algorithmic languages

participants edition

edited by
j.w. de bakker and j.c. van vliet

IFIP

MC

north-holland

ALGORITHMIC LANGUAGES

International Symposium on
Algorithmic Languages
Amsterdam, The Netherlands, 26–29 October 1981

organized by
The Mathematical Centre, Amsterdam,
under the auspices of
IFIP Technical Committee 2,
Programming
International Federation for Information Processing

Program Committee

O.-J. Dahl, R.B.K. Dewar, E.W. Dijkstra, A.P. Ershov,
C.A.R. Hoare, G. Kahn, C.H.A. Koster, B. Liskov, M. Paul (Chairman),
J.E.L. Peck, W.L. van der Poel, S.A. Schuman, M. Sintzoff,
T.B. Steel, Jr., W.M. Turski, J.C. van Vliet, N. Wirth, N. Yoneda

NORTH-HOLLAND PUBLISHING COMPANY
AMSTERDAM • NEW YORK • OXFORD

Adriaan van Wijngaarden

Editors' Preface

Adriaan van Wijngaarden, mathematician and computer scientist, was born in Rotterdam, November 2, 1916. He was educated at the Gymnasium Erasmianum in Rotterdam, and studied at the Delft Technological University, where he obtained his Ph.D. in Mechanical Engineering in 1945. The title of his thesis was 'Some applications of Fourier integrals to elastic problems'. His first positions were with the Delft Technological University – during the war years – and, during 1946, with the National Aerospace Laboratory.

In February 1946, the Mathematisch Centrum (MC) was founded in Amsterdam as a research institute in pure and applied mathematics by a number of far-sighted scientists who foresaw the importance of mathematics for the Dutch post-war society. On January 1, 1947, Van Wijngaarden started his work at the MC as founder of its Department of Computation. It was the beginning of his eminent career at our Institute. In the ensuing years, the MC grew from a handful of people to a staff or more than 150 employees. Moreover, computer science in the Netherlands was born, grew up and came of age, all due to the inspiring leadership and great scientific achievements of Van Wijngaarden.

We shall try to briefly outline the main events of Van Wijngaarden's years at the Mathematisch Centrum. Immediately after his appointment he left for an extensive tour – taking most of 1947 – of the UK and the USA. He visited many of the places and people involved in the fascinating development of the first computers and their applications, including Wilkes in Cambridge, Turing and Wilkinson at the National Physical Laboratory, and Goldstine and Von Neumann at the Institute for Advanced Study. Then, upon his return to Holland, Van Wijngaarden initiated the work on the construction of the first Dutch computers. In the early fifties, primarily at the Mathematisch Centrum and, later, also in a number of industrial laboratories, the first electronic computers of the

Netherlands were built. The ARRA was completed at the MC in 1952, and was one of the first machines on the continent. Members of the group headed by Van Wijngaarden were B.J. Loopstra and C.S. Scholten, G.A. Blaauw for a somewhat shorter period, and, at a later stage E.W. Dijkstra and W.L. van der Poel. The latter was employed at that time by the Dutch PTT Laboratory, but worked in close contact with the MC and was actually Van Wijngaarden's first Ph.D. student. (See also the list of Van Wijngaarden's Ph.D. students below.) After the ARRA, the MC constructed the ARMAC and the X1, the first fully transistorized machine. In the late fifties, it was felt that further manufacturing of computers was more appropriate in an industrial environment, rather than in a research institute, and the Electrologica company was founded as an independent firm for this purpose. Later, Electrologica was to become part of the Philips concern.

In the years of his involvement in the development of Dutch computers, Van Wijngaarden also worked very actively as a mathematician, publishing numerous papers on a variety of topics in applied and numerical mathematics, and a few in number theory as well. In fact, the first published algorithm in ALGOL 60 (the procedure *euler* of the Report on the Algorithmic Language ALGOL 60, cf. Peter Naur's invited lecture in these Proceedings) was based on Van Wijngaarden's publication [17] (see the list of publications to follow), one of his main contributions to numerical mathematics.

In the meantime, the importance of Van Wijngaarden's work was recognized by the Dutch scientific community in a number of ways. In 1952, he was appointed 'Bijzonder hoogleraar' at the University of Amsterdam. (This is a part-time appointment with the rank of full professor, financed, e.g., by a research foundation.) In the same year, he became a member of the Board of the Mathematisch Centrum. In 1958 he was appointed as 'Buitengewoon hoogleraar' at the University of Amsterdam (the difference with 'Bijzonder hoogleraar' being that the position is paid by the university) to teach Applied Mathematics. In 1959 he was elected member of the Koninklijke Nederlandse Akademie van Wetenschappen (the Royal Dutch Academy of Sciences), and he also received the 'Medaille d'argent de la ville de Paris'. In 1960 he was elected as a Senior Member of the Institute of Radio Engineers (now IEEE).

