fixed-point semantics for a core language, together with transformational semantics for extensions. This is the CIP-L approach, at least in part. Alternatives are a purely algebraic semantics, or an axiomatic semantics based on some formal proof system.

The choice of a suitable semantics is dominated by two major considerations: the question of indeterminate values and the means of defining new generic structures, including infinitary structures such as is provided in a number of current functional languages. We deal with these in turn.

3.1. Indeterminacy

A long time was spent discussing the question of whether some notion of indeterminacy should be allowed in the expression language. For the purposes of discussion it was agreed to confine attention to the desirability or otherwise of using one or more types of choice operator in the specification and derivation of algorithmic expressions. In particular, we did not consider the problem of using indeterminacy to model concepts of concurrency. It was certainly felt that some form of indeterminacy enabled a number of derivations to be expressed more naturally at the element level rather than the set level. The importance of avoiding over-specification, allowing one to postpone design decisions until a suitable stage in the development process, was also recognised. However, it was also appreciated that any attempt to incorporate indeterminacy involves more or less severe problems of semantic description. Three possible approaches to indeterminacy were identified for future discussion by the Working Group:

(1) Avoid it altogether and encapsulate indeterminacy through the medium of set constructions alone. In this approach, exemplified in the treatment of the text formatter problem (Bird[2]), one just formulates the set of all solutions to the problem under consideration, and then selects a particular member of it by some further synthesis step. Questions of indeterminacy are resolved at the transformational level, and no notion of arbitrary choice is present in the expression language. The main advantage of this method is that the semantics of the expression language are simpler. On the other hand, it then becomes necessary to extend the definitional requirements for the functions and objects involved in the specification beyond the purely equational into general predicates about set membership. Also, the synthesis of the one-element-of step does tend to duplicate much of the reasoning about the all-elements-of solution.

(2) Allow an indeterminate-choice operator $\square$ into the notation, but let it always denote some definite, though as yet unspecified, operator. The only property of $\square$ one is allowed to assume is that it is selective. In other words

$$x \square y = sel(x, y)$$

for some suitable function *sel* that selects one of its arguments (the smallest, or the left-most, or the funniest). The advantage of this method is again semantic simplicity, but once one realises that different occurrences of $\square$ must all be given the same eventual interpretation, certain laws that one might deem desirable are no longer valid. For instance, the laws

$$x \square y = y \square x$$

and

$$f(x \square y) = (fx) \square (fy)$$

cannot be both valid. (Take f, x and y such that $x \neq y$, $fx = y$ and $fy = x$.) Different uses of $\square$ must be given different colours to ensure consistency. Also the notion of refinement $\Rightarrow$ is more problematic in this context. Nevertheless we believe it is possible to turn this device into a useful vehicle, and RB is currently attempting a clean treatment of the formatter problem based

on the approach.

(3) Allow $[]$ in the notation as denoting arbitrary choice, as recommended by LM. There are various subdivisions of such an approach: angelic versus demonic indeterminacy, the decision as to how to model functions (as single functions from objects to sets of objects, or sets of object to object functions), among others. The advantages of arbitrary choice have been illustrated in Meertens[3], but there are definite complications in giving a denotational semantics of higher-order functions and infinitary structures (both involving non-flat domains) based on fixed points.

Developing point (3), we considered what properties one would like of a choice operator. The following properties of $[]$ seem desirable, but are mutually incompatible:

(a) A well-defined notion of refinement ($\Rightarrow$) which is reflexive and transitive and such that all constructs are monotonic with respect to $\Rightarrow$.

(b) The property that $x \Rightarrow y$ iff $(x [] y) = x$.

(c) The laws that $[]$ is commutative, associative and idempotent; note that this, together with (b), implies reflexivity and transitivity of $\Rightarrow$, whereas, conversely, (b) together with reflexivity of $\Rightarrow$ would imply idempotence of $[]$.

(d) The law $x [] y \Rightarrow x$, which would follow from (b) and (c).

(e) The requirement of referential transparency. This is closely related to the validity of unrestricted unfolding of definitions. The difficulty in maintaining referential transparency is illustrated by the failure of such assertions as $(x [] y) - (x [] y) = 0$. Approach (2) above keeps this property.

(f'')The law $f(x [] y) = (fx) [] (fy)$, or its weaker counterpart in which $=$ is replaced by $\Rightarrow$. Note that the latter would follow from (a) and (b).

(g) The law $(x \Rightarrow y) \wedge (x \Rightarrow z)$ implies $x \Rightarrow (y [] z)$. This would follow from (b) and (c).

(h) The law $\forall x: fx \Rightarrow gx$ implies $f \Rightarrow g$. The other direction would follow from monotonicity. This law concerning the extensionality of refinement would seem particularly important.

The mutual incompatibility, even if (e) is dropped, is shown by the following example. Define

$$F\phi = (\phi 1) + (\phi 2);$$

$$fx = 3 + x;$$

$$gx = 6 - x;$$

$$hx = (fx) [] (gx).$$

We first show that $(Ff) [] (Fg) = 9$.

$$\begin{aligned} Ff &= (f1) + (f2) \quad \text{(by the definition of } F\text{)} \\ &= 4 + 5 \quad \text{(by the definition of } f\text{)} \\ &= 9 \quad \text{(by elementary computation).} \end{aligned}$$

Similarly,

$$Fg = (g1) + (g2) = 5 + 4 = 9.$$

So

$$\begin{aligned} (Ff) [] (Fg) &= 9 [] 9 \\ &= 9 \quad \text{(by c: idempotence).} \end{aligned}$$

On the other hand, we show that $(Ff) [] (Fg) \Rightarrow 8$. To show this, we need an auxiliary lemma:

$f [] g \Rightarrow f$ (by d).

$(f [] g)x \Rightarrow fx$ (by the above and a: monotonicity).

$(f [] g)x \Rightarrow gx$ (similarly, also using c: commutativity).

$(f \,[]\, g)\, x \Rightarrow (f\, x) \,[]\, (g\, x)$ (from the above by g)

$\Rightarrow h\, x$ (by the definition of h).

$f \,[]\, g \Rightarrow h$ (from the above by h).

Now we proceed:

$(F f) \,[]\, (F g) \Rightarrow F(f \,[]\, g)$ (by f: strong version)

$\Rightarrow F h$ (by the lemma and a: monotonicity)

$\Rightarrow (h\, 1) + (h\, 2)$ (by the definition of F)

$\Rightarrow (f\, 1) + (g\, 2)$ (by the definition of h and c and d)

$\Rightarrow 4 + 4$ (by the definitions of f and g)

$\Rightarrow 8$ (by elementary computation).

So the combination of (a–d) and (f–h) leads to the consequence that $9 \Rightarrow 8$, which is unacceptable, and at least one more desirable property must be dropped.

3.2. The definition of structures

We obviously need a coherent and simple way of combining old structures to make new ones. For the sake of simplicity we confined our attention to the single example of the tree–sequence–bag–set hierarchy of LM [3] and considered one possible approach to defining these structures. There are two points to bear in mind about the example:

(1) Unlike the data type constructors of, say, HOPE and Miranda, the structures form a hierarchy in which the associative, commutative and idempotent laws are introduced one by one. These laws thus ensure that each structure in the hierarchy is a refinement of the next higher one. Modulo these laws, each element structure is a free structure. Since free structures are not capable of further refinement by the imposition of new identities, any decision to make a structure free must be made explicit in the notation.

(2) Operations such as map and filter are not only polymorphic, they are also structure-generic, to coin a word. For example, map can be applied to sets, sequences, trees and bags. A major objective in trying to describe type hierarchies is to be able to introduce such generic operations at the right level of abstraction. An alternative approach was noted here but not pursued in any detail. Since map and reduce are examples of homomorphisms, it may prove possible to treat the specification of homomorphisms of structures as a primitive means of definition. We would like to see further discussion of this approach.