In the late fifties – after the termination of the MC's involvement in the construction of computers – Van Wijngaarden's scientific interest

changed direction, and turned to the design of machine independent, general purpose algorithmic languages. It is in this area that the contributions of Van Wijngaarden have probably been the most profound. For this reason, the organizers of the Symposium have selected the theme Algorithmic Languages as an appropriate topic for a conference in his honour. We are very glad that these proceedings contain the excellent papers by Peter Naur and Władysław Turski describing Van Wijngaarden's share in the design of ALGOL 60, and his monumental efforts in the design of ALGOL 68. ALGOL 68 being essentially an IFIP project, it is only to be expected that in Professor Zemanek's impressive address on Van Wijngarden's role in the history of IFIP, a major part is played by the ALGOL 68 developments. The final judgement on Van Wijngaarden's work on algorithmic languages is in the hands of history. The editors cannot but admire its mathematical depth, conceptual richness and elegance, and sheer intellectual power, recognize its lasting influence on the theory and teaching of programming languages, and, at the same time, admit that the complete implementation of ALGOL 68 has posed serious problems, and its practical use has spread little outside the academic world.

In 1961, Van Wijngaarden was appointed director of the Mathematisch Centrum. Besides the demands of his scientific work, he now also carried the responsibility for our Institute – helped by the associate directors F.J.M. Barning and, later, J. Nuis. We feel that it has been a privilege for us to be led by a great scientist. The example he has set us by his outstanding research, his love for mathematics in general – and for the Mathematisch Centrum in particular –, and the way in which he has represented our Institute in national and international bodies concerned with the organization of scientific work have been vital for the MC, and, through this, for the whole Dutch mathematical community. Internationally, most of Van Wijngaarden's organizational contributions have been through IFIP, and we are grateful to Professor Zemanek for his splendid *laudatio* of Van Wijngaarden's IFIP work. In the Netherlands, Van Wijngaarden has been involved in so many organizations that we cannot begin to describe his contributions in full. He was founder and for many years member of the Board of the Nederlands Rekenmachine Genootschap, i.e., the Dutch Computer Society, which appointed him an honoary member in 1972. He was a member of the Board of the Wiskundig Genootschap (the Dutch Mathematical Society), and for many years chairman of its Committee for Scientific Computing. For many years, again, he was chairman of the

Academische Raad Sectie Informatica (the committee coordinating university education in computer science in the Netherlands). He was one of the founders of SARA, the joint computer centre of the Mathematisch Centrum, the University of Amsterdam and the Free University at Amsterdam. And, to close this very incomplete list with an activity which has always been precious to Van Wijngaarden: through the years he has taken a lively interest in computational linguistics, exemplified here by his membership of the committee for Frequency Investigations of the Dutch Language.

For almost thirty years now Van Wijngaarden has been a Professor of Applied Mathematics at the University of Amsterdam. During those years his teaching covered a wide spectrum of topics ranging from, e.g., numerical mathematics through the design and application of ALGOL 60 and ALGOL 68 to the art of two-level grammars. Numerous students have received their first introduction to computer programming through his lectures. The quest for elegance has always been one of Van Wijngaarden's driving forces, and often his audience marveled at the crystal beauty of the algorithms he taught them. Present day teaching of computer science in the Netherlands owes an immense debt to Van Wijngaarden. Virtually all Dutch professors of computer science were either his Ph.D. students (see list below), or spent some years at the Mathematisch Centrum, profiting from its stimulating research conditions. Besides his lectures at the Amsterdam University, Van Wijngaarden has given innumerable lectures in the Netherlands and abroad. Some impression of the scope of his travelling can be obtained by Professor Zemanek's listing of his participation in IFIP meetings. The full list of all his trips extends over ten pages. It includes prolonged stays as visiting professor at New York University, the University of California at Berkeley, and the University of Chicago. Further many invited lectures at important conferences – at the IFIP Congress 68 on ALGOL 68, to mention just one example –, special honours such as the first Fibonacci lecture in Pisa, 1967, and countless talks at universities around the world.

The importance of Van Wijngaarden's work for the Dutch society in general was recognized by his being honoured in 1973 as Ridder in de Orde van de Nederlandse Leeuw (one of the orders in the Queen's list of honours). In 1974, his international work was honoured by the International Federation for Information Processing which awarded him its Silver Core. In 1978, he was awarded an honorary doctorate by the Institut National Polytechnique in Grenoble.

Having started our brief description of Van Wijngaarden's scientific career with mentioning his Ph.D. at the Delft Technological University, we now come to a very appropriate ending: In 1979, Van Wijngaarden was awarded the Doctorate Honoris Causa by the Delft Technological University. W.L. van der Poel, his first Ph.D. student, was now his *promotor*.