The particular line we investigated was the following. We define a 'system' to be a tuple (possibly a singleton) consisting of some (possibly none) types and some (possibly none) operations involving these types. As well as constant types, type variables are also allowed. A 'module' (for want of a better word) is a static function which may take a system as argument and returns a system as result. Thus module definitions are just like function definitions, and conform to the same general principles of syntax, except they are statically rather than dynamically evaluated. For example:

$$\text{module } \mathit{groupoid}(A) = (S,\ \hat{}: A \to S,\ \text{op } +: S \to S \to S)$$

defines a structure for each type A (itself a singleton system), providing a new type S, a function $\hat{}$ for injecting into the new type and $+$ as a binary operation on the type. (The type $S \to S \to S$ used here, to be parsed as $S \to (S \to S)$, is the 'Curry'd type corresponding to $(S \times S) \to S$. Notation and other matters of concrete syntax are still open questions; in particular, how one is to denote operators rather than prefix functions in the signature.)

A system can be built from another system by taking quotients modulo given laws. We suppose these laws might have to be restricted in some way—to positive conditional equations for instance—

but do not want to pursue this. As examples of the kind of refinements we have:

$$\text{law } \mathit{assoc}(\text{op } +: A \to A \to A) = (\forall\, x, y, z \in A: (x+y)+z = x+(y+z))$$

$$\text{module } \mathit{sequences}(A) = \text{free}((S, \hat{}, +) \text{ modulo } \mathit{assoc}(+) \text{ where } (S, \hat{}, +) = \mathit{groupoid}(A))$$

$$\text{module } \mathit{bags}(A) = \text{free}((S, \hat{}, +) \text{ modulo } \mathit{assoc}(+), \mathit{comm}(+) \text{ where } (S, \hat{}, +) = \mathit{groupoid}(A))$$

and so on. A type constructor, *Seq* say, can now be specified by a definition

$$(\mathit{Seq}(A), [\cdot], +\!\!+) = \mathit{sequences}(A).$$

This device allows one to choose different symbols for the operations of a type in different contexts.

We can now give system-generic definitions of operators. For instance

$$\text{module } \mathit{reduce}(\mathit{groupoid}(A)) = (\text{op } /: (A \to A \to A) \to S \to A) \text{ modulo } (\forall\, \otimes \in A \to A \to A,\ x \in A,\ s, t \in S: \otimes/\hat{}x = x \ \wedge \otimes/(s+t) = (\otimes/s) \otimes (\otimes/t)) \text{ where } (S, \hat{}, +) = \mathit{groupoid}(A)$$

In a particular context we might have something like

$$\cdots /\!/ \cdots \text{ where } /\!/ = \mathit{reduce}(\mathit{sequences}(A))$$

or even

$$\cdots /\!/ \cdots \text{ where } /\!/ = \mathit{reduce}(\mathit{Seq}(A), [\cdot], +\!\!+)$$

which is allowed since sequences are a groupoid.

We now list some queries about such an approach:

(1) Do we need final as well as initial (= free) models? (In an initial model, ground terms are unequal unless they are, by the laws, provably the same. In a final model, ground terms are equal unless they are provably different.)

(2) Should one allow restrictions on parameters? Are such restrictions part of the type? (Note that there is a problem in the definition of *reduce*: if, say, $\otimes/$ is applied to a sequence, the laws given imply that $\otimes$ is associative on the range of $\otimes/$. The laws suggest, however, that the function $\otimes/$ also exists for sequences if $\otimes$ is not associative. We could possibly have '$\forall\, (\otimes \text{ modulo } \mathit{assoc}(\otimes)) \in A \to A \to A$' in the definition of *reduce*, but then it would no longer be system-generic.)

(3) How is one to generalise the above to infinite structures, and how should one annotate the definitions to obtain all desirable combinations (finite alone, finite and infinite, or just infinite)? The question is related to the problem of which semantic framework is proposed for the concepts.

(4) Can one give other useful examples of this kind of structure hierarchy?

(5) In the examples, the structures have been defined without identity law for +, which would introduce the empty structure '0'. A generic definition of 'map' must have a law $f * 0 = 0$, but this would make $*$ unusable for structures without identity. In LM's approach to algorithmics, 'fictitious values' play an important role, such as $\downarrow/0$, the least element of an empty structure, which is an identity element for the operation $\downarrow$, corresponding to ∞. These are explained by domain extensions. Is it possible to have such extensions in the approach under discussion without tremendous fuss?

(6) In general, we must deal with partial functions and therefore with 'error values'. What is a

pleasant way of doing this?

(7) What are the ramifications of allowing other than equational laws? Note that 'free' introduces other laws already, so that the problem can even occur if all explicit laws are equational. Assume, for example, that a free system *natural* has been defined for natural numbers. Now consider:

$$\begin{aligned}&\text{module } natlet = (smallish{:}N)\\ &\qquad \text{modulo } (\forall\, n \in N:\\ &\qquad\qquad ppred(0) = 0 \ \wedge\\ &\qquad\qquad ppred(suc(n)) = n \ \wedge\\ &\qquad\qquad ppred(smallish) = 0)\\ &\qquad \text{where } (N, 0, suc) = natural\end{aligned}$$

In each model *smallish* is either 0 or 1, and presumably derivations are only valid if they are valid for both possibilities. But what are the exact semantics?

(8) The algebraic model of finitely generated terms breaks down for functions, whereas in the example of *reduce* the $\otimes$, although a function, was treated as an object. Intuitively, the meaning seems sufficiently clear. Is there some way of giving a precise semantic definition?

4. Specific Concepts

To a large extent the idioms of a language are dominated by what concepts one chooses to emphasise, even though others are easily definable. Below we suggest—in no particular order of importance—a number of specific concepts to be included in any framework for Algorithmics. Concrete notations are also suggested, but formal definitions within the expression language are avoided in many cases, basically because this would involve commitment to a particular syntactic style about which we would prefer to postpone discussion until the next section.

The list of concepts with 'predefined' notations should, preferably, be small and be confined to functions and operators that come up again and again in diverse derivations. If a function is less ubiquitous but still rather general and useful, and not easily expressible in terms of other predefined functions and operators, its inclusion may also be warranted. An important criterion is also whether there are associated laws that are helpful in derivations.

Whereas RB and LM feel that the predefined infix operators should preferably be single symbols, DW prefers longer operator names. Moreover, LM does not like predefined names that are English words.

(a) Map: $(A \to B) \to \text{Struct } A \to \text{Struct } B$.
An infix operator '$*$'. For example

$$f{*}[a, b, c] = [fa, fb, fc].$$

(Note: the type given for 'map' is the 'Curry'd version of $((A \to B) \times \text{Struct } A) \to \text{Struct } B$.) LM's notation '$\bullet$' was rejected because it is likely to be concretely represented as '.'. Since it will often occur at the end of a clause, it is too easily confused with the full stop. Remark: the map operator can be generalised to accept other structures built from one carrier type as right operand.

(b) Filter: $(A \to \text{Bool}) \to \text{Struct } A \to \text{Struct } A$.
An infix operator '$\triangleleft$'. As examples

$$even \triangleleft \{1, 2, 3\} = \{2\},$$

$$odd \triangleleft [1, 2, 1, 3] = [1, 1, 3].$$

(c) Reduce: $(A \to A \to A) \to \text{Struct } A \to A$.

An infix operator '/'. For instance

$$\otimes/[a, b, c] = a \otimes b \otimes c.$$

See [3] for further details.