On September 1, 1980, Van Wijngaarden retired as director of the Mathematisch Centrum, and became advisor to the Board of Trustees and the Directorate of our Institute. His complete retirement from the MC will take place in the fall of 1981. We know that the last years have been hard for him, due to the untimely death of his beloved wife Willeke. She is remembered in sorrow by countless friends and colleagues of Van Wijngaarden.

Algorithmic – and other – languages continue to be central interests of Van Wijngaarden's scientific life. 'Languageless programming' is the intriguing title of his latest publication, and he remains enticed by the charms of etymology; we are eagerly looking forward to the results of his further studies.

Having reached the end of our Preface, we express our deepest gratitude for everything done for our Institute and for the world of science by Adriaan van Wijngaarden, Dutch mathematician and computer scientist.

The Editors

Publications by A. van Wijngaarden

1942 1 *Laminar flow in radial direction along a plane surface*,
A. van Wijngaarden, Proc. Ned. Akad. Wet., *XLV* (1942) 269–275.

1943 2 *Stroming in radiale richting tussen twee vlakke wanden*,
A. van Wijngaarden, Verslagen Ned. Akad. Wet., *LII* (1943) 29–36.

1945 3 *Enige toepassingen van Fourierintegralen op elastische problemen*,
A. van Wijngaarden, Thesis T.H. Delft, Delft (Meinema, 1945) 122 pp.

1946 4 *Large distortions of circular rings and straight rods*,
A. van Wijngaarden, Proc. Kon. Ned. Akad. Wet, (1946) 648–664.

1947 5 *The elastic stability of flat sandwich plates*,
A. van Wijngaarden, Rep. & Trans. Nat. Aeron. Res. Inst., *XIII* (1947) S37–56.

6 *Over het niet-lineaire verband tusschen de doorbuiging en de belastende kracht van een in zijn beide uiteinden ingeklemden halfcirkelvormigen ring*,
A. van WIjngaarden, De Ingenieur 1947 no 8, Techn. Wet. Ond. 2, 1–3.

1948 7 *Principes der electronische rekenmachines*,
A. van Wijngaarden, Syllabus Math. Centre (1948) 27 pp.

1949 8 *Algemeen overzicht over moderne rekenmachines*,
A. van Wijngaarden, Ned. Tijdschrift voor Natuurkunde, *15* (1949) 243–253.

9 *Ecoulement potential autour d'un corps de révolution*,
A. van Wijngaarden, in Méthodes de calcul dans des problèmes de mécanique, Paris (CNRS, Coll. Int. *XIV*, 1949) 72–87.

10 *Cursus moderne rekenmethoden*,
A. van Wijngaarden, Syllabus Math. Centre, 166 pp.

11 *Table of Fresnel integrals*,
A. van Wijngaarden and W.L. Scheen, Verhandelingen Kon. Ned. Akad. Wet. (Afd. Natuurkunde, 1e Sectie, *XIX*, no 4, 1949) 1–26.

12 *Practisch rekenen*,
A. van Wijngaarden, Eerste Ned. Syst. Inger. Encycl. *IV* (1949) 104–112.

1950 13 *Afrondingsfouten*,
A. van Wijngaarden, Math. Centre, Comp. Dept., MR3 (1950) 20 pp. (Translation: 'Rounding-off errors' by E. Lever and T.W. Hill, Division of Math. Stat. CSIRO, Australia, Techn. Rep. 7 (1972), 16 pp.).

14 *Grundsätzliche Probleme der Abrundungsfehler*,
A. van Wijngaarden, ZAMM, *30* (1950) 275–276.

15 *Table of the cumulative symmetric binomial distribution*,
A. van Wijngaarden, Proc. Kon. Ned. Akad. Wet., *LIII* (1950) 301–312.

16 *A table of partitions into two squares with an application to rational triangles*,
A. van Wijngaarden, Proc. Kon. Ned. Akad. Wet., *LIII* (1950) 313–325.

1951 17 *Large deflections of semi-oval rings*,
A. van Wijngaarden, Rep. & Trans. Nat. Aeron. Res. Inst., *XVI* (1951) S1–7.

18 *Programmeren voor de ARRA*,
A. van Wijngaarden, Math. Centre, Comp. Dept., MR7 (1951), 44 pp.

19 *Decimal-binary conversion and deconversion*,
A. van Wijngaarden, Math. Centre, Comp. Dept., R130 (1951) 41 pp.

1952 20 *Tables for use in rank correlation*,
L. Kaarsemaker and A. van Wijngaarden, Statistica, 7 (1953) 41–54.

21 *Harmonic analysis of earth-tides measurements*,
J. Berghuis and A. van Wijngaarden, Math. Centre, Comp. Dept., R97 (1952) 18 pp.

22 *Table of the integral* $\int_0^1 \exp(-v^{-2} - xv)v^{-p}dv$,
A. van Wijngaarden, Math. Centre, Comp. Dept., R176 (1952) 6 pp.

23 *Rekenmachines*,
A. van Wijngaarden, in Winkler Prins Encyclopaedie, 6th edition, *XV* (1952) 841–842.

24 *Electronische rekenmachines*,
A. van Wijngarrden, (1952) 67–76.

25 *Rekenen en vertalen*,
A. van Wijngaarden, Oratie UVA, Delft (Waltman, 1952) 21 pp.

26 *A note on Bernoulli numbers*,
A. van Wijngaarden, Math. Centre, Comp. Dept., DR7 (1952) 3 pp.

1953 27 *On a certain asymptotic expansion*,
A. van Wijngaarden, Quarterly of Applied Math., *XI* (1953) 244–246.

28 *A tranformation of formal series*,
A. van Wijngaarden, Proc. Kon. Akad. Wet., ser. A, *LVI* (1953) 522–543.

29 *On the coefficients of the modular invariant J (τ)*,
A. van Wijngaarden, Proc. Kon. Ned. Akad. Wet., ser. A, *LVI* (1953) 389–400.

30 *Erreurs d'arrondiment dans les calculs systématiques*,
A. van Wijngaarden, in Les machines à calculer et la pensée humaine, Paris (CNRS, Coll. Int., *XXXVII*, 1953) 285–293.

31 *Ut tensio sic vis*, (in English),

A. van Wijngaarden, in C.B. Biezeno Anniversary volume on applied mechanics, Haarlem (Stam, 1953) 214–224.

32 *A remark on Fermat's last theorem*,
H.J.A. Duparc and A. van Wijngaarden, Nieuw Archief voor Wiskunde (3) *I* (1953) 123–128.

33 *Note on a previous paper on Fermat's last theorem*,
H.J.A. Duparc and A. van Wijngaarden, Nieuw Archief voor Wiskunde (3) *II* (1954) 40–41.

34 *Het gebruik van automatische rekenmachines*,
A. van Wijngaarden, in Verslag van het tiende Nederlands Congres van leraren in de wiskunde en de natuurwetenschappen, Groningen (Wolters, 1954) 13–18.

35 *Mathematics and computing*,
A. van Wijngaarden, in Automatical digital computation, NPL, Her Majesty's Stationary Office, (1954) 125–127 (also p. 124).

36 *Table of Everett's interpolation coefficients*,
E.W. Dijkstra and A. van Wijngaarden, Math. Centre, Comp. Dept., R294 (1955) 204 pp.

37 *Introduction*,
A. van Wijngaarden, Proc. Inst. Electrical Engineers, *103* part B Supplement number 1 (1956) 112–113.

38 *Capita uit de numerieke wiskunde*,
A. van Wijngaarden (ed), J. Berghuis, E.W. Dijkstra, Math. Centre, Syllabus Coll. 1955/56, 25 pp.

39 *Programmeren voor automatische rekenmachines*,
A. van Wijngaarden, E.W. Dijkstra (eds.), Math. Centre, Syllabus cursus 1955/56 (1956), 128 pp.

1957 40 *Automatisering in de Wetenschap*,
A. van Wijngaarden, in Economisch-Statistische Berichten, no. 2103 (1957) 832–833.

41 *Programmeren voor automatische rekenmachines*,
A. van Wijngaarden (ed), T.J. Dekker, E.W. Dijkstra, Math. Centre, Syllabus cursus 1956/57 (1957) 106 pp.

1959 42 *The state of computer circuits containing memory elements*,
A. van Wijngaarden, in Proc. Int. Symp. on the theory of switching (1957), Annals Comp. Lab. Harvard Univ. *30* (1959) 213–224.

1960 43 *Report on the algorithmic language ALGOL 60*,
J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur (editor), A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, M. Woodger, Num. Mathematik *2* (1960) 106–136, and elsewhere.

1961 44 *L'influence de ALGOL sur l'analyse numérique*,

A. van Wijngaarden, in Colloque sur l'analyse numérique tenu à Mons les 22, 23 et 24 Mars 1961, CBRM, Paris (Gauthier-Villars, 1961) 89–97.

1962 45 *Generalized ALGOL*,
A. van Wijngaarden, in Symbolic languages in data processing, New York (Gordon and Breach, 1962) 409–419 and in Annual Review in Automatic programming, *3* (1963) 17–26.