(d) Left accumulate: $(D \to R \to R) \to R \to \text{Seq } D \to R$.
Right accumulate: $(R \to D \to R) \to R \to \text{Seq } D \to R$.
Infix operators '$\not\leftarrow$' and '$\not\rightarrow$'. For instance

$$\otimes \not\leftarrow e\ [a, b, c] = a \otimes (b \otimes (c \otimes e)),$$

$$\otimes \not\rightarrow e\ [a, b, c] = ((e \otimes a) \otimes b) \otimes c.$$

We do not particularly like the notations but cannot think of better alternatives. The 'starting element' e can be placed in subscript position whenever convenient, that is, one can write $\not\rightarrow_e$ and $\not\leftarrow_e$. These operators give 'asymmetric forms' of 'reduce'. In particular, if $\otimes/s$ and $e = \otimes/[\,]$ are defined, then $\otimes \not\rightarrow_e s = \otimes \not\leftarrow_e s = \otimes/s$.

(e) Specific structure building operations.

Suppose $(0, \hat{}, +) = \textit{groupoid}(A)$.

generic name	0	$\hat{}$	+
sets	$\{\}$	$\{\cdot\}$	$\cup$
bags	$\langle\rangle$	$\langle\cdot\rangle$	$\sqcup$
sequences	$[\,]$	$[\cdot]$	$+\!\!+$

There are two reasons for having structure-specific names next to the generic names. One is that frequent dictions like $+/\hat{}*$ or $1 \hat{+} 2 \hat{+} 3$ are ambiguous if the result type of $\hat{}$ is not specified. In unambiguous cases the additional redundancy may aid the interpretation. The second is that writing $1 \hat{+} 2 \hat{+} 3$ is more awkward than writing [1, 2, 3]. The generic names can be used instead of the structure-specific names if one wishes no commitment as to the specific structure, or if no confusion can arise.

We would have liked a notation in which there is a simple relationship between the graphic symbols for the brackets and those for $\cup$ etc. We could, however, not find a nice set of symbols with such properties. The best we could come up with is the set

$()$ $\cup$
$[\,]$ $\sqcup$
$\langle\rangle$ $\vee$

This would have meant giving up using the round parentheses for normal grouping, which we deemed unacceptable. It would also be nice if symbols for structure inclusion could be derived from the structure-specific forms of $+$. Whereas the relationship between the signs '$\cup$' and '$\subseteq$' suggests a notation for bag inclusion too, an extension to sequences based on '$+\!\!+$' is impossible.

RB also suggests ';' as the Lisp cons, i.e. $a;x = [a] +\!\!+ x$. Although LM is not averse to predefining an operator for this oft-occurring diction, he feels that we must then also supply a notation for $x +\!\!+ [a]$, and that the graphics for the two operators should be each other's mirror image. Moreover, he is loathe to give up his parenthesis-dispelling use of ';'.

(f) Repeat: $(A \to B) \to A \to \text{Seq } B$.
A prefix function. For instance

$$\text{repeat } f\, x = [x, fx, f(fx), \cdots].$$

This function makes it possible to build infinite sequences. An infinite sequence of 1's, e.g., is given by repeat id 1. The result of the 'program'

$$\mathbf{do}\ p\,x \rightarrow x := f\,x\ \mathbf{od};\ x$$

can be written as *first* $(\neg p) \triangleleft$ repeat $f\,x$, in which *first* gives the first element of a sequence. However, the equivalence becomes dubious if f becomes undefined for arguments for which p does not hold.

LM would prefer 'rep' to 'repeat', and DW would prefer an infix operator.

(g) While: $(A \rightarrow \mathit{Bool}) \rightarrow \mathrm{Seq}\,A \rightarrow \mathrm{Seq}\,A$.
A prefix function. For instance,

$$\text{while } \mathit{even}\ [2, 4, 2, 1, 2] = [2, 4, 2].$$

If p is total, while $p\ s$ can be expressed, using the function α (see (t)), as $\mathit{last} * ((\wedge/) \circ (p *)) \triangleleft \alpha\, s$.

Meertens doesn't like 'while', while Wile would prefer an infix operator. This is also reasonable because of the resemblance to the filter; see (b).

(h) Tuples (elements of Cartesian product).
We suggest ',' as a syntactic n-ary infix operator, where $n \geqslant 2$. There are no specific tuple brackets (the '‹' and '›' have already been given to bags), but this syntactic operator has a very low priority, so that one is forced to write parentheses in almost all positions, and certainly between structure-parentheses like '{' and '}'. Thus (a, b, c) is a triple, and (a, b, c, d) a quadruple. As projection functions we can think of no better notation than π_1, π_2, etc., or just 'fst', 'snd', etc.

(i) Zip: $\mathrm{Seq}\,A \rightarrow \mathrm{Seq}\,B \rightarrow \mathrm{Seq}\,(A \times B)$.
An infix operator, notation not decided. For example,

$$[a, b, c]\ \mathrm{zip}\ [x, y, z] = [(a, x), (b, y), (c, z)].$$

The operands must have equal length. Alternatively, the length of the result might be that of the shorter of the two sequences. Although the binary case is the most frequent one, it is possible to generalise this to an n-ary operator. A possible notation might then be ',,'. The notation for this concept in [4] and [5] is 'with'.

(j) Transpose: Struct (Seq A) $\rightarrow$ Seq (Struct A).
A prefix function 'trans'. For instance

$$\mathrm{trans}\ ⟨[a, b, c], [d, e, f]⟩ = [⟨a, d⟩, ⟨b, e⟩, ⟨c, f⟩].$$

All elements of the argument must have equal length. If the argument is a sequence s, then trans trans $s = s$. Alternatively, we might have, e.g.,

$$\mathrm{trans}\ ⟨[a], [b, c, d], [e, f]⟩ = [⟨a, b, e⟩, ⟨c, f⟩, ⟨d⟩],$$

that is, the structure of all first elements, followed by the structure of all second elements, etc. To save the property trans trans $s = s$ for sequences, we should then require that the lengths of the elements of s form a non-increasing sequence.

(k) Composition: $(B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C)$.
An infix operator $\circ$. We have $(f \circ g)x = f(gx)$. See also section 5.

(l) Length (size): Struct $A \rightarrow \mathbb{N}$.
A prefix function '#', so $\#[a, b, c] = 3$.

(m) Closure.
We feel this is an important function, but are in a quandary as to the exact type the function should possess. The two possibilities are

(1) $(\mathrm{Set}\,A \rightarrow \mathrm{Set}\,A) \rightarrow \mathrm{Set}\,A \rightarrow \mathrm{Set}\,A$;
(2) $(A \rightarrow \mathrm{Set}\,A) \rightarrow \mathrm{Set}\,A \rightarrow \mathrm{Set}\,A$.

For the former, $\mathrm{Clo}_1\ f\ S = S \cup (f S) \cup (f(S \cup (f S))) \cup \cdots$. For the latter, $\mathrm{Clo}_2\ f\ S = S \cup (\cup/f * S) \cup \cdots$. The first possibility is more general; in particular, $\mathrm{Clo}_2\ f\ S = \mathrm{Clo}_1\ ((\cup/) \circ (f *))\ S$. On the other hand this is somewhat awkward, since in most

applications we encounter the second case. Also, $\mathrm{Clo}_1\, f$ can be expressed as $(\cup/) \circ$ (repeat g) (see (f)), where $g\ S = S \cup (f S)$.

(n) Optimisation functions.
For minimisation and maximisation we suggest infix operators '↓' and '↑'. For instance,

$$(3 \downarrow 4) \uparrow (5 \downarrow 6) = 5.$$

The function $\downarrow/$ selects the least element of a structure. These two operations are coupled to whatever order relation '$\leqslant$' is defined on the operands domain. A question is whether these operations should be generalised to work on (semi-)lattices, with '↓' standing for '$\wedge$', and '↑' for '$\vee$'.

We would also like to have functions for 'a (or all) f-minimising (-maximising) element(s)'. This awaits resolution of the treatment of indeterminacy.

(o) Conditional.
A $2n$-ary operator pair, as in $p \to x \,[]\, q \to y \,[]\, r \to z$. If no indeterminacy is allowed, the guards must be mutually insatisfiable, or the expressions following '$\to$' must have the same value. Note: these operators are supposed to have low syntactic priority, so that we can write $x \leqslant y \to x+1 \,[]\, x \geqslant y \to y+1$.