1963 46 *Revised Report on the algorithmic language ALGOL 60*,
J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur (editor), A.J. Perlis, H. Rutishauer, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, M. Woodger, Num. Mathematik *4* (1963) 420–453, and elsewhere.

47 *Switching and programming*,
A. van Wijngaarden, in Switching theory in space technology, Stanford (Un. Press, 1963) 275–283.

1964 48 *Recursive definition of syntax and semantics*,
A. van Wijngaarden, in Formal language description languages for computer programming, Amsterdam (North Holland, 1966) 13–24.

49 *Rekenen in Nederland*,
A. van Wijngaarden, in NRMG 1959–1964, Amsterdam (NRMG, 1964) 5–23.

50 *The language of the future*,
A. van Wijngaarden, in Proc. Nat. Conf. on data processing, Rehovoth (1964), Inf. Proc. Ass. of Israel (1965) 189–211.

51 *Formal properties of newspaper Dutch*,
J.A.Th.M. van Berkel, H. Brandt Corstius, R.J. Mokken and A. van Wijngaarden, Math. Centre Tracts 12, 123 pp.

1965 52 *Orthogonal design and description of a formal language*,
A. van Wijngaarden, Math. Centre, Comp. Dept., MR76 (1965) 25 pp.

53 *Procesanalyse*,
A. van Wijngaarden, Syllabus Math. Centre (1965) 129 pp.

1966 54 *Numerical analysis as an independent science*,
A. van Wijngaarden, BIT, Nordisk Tidskrift for Informationsbehandling, *6* (1966) 66–81.

55 *Triangular arrays of digits*,
A. van Wijngaarden, Math. Centre, Comp. Dept., MR83 (1966) 28 pp.

56 *De kwadratuur van de cirkel*,
A. van Wijngaarden, Mededelingen NRMG, 8, 1 (1966) 1–3 and cover.

57 *A draft proposal for the algorithmic language ALGOL X*,
A. van Wijngaarden, B.J. Mailloux, Math. Centre, Comp. Dept. (1966) 63 pp.

1967 58 *A draft proposal for the algorithmic language ALGOL 67*,

A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, Math. Centre, Comp. Dept., MR88 (1967) 129 pp.

59 *A draft proposal for the algorithmic language ALGOL 68*,
A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, Math. Centre, Comp. Dept., MR92 (1967) 124 pp.

1968 60 *Draft report on the algorithmic language ALGOL 68*,
A. van Wijngaarden (ed), B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, Math. Centre, Comp. Dept., MR93 (1968) 111 pp.

61 *Working document on the algorithmic language ALGOl 68*,
A. van Wijngaarden (ed), B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, Math. Centre Comp. Dept., MR95 (1968) 180 pp.

62 *Penultimate draft report on the algorithmic language ALGOL 68*,
A. van Wijngaarden (ed), B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, Math. Centre, Comp. Dept., MR99 (1968) 183 pp.

63 *Final draft report on the algorithmic language ALGOL 68*,
A. van Wijngaarden (ed), B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, Math. Centre, Comp. Dept., MR100 (1968) 155 pp.

1969 64 *Report on the algorithmic language ALGOL 68*,
A. van Wijngaarden (ed), B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, Math. Centre, Comp. Dept., MR101 (1969, 3 printings) and elsewhere, especially Numerische Mathematik *14* (1969) 79–218. Translated into French, German, Russian, Bulgarian and Chinese.

65 *Physionomie, psyche en chironomie*,
Leo Geurts, Lambert Meertens, Reind van de Riet, Aad van Wijngaarden, Math. Centre, 4 october 1969, 34 pp.

66 *Rekenmethodes voor automatische rekenmachines*,
A. van Wijngaarden, in Colloquium Moderne Rekenmachines 1, MC Syllabus 7.1 (1969) 73–80.

67 *Dynamica van rekenmachines*,
A. van Wijngaarden, in Colloquium Moderne Rekenmachines 1, MC Syllabus 7.1 (1969) 120–128.

1970 68 *On the boundary between natural and artificial languages*,
A. van Wijngaarden, in Linguaggi nella società e nella tecnica, Milano (edizioni di Communità, 1970) 165–176.

1971 69 *MC: MCMXLVI - MCMLXXI*,
A. van Wijngaarden, in ZWO Jaarboek 1971, 105–109.

1973 70 *Ontwikkelingen op computergebied*,
A. van Wijngaarden, in Een kwart eeuw wiskunde 1946–1971, MC Syllabus *18* (1973) 59–79.

1974 71 *On the generative power of two-level grammars*,
A. van Wijngaarden, in Proc. 2nd Colloquium on Automata, Languages and Programming, Saarbrücken, Springer Lecture Notes in Comp. Sci. *14* (1974) 9–14.

1975 72 *Revised Report on the algorithmic language ALGOL 68*,
A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker, editors, Acta Informatica *5* (1975) 1–234, and elsewhere. Translated into French, German, Russian.

73 *Programmacorrectheid en grammatica's*,
A. van Wijngaarden in Colloquium Programmacorrectheid 1975, MC Syllabus *21* (1975) 177–186.

1977 74 *Het hangbuikzwijnwijfje*,
A. van Wijngaarden, in Lexicologie, opstellen voor F. de Tollenaere, Groningen (Wolters-Noordhoff, 1977) 309–313.

1979 75 *Thinking on two levels*,
A. van Wijngaarden, in Proc. bicentennial congress Wiskundig Genootschap, part II, Math. Centre Tracts *101* (1979) 417–428.

1981 76 *Languageless programming*,
A. van Wijngaarden, in Proc. IFIP/TC2/WG2.5 working conference on the relations between numerical computation and programming languages, Boulder, North-Holland, to appear.

Ph.D. students of A. van Wijngaarden

1. W.L. van der Poel,	The logical principles of some simple computers, 1956:02:01.
2. N.C. de Troye,	Classification and minimization of switching functions, 1958:07:09.
3. E.W. Dijkstra	Communication with an automatic computer, 1959:10:28.
4. G. Zoutendijk,	Methods of feasible directions, 1960:06:22.
5. J.A. Zonneveld,	Automatic numerical integration, 1964:06:17.
6. J.W. de Bakker,	Formal definition of programming languages, 1967:05:17.
7. R.P. van de Riet,	ALGOL 60 as formula manipulation language, 1968:02:07.
8. B.J. Mailloux,	On the implementation of ALGOL 68, 1968:06:12.
9. J. Verhoeff,	Error detecting decimal codes, 1969:06:25.
10. H. Brandt Corstius,	Exercises in computational linguistics, 1970:01:21.
11. M.H. van Emden,	An analysis of complexity, 1971:06:09.
12. P. van Emde Boas,	Abstract resource-bound classes, 1974:09:18.

13. H.J.J. te Riele,	A theoretical and computational study of generalized aliquot sequences, 1976:01:21.
14. J.C. van Vliet,	ALGOL 68 transput, 1979:10:03.
15. D. Grune,	On the design of ALEPH, 1981: ...

Foreword

The International Symposium on Algorithmic Languages is an event which, while dealing timely with a wide selection of appropriate topics in the field, brings also back memories from earlier years when this subject caught first the interest of scientists. In fact the advent of computers posed almost immediately the problem how one could best describe the algorithms that one wanted to be performed with the help of these machines. K. Zuse developed already during the year 1945 for this purpose his 'Plankalkül' which allowed him to formulate algorithms, albeit in a form which was mostly machine oriented.

An important step forward was made when, apart from the mere descriptive details, research about the fundamental concepts of programming began. Strong impulses in this direction were given in 1951 by H. Rutishauer in his paper on automatic design of calculating plans for programmable computers. Influenced by this work one of the fundamental discoveries was made by F.L. Bauer and K. Samelson when they detected the cellar principle. Through it the parsing of bracket structures became very transparent, easily understandable, and efficiently implementable. The latter being the most apparent and immediate aim since it allowed an important part of compiler writing to be handled in a very satisfactory way, the then new principle really goes much deeper than this. It marks in truth the discovery of a basic equivalence: the equivalence of the abstract data structure cellar (resp. push-down-store or LIFO-list) with trees in their depth-first interpretation and consequently, therefore, among many others with block- and bracket-structures in programming languages. This principle has led the authors of the Report on the Algorithmic Language ALGOL 60 to introduce rigorously the block structure for controlling the scope of variables; moreover it was also used dynamically insofar as the hierarchies of incarnations of procedure-bodies invoked by procedure calls followed the same principle: the tree structure of hierarchies formed by

procedure calls allows the storage allocation to be handled by a cellar. Without the equivalence mentioned above stated explicitly at that time, for some working in this field the run-time stack was then seen as something completely different from the operator-cellar for parsers of block- and bracket-structures. Samelson's suggestions for the design of ALGOL, however, were guided by his fully understanding the equivalence principle.

Meanwhile new challenges for the dealing with algorithmic languages developed. Especially recursive data structures had become more important the more computer applications went into non-numerical computations on a broader scale. List structures by J. McCarthy and the record structures – abstract and concrete – by C.A.R. Hoare were the appropriate answer given to this challenge in the design of programming languages. However, the greatest challenge for algorithmic languages in the 1960ies came, when computing scientists felt utterly distressed by the fact that programming and writing programs in an algorithmic language was becoming so complex that it was extremely error-prone and had developed into a hardly manageable engineering discipline.