In view of the frequency of forms like $p \to x \,[]\, \neg p \to y$ (in which p may be a complicated expression), some notation for 'if ... then ... else' would be nice, like an asymmetric 'choice operator', say $[]:$, so that we can write $p \to x \,[]: y$.

(p) K combinator.
This is a function for turning a value v into the constant function $\lambda x\colon v$, which is often needed. If the infix operator $\ll$ selects its left operand, then we could use the notation $v \ll$ (a 'section'), as in [3]. However, we do not like this notational trick, nor do we like 'K' itself.

(q) Switching operands to operator.
A notational device: $x \otimes_z y = y \otimes x$. For example: *reverse* $s = +_z/[\cdot]{*}s$. The same device could be used for functions too: $f_z\, x\ y = f\, y\, x$. An alternative suggestion is to use the notation $\otimes^{\sim}$, or possibly $\tilde{\otimes}$, which should be more familiar to mathematicians.

(r) Turning operators to functions.
Another notational device: $(\otimes)\, x\ y = x \otimes y$. DW is quite averse to this notation, and recommends the use of an explicit conversion operator.

(s) Turning functions to operators.
This is useful for supplying an operand to operators like $/$ or $\not/$. Simply allow $f/$ to mean $\otimes/$, where $x \otimes y = f\, x\ y$.

(t) Initial parts of sequence: Seq $A \to$ Seq(Seq A)).
A prefix function. For instance

$$\alpha[a, b, c] = [[a], [a, b], [a, b, c]].$$

Note that the initial empty sequence is not included, so that $\# \circ \alpha = \#$.

Question: do we also need final parts:

$$\omega[a, b, c] = [[a, b, c], [b, c], [c]]?$$

(u) First, last: Seq $A \to A$.
Tail, head: Seq $A \to$ Seq A.
For first and last element of a sequence we could use $\ll/$ and $\gg/$, as in [3]. On the other hand, we would like suggestive pairs of notations for (first, tail) and (head, last), where (first s)⧺(tail s) = (head s)⧺(last s) = s for non-empty sequences s.

It is understood that existing standard mathematical notation is allowed as well, although some care must be exercised lest confusion arise between different sets of conventions. In particular, we have:

- Arithmetic expressions, like ax^2+bx+c. However, the symbols '$*$' and '$/$' have been preempted for algorithmic purposes; possible alternatives are '$\cdot$' and '$\div$'. A potential source of confusion is

also the parsing of $x + y - z$.

- Set formers, like $\{(i, j) \mid i \in \mathbb{N}, j \in \mathbb{N} : 1 \leqslant i < j \leqslant n\}$. Note that $\{f x \mid x \in s : p x\}$ can be written as $f * p \lhd s$, and it is recommended that set formers be used only if no convenient such expression exists. We also recommend the use of forms like $\{1 .. n\}$. Also, set formers can be generalised to other structure formers by using other brackets, as in SETL: $[(\#s, \gg/s) \mid s \in \alpha t : s = \mathit{reverse}\ s]$.
- Lambda forms, such as $\lambda n \in \mathbb{N} : (n \cdot (n + 1)) \div 2$. If the domain is sufficiently clear, we can abbreviate this to $\lambda n : (n \cdot (n + 1)) \div 2$.
- Quantifications, such as $\forall n \in \mathbb{N} : f n > 0$. Note that $\forall x \in s : p s$ can be written as $\wedge / p * s$.

The following concepts were considered, but rejected for separate predefined notation:

- Scan (the '\' of APL): use $\otimes / \alpha$.
- Limit (repeat until convergence): use closure if appropriate (see (m) above).
- Shuffle (indeterminate merge) of sequences: probably not important enough for inclusion.
- Unless, until: use $(\neg \circ p) \lhd$ and $\mathit{while}(\neg \circ p)$.

Not discussed, but possibly important are: (a) a notation for finite maps (functions with finite domains) and an operator for function update; (b) a sort of Cartesian product for structures (named 'cross' in [5]); for example $\{a, b, c\} \times [d, e] = \{[(a, d), (a, e)], [(b, d), (b, e)], [(c, d), (c, e)]\}$; (c) an explicit notation for function application, say 'appl', as in $\mathrm{appl} * ([f, g, h]\ \mathrm{zip}\ [x, y, z]) = [f x, g y, h z]$; (d) notations for 'pattern matching', binding variables in a pattern to actual values; both in function definitions, in 'where' clauses and in conditions; (e) other devices that can replace the awkward projection functions.

5. Notation

The question of concrete syntax for expressions was debated at length during our meetings and was the source of much (good-natured) conflict. Since matters of syntax are to a large extent questions of personal taste and cultural background, it seems appropriate to preface this section with some historical remarks.

In 1973 DW had developed a language emphasising the relationships between program structures and data structures, with a small but powerful set of operators, mainly directed towards (possibly infinite) streams and trees [4]. In particular, we find 'sections' there as a useful device. This work has not had any noticeable immediate follow-up. In 1981 RB presented to the Nijmegen meeting of WG2.1 some notational suggestions for transformational programming [1]. Apart from reinventing map and filter and suggesting concrete operator notations, the proposed syntax was rather free wheeling. Subsequent presentations at various UK universities convinced him that the syntactical conventions were totally alien to audiences, prone to ambiguity and generally unworkable. RB then adopted an alternative notation broadly in agreement with Turner's suggestions for KRC (henceforth called KRC style). Meanwhile LM took RB's original proposal, modified and refined it, and gave an unambiguous grammar. LM has since used this syntactic style in a number of publications and presentations, and found it convenient, succinct and transformation friendly. RB, working with the KRC-style, came to similar conclusions with that notation. During our meetings we tried to resolve this situation, but were unable to do so in a satisfactory manner. With the insertion of one or two brackets the terminal productions of the KRC-style appear to be a strict subset of the LM style, but the problem is that common terminal productions are sometimes assigned different meanings in the two styles. There is also the question of taste. LM prefers to carry out manipulations at the function level, to which his notation is more suited. In particular, the use of an explicit composition operator can be avoided whenever the context makes it clear which interpretation is intended. RB prefers to carry out many derivations at the point level, with only occasional references to functional identities. Experiments were performed trying to do essentially the same derivation in the two styles. The conclusions were interesting: in neither derivation did brackets proliferate, but this was entirely a consequence of the point versus function decision. The rest of this section is devoted to short expositions of the two alternative proposals, followed by a short exposition of DW's preferred style, with an

appraisal of their respective merits.

Bird's Proposal. Ignoring productions involving explicit set, tuple and sequence constructions, the proposed syntax follows these rules:

expression ::= term {op term} | term op | op term | op
term ::= {primary} primary
primary ::= constant | identifier | (expression)

The notation '{...}' stands here for zero or more repetitions of the enclosed part, whereas the '(' and ')' are literals. An 'op' is an operator.

With the exception of sections—expressions of the form 'term op', 'op term' and 'op'—such a syntax conforms to the generally recognised KRC style. Function application (a 'term' applied to a 'primary') is left associative and binds tightest; the other operators are all right associative and of equal binding power (but lower than application). For example, $fx + y$ is parsed as $(fx) + y$. The alternative of providing explicit precedence rules between operators is not precluded.

A section like $x +$ stands, of course, for the function $\lambda y\colon x + y$, and $+y$ stands for $\lambda x\colon x + y$. Used by itself, $+$ stands for $\lambda x\colon \lambda y\colon x + y$.

Advantages of this style are: (a) simplicity; (b) familiarity to programmers of the KRC persuasion; (c) type independence, by which is meant that the rules of syntactic composition are not dependent on the types of the components.