These sorrows were openly discussed at the famous Garmisch Conference on Software Engineering. One decisive step towards making programming more reliable was the introduction of the axiomatic method by R.W. Floyd and C.A.R. Hoare. With it programs written in an algorithmic language could be proved to be correct; but this approach was really reaching much further than that in as far as it laid the foundations for research in the field of constructing correct programs. It had a strongly stimulating effect for work in this direction. The latter came about with the predicate transformers introduced by E.W. Dijkstra, an idea that must be considered as fundamental in the design and evolution of correct programs starting from correct specifications of given problems. Considering this important area makes clear that studying algorithmic languages involves the investigation not only of data structures and function applications but also of the process of designing algorithms. In fact this latter aspect has recently drawn at least as much interest as the former, and it will become even more demanding with application programs being required for further and larger problem areas.

Another great challenge in dealing with algorithmic languages is presented by distributed processing which has arrived in the wake of the technological progress achieved with microprocessors and microprogramming. There is a host of research work going on at present in this field with

regard to establishing principles and finding basic concepts for interacting processes and their mutual communications. Much has been achieved already if we consider concepts like semaphores, monitors, critical regions, tasks, or if we look at models like Petrinets and data-flow machines. However, much more remains to be done if we want to understand and keep under control the enormous complexity of parallel programming which is needed with distributed systems.

The development of algorithmic languages and typical research areas connected with them as indicated above is reflected to a considerable extent in the definition of high-level programming languages beginning with the publication of FORTRAN in 1956 and ALGOL in 1958. It can be said in regard of this line of languages and research areas that many members of working groups under IFIP/TC 2 have had a great impact and influence upon this evolution, thereby making a number of important steps toward the point where we stand today. ALGOL 60 was the first of these steps, and that language has played an important part as a stimulus for many research projects in programming and compiling just as much as it was and is used as a programming language.

Starting from ALGOL 60, within WG 2.1, ALGOL 68 was developed, and new ideas (e.g. records, concise parameter linkage) went into that language. It is known that ALGOL 68, unlike ALGOL 60, is controversial in many details, and has by far not found the widespread use of ALGOL 60. It has, however, again served as a focus for interesting and stimulating discussions in the field of programming languages. Furthermore, in the Report on the Algorithmic Language ALGOL 68 a powerful and adaptable form of grammar, which A. van Wijngaarden had introduced, was used for the definition of the language. It were this class of two-level-grammars and the class of VDL-languages which served as a starting point for discussions in WG 2.2 which had been founded as a result of the succesful Working Conference on Formal Language Definition Languages, 1964 in Baden near Vienna. The controversies over ALGOL 68 had on one hand the deplorable effect of splitting WG 2.1 but, as it turned out, also the very positive effect of creating WG 2.3 on Programming Methodology on the other.

All three working groups mentioned and, since 1973, WG 2.4 on Systems Implementation Languages have had a sizeable share in creating a better understanding of algorithmic languages, their concepts, their definition, and their use in writing programs. TC 2 has therefore good reasons to

appreciate that, under its auspices, an International Symposium on Algorithmic Languages is organized by the Mathematical Centre as a tribute to A. van Wijngaarden. The submitted papers which have been selected deal with most of the areas which were mentioned above. In addition, the program committee has invited speakers who will give talks about selected topics on algorithmic languages as well as about A. van Wijngaarden's contributions to ALGOL and his work for IFIP.

With great thankfulness to the authors and to the organizers from the Mathematical Centre I express, also on behalf of IFIP/TC 2, my belief that this Symposium deserves to be well received.

München, September 1981

M. Paul,
Chairman Program Committee

Table of Contents

*This paper has been omitted from the Participants' Edition at the explicit request of the author.

Algorithmic Languages, de Bakker/van Vliet (eds.)

Invited Opening Address

The Role of Professor A. van Wijngaarden in the History of IFIP

Heinz Zemanek

University of Technology, Vienna, Austria

Speaking of the role of Professor van Wijngaarden means speaking of the European history of computing and of programming languages from EDSAC to the present day. It also means speaking about the history of IFIP. It is impossible to separate these subjects.

It is, however, equally impossible for me to treat this compound as a whole or tell the entire Van Wijngaarden story. I would never dare to embark on such a giant enterprise. What I can do and what I have been asked to do is to give a description of what I have seen and experienced in 25 years of my acquaintance with him and leave out the formal, the seriously scientific part, which is much better reflected by the symphony of papers that is to follow in this week. My personal view will resemble a shadow showing the contours, but never acquiring the full splendour of a portrait painted in colours.