Possible disadvantages include: (a) the composition operator ∘ has at all times to be made explicit; (b) operators which take operators as left arguments (such as reduce) have to be written with those operators in brackets—for example, $(+)/$ rather than $+/$; (c) sections have to be bracketed—e.g., $(foo*)*$ rather than $foo**$. Note here that '(op)' denotes a prefix function, not operator, so that, e.g., $(+)\ x\ y$ is meaningful and denotes $x + y$. It follows that operators now never take operators as left arguments, only functions. Hence greater flexibility is achieved and one can write, e.g., *sort* = *insert*$\not$[·], where *insert* is a function with its usual KRC definition.

Meertens' Proposal. The proposed syntax for the same part of the language can be given as:

expression ::= [op] {factor} factor | op
factor ::= primary {other-op}
primary ::= constant | identifier | op op-op | (expression)
op ::= op-op | other-op

The same notation as above applies, and, moreover, '[...]' denotes an optional occurrence of the enclosed part.

This is the same notational style as has been used in previous examples by LM, but with sections of the form $+y$ added. The meaning of an 'apposition' $f\ x$ depends on the types of f (which must be a function) and x (which could be a function). If x is a meaningful argument to f, then the meaning is f applied to x. Otherwise, the meaning is f composed with x. This has some theoretical background, insofar as a non-function x could be viewed as a 0-ary function, and then functional composition yields the 0-ary function corresponding to f applied to x. (The word 'apposition' can be understood not only in its usual meaning of 'juxtaposition', but also as a portmanteau word for 'application'+'composition'. It has been pointed out that a more apposite portmanteau might be 'complication'.)

The operators are divided into two disjoint classes: 'op-op' for operators taking an operator as left operand (notably '/'), and 'other-op' for the other operators (like '+'). By allowing 'primary', next to 'op', for the left-operand of 'op-op's, it is likewise possible to allow *insert*$\not$[·]. A formula like $x + y$ is allowed, but although it has the same meaning as one would expect, it has the unexpected parsing $x +$ applied to y.

The major difference with Bird's approach is that the expression $fx + y$ is likewise allowed, but this time has the same meaning as $f(x + y)$. Also, $fg\,x$ is allowed with meaning $f(g\,x)$, which in

Bird's approach would require the parentheses.

Advantages of this style are: (a) greater 'substitutivety' and fewer trivial derivation steps: from a functional identity $f = g$ one may conclude that $fx = gx$ without having to change the syntactic form of f or g, and usually the step from $x = y$ to $fx = fy$ requires no syntactic changes either; so, for example, $f = gh$ and $hx = y$ imply $fx = gy$ in fewer steps than $fx = (g \circ h)x = g(hx) = gy$; (b) although $+/ \ast\ast +/x$, e.g., may indeed be abstruse and require familiarity of this style to be interpreted, it is still felt to be more readable than $((+)/\ast)\ast(+)/x$ or $((((+)/\ast)\ast)\circ((+)/))x$, which hide the patterns involved rather than disclose them, and are, moreover, tedious to write if they have to be copied in derivation steps.

Possible disadvantages include: (a) the grammar is more complicated; (b) people familiar with the KRC style are apt to misinterpret $fx + y$ as $(fx) + y$—but on the other hand, the converse applies to Birds' approach for APL-fandom; (c) without contextual knowledge, in particular the types involved, fx cannot be interpreted (composition or application?); for example, an expression $f(gx)y$ can, depending on the types of f and g, stand for either $f((gx)y)$ or $(f(g(x))y$, and due to generic types an explicit composition operator may be required for genuinely ambiguous cases.

Wile's Proposal. Rather than giving a syntax description here, let it suffice to say that DW prefers postfix functions. A possible grammar might be the mirror image of LM's grammar. For example, $+/p \lhd f \ast s$ would become $s \ast f \lhd p/+$, or rather s obtain f when p accrue $+$. For some operators, the operands would stay as they are; in particular so for $\rightarrow$ (conditional, see (o) in section 4). Prefix function like 'repeat' and 'while' (see (f) and (g) in section 4) would become infix operators. For more examples of the style, see [5].

Advantages of this style include: (a) the parsing is from left to right again and so is more natural; in particular, a formula like $a + b - c$ regains it usual meaning; (b) LM's syntax favours sections in which the operand precedes the operator, and so makes one use for instance $0>$, although the section <0 would be more natural; in this style, the latter becomes the favoured form; (c) using longer operator names gives a considerable help to human interpreters.

Possible disadvantages are: (a) it is one more step away from mathematical tradition; (b) longer operator names tend to get tedious in transformational developments.

6. Laws

Unfortunately, time did not permit us to draft this section.

References

1. R. S. Bird. *Some notational suggestions for transformational programming.* WG 2.1 working paper NIJ-3 (unpublished), 1981.
2. R. S. Bird. *Transformational derivation of a text formatter.* (unpublished), 1984.
3. L. Meertens. *Algorithmics—Towards programming as a mathematical activity.* WG 2.1 working paper ADP-3, 1984. To appear in: *Mathematics and Computer Science,* CWI Monographs Vol. 1 (J. W. de Bakker, M. Hazewinkel and J. K. Lenstra, eds.), North-Holland, []/1985≤◁U.
4. D. S. Wile. *A Generative, Nested-Sequential Basis for General Purpose Programming Languages.* CMU Dept. of Comp. Science, Pittsburgh, 1973.
5. D. S. Wile. *Generator Expressions.* WG 2.1 working paper ADP-8, 1984.

TWO EXERCISES FOUND IN A BOOK ON ALGORITHMICS

So engrossed was the Butcher, he heeded them not,
As he wrote with a pen in each hand,
And explained all the while in a popular style
Which the Beaver could well understand.

The last paper of this reader was first presented on the April 1985 boat, and again, exactly one year later, at the Bad Tölz Working Conference on Program Specification and Transformation.

The title reflects that old guiding concept in the search for the genuine Abstracto: an advanced book on algorithmics. The exercises here are rather elementary, of course, but one must start somewhere.

If the algorithmic developments here are eminently more readable than in the other papers, this is not only due to the greater simplicity, but also or mainly to Richard's influence. You see, I will always remain something of a hacker, I'm afraid, whether I write my programs the "old" way or develop them formally.

Program Specification and Transformation
L.G.L.T. Meertens (Editor)
Elsevier Science Publishers B.V. (North-Holland)

Two Exercises Found in a Book on Algorithmics

Richard S. Bird
Programming Research Group
University of Oxford, UK

Lambert Meertens
Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

1. Introduction.

A major test of a good notation is how suggestive it is of new relationships between the objects described, and how susceptible it is to the manipulations required to establish such relationships formally. Indeed, if the associated calculus is sufficiently attractive to use, new relationships can come to light simply by the process of manipulating formulae.

The term 'algorithmics' was coined in [Geurts & Meertens 2]: 'Suppose a textbook has to be written for an advanced course in algorithmics. Which vehicle should be chosen to express the algorithms? Clearly, one has the freedom to construct a new language, not only without the restraint of efficiency considerations, but without any considerations of implementability whatsoever.' It was elaborated upon in [Meertens 3], and stands now for both a notation and a calculus for manipulating algorithmic expressions, designed with the aim of meeting the criteria enunciated above. Algorithmics corresponds, broadly speaking, to what is currently known as Transformational Programming, but the level of abstraction is arguably higher than one would normally encounter, and a wide range of specific notation is emphasised. The subject is still in its infancy and it is not the purpose of the present paper to give a comprehensive account. Instead we want to present, in as simple and direct a fashion as possible, two exercises in manipulation as they might appear in some future book on Algorithmics. If, in studying these problems and their solutions, the reader is by turns puzzled, intrigued and finally enlightened as to the real possibilities for a useful calculus of algorithm derivation, then we shall have achieved what we set out to do.

In order to describe the two problems without further preamble, it is necessary to refer to certain concepts without giving them a formal definition; consequently the statement of exactly what is provided and what is required will not be very precise. The first part of the paper is devoted to developing enough of the calculus of algorithmics to remedy this deficiency. We shall then be in a position in the last two sections both to state the two problems precisely and solve them simply by a process of formula manipulation.

Problem 1. The reduction operator '/' of APL takes some binary operator $\oplus$ on its left and a vector x of values on its right. The meaning of $\oplus/x$ for $x = [a, b, \ldots, z]$ is the value $a \oplus b \oplus \cdots \oplus z$. For this to be well-defined in the absence of brackets, the operation $\oplus$ has to be associative. Now there is another operator '\' of APL called 'scan'. Its effect is closely related to reduction in that we have

$$\oplus\backslash x = [a, a \oplus b, a \oplus b \oplus c, \ldots, a \oplus b \oplus \cdots \oplus z] .$$

The problem is to find some definition of scan as a reduction. In other words, we have to find some function f and an operator $\otimes$ so that

$$\oplus\backslash x = (fa) \otimes (fb) \otimes \cdots \otimes (fz).$$

Problem 2. This problem was suggested to us by Phil Wadler. Define a line to be a sequence of characters not containing the newline character NL. It is easy to define a function *Unlines* that converts a non-empty sequence t of lines into a sequence of characters by inserting newline characters

between every two lines of t. Indeed, *Unlines* can be written as a simple reduction as described in Problem 1. Since *Unlines* is injective, the function *Lines*, which converts a sequence of characters into a sequence of lines by splitting on newline characters, can be specified as the inverse of *Unlines*. The problem, just as in Problem 1, is to find a definition by reduction of the function *Lines*.

It is worth remarking that neither problem, both of which are fairly simple to solve, is just an academic exercise of no practical relevance; both illustrate quite serious and important concerns in computation. The former seeks to replace a quadratic time algorithm with a linear one, while the latter is an instance of the problem of finding a computationally effective definition of some operation, given only an effective definition for its inverse. This problem arose in interactive text-formatting.

2. Notation

Our formulae will be (equations between) expressions. The class of allowable expressions (actually, a simplified version) is described below. Certain equations are taken as definitions of the operator or function appearing on the left hand side; these equations may define the function recursively. As simple examples we have

$$fact\ n = \times/[1..n]$$

and

$$fib\ 0 = 0$$

$$fib\ 1 = 1$$

$$fib\ (n+2) = fib\ (n+1) + fib\ n.$$

The class of expressions is given by the following BNF syntax, in which the meta-brackets '{' and '}' signify zero or more occurrences of the enclosed part, and signs in double quotes are literals.

expression ::= term {op term} | term op | op term | op

term ::= {primary} primary

primary ::= constant | identifier | "(" expression ")" | "[" expression-list "]"

expression-list ::= | expression {"," expression}

Here, 'op' is short for 'operator'. We shall use symbols such as $\oplus$, $\otimes$, $/$ and $\cdot$ to denote operators. The precedence rule between operators is that all have the same precedence and all are right associative. (So $a \oplus b \otimes c$ means $a \oplus (b \otimes c)$.) This is the convention adopted in [Bird 1]. Function application is denoted by a space; this operator is of higher precedence than others and is left associative. (So $f\ a\ b \otimes c$ means $((f\ a)\ b) \otimes c$.) Again this is the convention adopted in [1] and is one that is familiar to many functional programmers, being copied from Turner's KRC [4].

The forms (term op), (op term) and (op) are known as sections. If they stand alone, no brackets are needed. A presection, of the form $(x \otimes)$, is a prefix function with the property

$$(x \otimes)y = x \otimes y.$$

A postsection, of the form $(\otimes x)$, is a prefix function with the property

$$(\otimes x)y = y \otimes x.$$

A full section, of the form $(\otimes)$, is a prefix function with the property

$$(\otimes)x\,y = x \otimes y.$$

Certain operators, for example the reduction operator '/' of Problem 1, expect operators as their left argument. However, productions such as $\otimes/$ are not allowed by the foregoing grammar, and one has to write $(\otimes)/$. It is a harmless abuse of notation to permit the brackets to be dropped in such a situation, and we shall henceforth do so.

This is all there is to say about syntax. The equal precedence and right-associative rule for all operators except application takes a little getting used to, but turns out to be very convenient, at

least for formulae not involving common arithmetic operations, where other precedence rules are deeply ingrained.

3. STRUCTURES AND HOMOMORPHISMS.

As was stated before, $\oplus/[a, b, \ldots, z] = a \oplus b \oplus \cdots \oplus z$, in which no brackets are needed if $\oplus$ is associative. Henceforth we shall require $\oplus$ to be associative if used in reductions. This means that it is unnecessary to specify the order in which the reduction is performed: left to right, right to left and recursive computation by splitting in two halves all yield the same result. Such 'underspecifications' are generally helpful in algorithmic developments, since they allow one to explore various strategies. In fact, it may be argued that the imposition of an order if it is irrelevant is an *over*specification, an undue commitment that may stand in the way of a useful transformation.

Since the computation order for reduction by an associative operator $\oplus$ is immaterial we can give a symmetric recursive characterisation of $\oplus/$, which can be taken as its formal definition. Let $+\!\!+$ stand for sequence concatenation, so $[a, b] +\!\!+ [c, d, e] = [a, b, c, d, e]$. Also let the identifiers x and y stand for arbitrary sequences. Then

$$\oplus/[a] = a;$$

$$\oplus/(x +\!\!+ y) = (\oplus/x) \oplus (\oplus/y).$$

If the operation $\oplus$ has a unit, then reduction of an empty sequence, $\oplus/[\,]$, stands for that unit. Otherwise, such a reduction is undefined and x and y must not be empty.

The sequence concatenation operator $+\!\!+$ is associative. Associativity is also the requirement on $\oplus$ for $\oplus/$ to be meaningful. This is, of course, not a coincidence. We may, likewise, define reduction over a set, so that $\oplus/\{a, b, \ldots, z\} = a \oplus b \oplus \cdots \oplus z$. The formal definition is similar to the one for sequences, with $\{\cdot\}$ replacing $[\cdot]$ and $\cup$ replacing $+\!\!+$. Not only is set union associative, but also commutative and idempotent. These are precisely the requirements that $\oplus$ has to meet in order to make the given definition of reduction over sets unambiguous. In general, one may consider *structures* that are built by taking singletons from some domain and by applying a binary construction operation to previously erected structures. In the absence of specific properties for the construction operator, we obtain the set of binary trees whose leaves are labelled with atoms. As we have seen, familiar algebraic properties give other familiar data structures: sequences and sets. The algebraic properties of associativity and commutativity together yield yet another familiar data structure: bags or multisets. This means we can give a generic definition of reduction, and also obtain generic algorithmic laws and developments that have, as yet, no commitment to a choice of specific data structure. As the examples in the present paper are only concerned with sequences, this point will not be elaborated upon here.

The identity law (which is an algebraic property that the construction operation may, or may not, have) corresponds to the empty structure (tree, sequence, bag or set, as the case may be). It may happen, and indeed it often does, that an operation $\oplus$ has no unit, but that an algorithmic development naturally leads to forms $\oplus/x$ in which x may be empty. This is a common nuisance that would require special measures to cater for 'exceptional' cases, causing complication of the algorithmic specifications under consideration, which has to be dragged along in the development. Fortunately, in many cases it is possible to employ an expedient stratagem: extend the domain of $\oplus$ with a 'fictitious value', an adjoined element, that assumes the role of the missing unit. For example, the binary operation of taking the minimum value of two operands has no unit in the domain of real numbers. By adding a fictitious value, which we might call ∞, we can assign a meaning to the minimum reduction of an empty structure. This can help to simplify algorithmic developments, and sometimes very much so. It is not uncommon that such fictitious values only have an ephemeral role in a derivation. This is similar to the mathematical 'trick' of solving problems concerning real numbers through a temporary excursion into the complex domain.

Another high-level operation is the map operator '$\cdot$', which takes a function on its left and a sequence, or in general a structure, on its right and replaces each element by its image under the function. For example, we have $f \cdot [a, b, \ldots, z] = [fa, fb, \ldots, fz]$. As for reduction, we can give a recursive characterisation:

$$f \cdot [a] = [fa];$$

$$f\bullet(x +\!\!+ y) = (f\bullet x) +\!\!+ (f\bullet y).$$

For an empty sequence, we must have $f\bullet[\,] = [\,]$.

DEFINITION. A function h defined on sequences over a given domain is a 'homomorphism' if there exists an operation $\oplus$ such that, for all sequences x and y over that domain:

$$h(x +\!\!+ y) = h\,x \oplus h\,y.$$

If h is defined on empty sequences, then, moreover, $h[\,]$ must be the (possibly fictitious) unit of $\oplus$. The generalisation to other structures is obvious. Both reductions and maps are examples of homomorphisms. This is immediate from their recursive definitions. For a given homomorphism h, the operation $\oplus$ is uniquely determined (on the range of h) by h, and we shall refer to it as 'the' operation of h. The operation of a reduction $\oplus/$ is, of course, $\oplus$. That of a map $f\bullet$ is $+\!\!+$. Not all functions on structures are homomorphisms. A counterexample is the 'number of elements' function # on sets. On sequences and bags, however, # is a homomorphism. Moreover, all injective functions are homomorphisms. The importance of homomorphisms is essentially the same as mentioned before for reductions: they allow a variety of computational strategies, among which such important paradigms as iteration (left-to-right construction) and divide-and-rule. The assumption is that $\oplus$ (and h on singleton structures) are relatively cheap to compute. The formulation of a function as a homomorphism shows then also how to develop an incremental approach, as in formal-difference methods.

Although there are other homomorphisms than reductions and maps, these can be viewed as the stuff homomorphisms are made of:

HOMOMORPHISM LEMMA. *A function h is a homomorphism if and only if there exist an operation $\oplus$ and a function f such that $h = (\oplus/)\circ(f\bullet)$.*

PROOF. The 'if' part follows straightforwardly from the definitions of reduction, map and homomorphism. For the 'only if' part, use induction on the size of the argument sequence, taking for $\oplus$ the operation required by the definition of homomorphism and putting $f = h\circ[\cdot]$, in which the function $[\cdot]$ turns an element into a singleton sequence.

For short, we say then that h is the homomorphism $(\oplus, f)$. The reduction $\oplus/$ is the homomorphism $(\oplus, \mathrm{id})$, in which 'id' stands for the identity function, and the map $f\bullet$ can be written as $(+\!\!+, [\cdot]\circ f)$. Although certainly not all of algorithmics can be reduced or mapped to the construction of homomorphisms, this is a major constructive paradigm.

4. LAWS

We give, without proof, some simple laws about homomorphisms.

LAW 1. $(f\circ g)\bullet = (f\bullet)\circ(g\bullet)$.

LAW 2. *Let f, $\oplus$ and $\oplus'$ satisfy $f(x\oplus y) = (fx)\oplus'(fy)$ and $f(\oplus/[\,]) = \oplus'/[\,]$.*
Then $f\circ(\oplus/) = (\oplus'/)\circ(f\bullet)$.

LAW 3. *Let h be a homomorphism with operation $\oplus$.*
Then $h\circ(+\!\!+/) = (\oplus/)\circ(h\bullet)$.

Laws 2 and 3 are applications of the homomorphism lemma. For the proofs we refer to [3]. Law 3 can also be derived as a special case of law 2. The second condition of law 2 may be left out if the functions in its conclusion are not required to work on empty sequences. From law 3 we also derive

COROLLARY. (a) $(\oplus/)\circ(+\!\!+/) = (\oplus/)\circ((\oplus/)\bullet)$.

(b) $(f\bullet)\circ(+\!\!+/) = (+\!\!+/)\circ((f\bullet)\bullet)$.

Conversely, law 3 follows from the successive application of (b) and (a) of the corollary (using the homomorphism lemma), followed by an application of law 1. From these laws and the corollary one can derive many standard program transformations. For example, some forms of loop fusion can be viewed as an application of law 1, and, as we shall see, filter promotion can also be derived from these laws. The importance of the corollary is that in contrast to laws 2 and 3 it needs no applicability condition.

As example, we give a simple application of law 2. Define the function *last* on non-empty sequences by $last\,(x +\!\!+ [a]) = a$. If we define the operator $\gg$ by $a \gg b = b$, the function *last* can be expressed as a reduction: $last = \gg/$. Let f be a strict function, that is, f 'undefined' = 'undefined'. (Note that $\gg/[\,]$ is undefined.) Since $f(a \gg b) = (fa) \gg (fb)$, law 2 gives us now:

$$f \circ last = f \circ (\gg/) = (\gg/) \circ (f\cdot) = last \circ (f\cdot).$$

The following plays no role in the exercises to follow.
Let $P \lhd x$ be the notation for filtering the sequence x with the predicate P. For example, if *even* is the predicate testing for the property of being even, then $even \lhd [1, 2, 3, 5, 8] = [2, 8]$. It is easy to see that a filter is a homomorphism, with operation $+\!\!+$. So we have, by law 3,

$$(P \lhd) \circ (+\!\!+/) = (+\!\!+/) \circ ((P \lhd)\cdot).$$

This corresponds to the filter-promotion or generate-and-test paradigm: rather than filtering one huge structure, we can divide it into smaller structures, filter each of these, and collect the outcomes.

5. THE SCAN–REDUCE PROBLEM.
First of all, we must give a precise definition of the scan operator '$\backslash$'. This is done with the help of a function α that takes a non-empty sequence x and returns the sequence of non-empty initial subsequences of x, arranged in order of increasing length. We have

$$\alpha[a] = [[a]] \tag{1}$$

$$\alpha(x +\!\!+ y) = (\alpha x) +\!\!+ (x +\!\!+)\cdot \alpha y \tag{2}$$

and now we can define

$$\oplus\backslash x = (\oplus/)\cdot \alpha x. \tag{3}$$

The task before us is to find a homomorphism $(\otimes, f)$ so that

$$\oplus\backslash x = \otimes/f\cdot x. \tag{4}$$

First we determine f:

$$\begin{aligned}
fa &= \otimes/[fa] && \text{(definition of } / \text{ on singletons)}\\
&= \otimes/f\cdot[a] && \text{(definition of } \cdot\text{)}\\
&= \oplus\backslash[a] && \text{(by (4))}\\
&= (\oplus/)\cdot\alpha[a] && \text{(by (3))}\\
&= (\oplus/)\cdot[[a]] && \text{(by (1))}\\
&= [\oplus/[a]] && \text{(definition of } \cdot\text{)}\\
&= [a] && \text{(definition of } / \text{ on singletons).}
\end{aligned}$$

Next we determine $\otimes$ by calculating $x \otimes y$. Suppose $x = \otimes/f\cdot x'$ and $y = \otimes/f\cdot y'$; equivalently, we have $x = \oplus\backslash x'$ and $y = \oplus\backslash y'$. We may assume that x' and y' are non-empty sequences. Then

$$\begin{aligned}
x \otimes y &= (\otimes/f\cdot x') \otimes (\otimes/f\cdot y') && \text{(definition of } x \text{ and } y\text{)}\\
&= \otimes/(f\cdot x') +\!\!+ (f\cdot y') && \text{(definition of } /\text{)}\\
&= \otimes/f\cdot(x' +\!\!+ y') && \text{(definition of } \cdot\text{)}\\
&= \oplus\backslash x' +\!\!+ y' && \text{(by (4))}
\end{aligned}$$

$$
\begin{aligned}
&= (\oplus/)\cdot\alpha(x' +\!\!+ y') && \text{(by (3))}\\
&= (\oplus/)\cdot(\alpha x') +\!\!+ (x' +\!\!+)\cdot\alpha y' && \text{(by (2))}\\
&= ((\oplus/)\cdot\alpha x') +\!\!+ (\oplus/)\cdot(x' +\!\!+)\cdot\alpha y' && \text{(definition of } \cdot).
\end{aligned}
$$

In the last line, $(\oplus/)\cdot\alpha x' = \oplus\backslash x' = x$, and $(\oplus/)\circ(x' +\!\!+) = ((\oplus/x')\oplus)\circ(\oplus/)$, so that

$$
\begin{aligned}
x \otimes y &= x +\!\!+ ((\oplus/x')\oplus)\cdot(\oplus/)\cdot\alpha y'\\
&= x +\!\!+ ((\oplus/x')\oplus)\cdot y,
\end{aligned}
$$

using definition (3) again. The last expression still contains a reference to x', which remains to be eliminated. We note that x' is the last element of the sequence $\alpha x'$. So, using the rule found for *last* in Section 4 from law 2, $\oplus/x' = \oplus/last\,(\alpha x') = last\,((\oplus/)\cdot\alpha x')$. By definition (3), the argument of *last* equals $\oplus\backslash x' = x$, and so $\oplus/x' = last\ x$. Hence

$$x \otimes y = x +\!\!+ ((last\ x)\oplus)\cdot y,$$

and we are done.

6. THE LINES–UNLINES PROBLEM.

Suppose CH is some set of characters, including the newline character NL. Let $CH' = CH - \{\mathsf{NL}\}$. The function *Unlines* has type $\mathrm{Seq}_+\,(\mathrm{Seq}\ CH') \to \mathrm{Seq}\ CH$, and is defined by

$$Unlines = \oplus/ \tag{1}$$

$$x \oplus y = x +\!\!+ [\mathsf{NL}] +\!\!+ y. \tag{2}$$

The reason we insist *Unlines* takes a non-empty sequence as argument is that the operator $\oplus$ does not have a unit, i.e., $\oplus/[\,]$ is not well-defined. It is easy to show that *Unlines* is injective, that is, *Unlines* xs = *Unlines* ys implies $xs = ys$, and so we can specify the function *Lines* to be

$$Lines = Unlines^{-1}. \tag{3}$$

The task before us is to find a homomorphism $(\otimes, f)$ so that *Lines* $x = \otimes/f\cdot x$. We shall discover the definitions of $\otimes$ and f simply by making use of the identity *Lines*(*Unlines* xs) = xs, or, in other words,

$$\otimes/f\cdot\oplus/xs = xs. \tag{4}$$

First we determine f for arguments $a \neq \mathsf{NL}$:

$$
\begin{aligned}
f\,a &= \otimes/[f\,a] && \text{(definition of } /\text{ on singletons)}\\
&= \otimes/f\cdot[a] && \text{(definition of } \cdot)\\
&= \otimes/f\cdot\oplus/[[a]] && \text{(definition of } /\text{ on singletons)}\\
&= [[a]] && \text{(by (4)).}
\end{aligned}
$$

Also

$$
\begin{aligned}
f\,\mathsf{NL} &= \otimes/[f\,\mathsf{NL}] && \text{(definition of } /\text{ on singletons)}\\
&= \otimes/f\cdot[\mathsf{NL}] && \text{(definition of } \cdot)\\
&= \otimes/f\cdot[\,] +\!\!+ [\mathsf{NL}] +\!\!+ [\,] && \text{(as } [\,] \text{ is unit of } +\!\!+)\\
&= \otimes/f\cdot\oplus/[[\,],[\,]] && \text{(definition of } \oplus \text{ and } /)\\
&= [[\,],[\,]] && \text{(by (4)).}
\end{aligned}
$$

Next we must determine $\otimes$. Since each argument of $\otimes$ is a non-empty sequence we need only consider the definition of $(xs +\!\!+ [x]) \otimes ([y] +\!\!+ ys)$. We have

$$
\begin{aligned}
&(xs +\!\!+ [x]) \otimes ([y] +\!\!+ ys)\\
&= (\otimes/f\cdot\oplus/xs +\!\!+ [x]) \otimes (\otimes/f\cdot\oplus/[y] +\!\!+ ys) && \text{(by (4))}\\
&= \otimes/(f\cdot\oplus/xs +\!\!+ [x]) +\!\!+ (f\cdot\oplus/[y] +\!\!+ ys) && \text{(definition of } /)
\end{aligned}
$$

$$
\begin{aligned}
&= \otimes/f\bullet(\oplus/xs +\!\!+ [x]) +\!\!+ (\oplus/[y] +\!\!+ ys) && \text{(definition of } \bullet\text{)}\\
&= \otimes/f\bullet((\oplus/xs)\oplus(\oplus/[x])) +\!\!+ ((\oplus/[y])\oplus(\oplus/ys)) && \text{(definition of } /\text{)}\\
&= \otimes/f\bullet(\oplus/xs) +\!\!+ [\mathsf{NL}] +\!\!+ (\oplus/[x]) +\!\!+ (\oplus/[y]) +\!\!+ [\mathsf{NL}] +\!\!+ (\oplus/ys) && \text{(definition of } \oplus\text{)}\\
&= \otimes/f\bullet(\oplus/xs) +\!\!+ [\mathsf{NL}] +\!\!+ (\oplus/[x +\!\!+ y]) +\!\!+ [\mathsf{NL}] +\!\!+ (\oplus/ys) && \text{(definition of } /\text{)}\\
&= \otimes/f\bullet(\oplus/xs)\oplus(\oplus/[x +\!\!+ y])\oplus(\oplus/ys) && \text{(definition of } \oplus\text{)}\\
&= \otimes/f\bullet\oplus/xs +\!\!+ [x +\!\!+ y] +\!\!+ ys && \text{(definition of } /\text{)}\\
&= xs +\!\!+ [x +\!\!+ y] +\!\!+ ys && \text{(by (4)),}
\end{aligned}
$$

and we are done. Note that the above derivation actually juggles with some potentially fictitious values. We have assigned no meaning to $\oplus/[\,]$, and yet the term $\oplus/xs$ appears above in a context where it is not required that xs be non-empty. No confusion can arise because $\oplus$ does not have a unit in Seq CH, so $\oplus/[\,]$ is adequately defined by the properties it must have. It is consistent with these properties to add the laws

$$(\oplus/[\,]) +\!\!+ [\mathsf{NL}] \;=\; [\mathsf{NL}] +\!\!+ (\oplus/[\,]) \;=\; [\,],$$

which shows how to do actual computations in the extended domain (Seq CH) $\cup\ \{\oplus/[\,]\}$. Note also from the definition of $\otimes$ that its unit must be $[[\,]]$; for example, we can calculate

$$
\begin{aligned}
(xs +\!\!+ [x])\otimes[[\,]] &= (xs +\!\!+ [x])\otimes([[\,]] +\!\!+ [\,])\\
&= xs +\!\!+ [x +\!\!+ [\,]] +\!\!+ [\,]\\
&= xs +\!\!+ [x]\,.
\end{aligned}
$$

This follows also from *Lines* $[\,] = \otimes/[\,]$ and *Unlines* $[[\,]] = [\,]$, since we find $\otimes/[\,] =$ *Lines* (*Unlines* $[[\,]]) = [[\,]]$.

REFERENCES

1. R. S. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming 6* (1986) 159–189.
2. L. Geurts & L. Meertens. Remarks on Abstracto. *ALGOL Bull. 42* (1978), 56–63.
3. L. Meertens. Algorithmics—Towards programming as a mathematical activity. *Proc. CWI Symp. on Mathematics and Computer Science*, CWI Monographs Vol. 1 (J.W. de Bakker, M. Hazewinkel and J. K. Lenstra, eds.) 289–334, North-Holland, 1986.
4. D. Turner. Recursion equations as a programming language. *Functional Programming and its Applications*, Cambridge University Press, Cambridge, UK, 1982.

"I said it in Hebrew--I said it in Dutch--
I said it in German and Greek:
But I wholly forgot (and it vexes me much)
That English is what you speak!"