Before 1959

I am not entirely sure about when our relationship began, but I believe that I first met Professor van Wijngaarden in Darmstadt at the first European computer conference with some international flavour which I had an opportunity to assist. From the very beginning I have sensed the dual character of his unique personality: the large mind which has always extended beyond my horizon, and the sharp brain that can suddenly focus on the smallest detail, but will illustrate by it some general aspect; the

'generalizer' who generalized even a general purpose programming language, and the 'specializer' whose production of sentences and questions has often reminded me of a pencil sharpener.

At the Darmstadt GAMM-NTG-Fachtagung in October 1955 on *Electronic Digital Computers and Information Processing,* organized by Professor Alwin Walther, Professor van Wijngaarden gave a survey on *Scientific computing in The Netherlands* [1]. It started with the observation made by someone during the conference that the per capita number of computers in The Netherlands was astonishingly high, maybe the highest – at that time – in Europe. Professor van Wijngaarden left some doubts whether this was really true, but he stressed the vivid activity in computer research in his country.

Apart from a Ferranti computer in the Shell Laboratories, there were at that time four computers that had been developed by and realized for research in The Netherlands as well as several others still in planning stage, and in all these cases – he himself did not say that clearly – he and his students played a leading role: there was PTERA in PTT, which had been developed by Kosten and Van der Poel and was running already for some years, and there was ARRA, an electronic replacement of the earlier relay computer of the same name, at the Mathematisch Centrum. This institution had cooperated with Fokker to copy this machine for them – it was then called FERTA – and a second, faster machine, ARMAC. The paper included many slides of all those computers.

Two years later we met again in Cambridge, MA, at Howard Aiken's conference of 1957 where Professor van Wijngaarden's paper was on *The state of computer circuits containing memory elements* [2], giving his version of sequential switching algebra and elementary automata theory.

Another two years later we were together at the ALGOL conference in Copenhagen in February 1959, which was devoted to the exchange of ideas and experiences with this new language. The prehistory is the following. After the Darmstadt Fachtagung GAMM established a committee for programming, and when in April 1958 they compared their work with the results of a similar committee of ACM, they found that there was a lot in common. It was therefore easy for both sides to accept cooperation. A joint ACM-GAMM Committee was appointed and met in Zurich in May 1958. They formulated a preliminary report on an *International Algorithmic Language* [10], first abbreviated by IAL and later called ALGOL (58). The members of the Joint Committee were, for ACM,

D. Arden, J. Backus, P. Desilets, D.C. Evans, R. Goodman, S. Gorn, H. Huskey, C. Katz, J. McCarthy, A. Orden, A.J. Perlis, R. Rich, S. Rosen, W. Turanski and J.H. Wegstein, and for GAMM, F.L. Bauer, H. Bottenbruch, P. Graeff, P. Läuchli, M. Paul, F. Penzlin, H. Rutishauser and K. Samelson.

As an ACM-GAMM-creation, ALGOL was an achievement of two sub-societies of the later IFIP member organizations AFIPS and DARA, and since the 13 ALGOL fathers decided to bring ALGOL under the umbrella of IFIP, ALGOL is a keyword of this paper, in particular because Professor van Wijngaarden is the father of ALGOL 68. I will come back later to this stream of events.

1959: ICIP

The meetings I have so far mentioned can be seen today as events leading to the big bang in international information processing: to ICIP, the International Conference on Information Processing organized under the auspices and at the headquarters of UNESCO in Paris, in August 1959. Professor van Wijngaarden was a leading figure in this extremely important gathering, not only because he had the honouring title Vicepresident of the Congress, but mainly because of his contributions to the congress organization and programme. It is impossible to evaluate or estimate the number of acquaintances, friendships, events and developments which this first large-scale international computer conference initiated. It is fascinating to read today, 22 years later, the proceedings of that conference, including the paper by Backus on the definition of ALGOL syntax by production rules, a paper by Bauer and Samelson on ALGOL (58) and a paper with the famous title *Processing data in bits and pieces* by Brooks, Blaauw and Buchholz. It is equally impressive to read the list of participants; hardly any name famous in our field is missing.

1960: IFIP

The main consequence of the UNESCO Congress was the foundation of

IFIP, the International Federation of Information Processing, which had been prepared in parallel and completed in 1960 by essentially the same group of people, with I.L. Auerbach of the U.S.A. and J.A. Mussard of UNESCO as the main driving forces. IFIP should not only continue to organize international computer congresses, it should become the basis of international cooperation in all fields of information processing and the clearinghouse of ideas and activities. In 1959 nobody in Paris would have dared to predict that within 20 years IFIP would have 40 member nations, 10 Technical Committees, 30 Working Groups and half a thousand members making up all those committees. This is certainly no reason to congratulate ourselves, and critical judgement does not only come from the outside – IFIP is well aware of its shortcomings and is continuously reviewing its structure and its activities, its policies and motivations.

Sometimes critical remarks and reorganization proposals have been unrealistic or naive. IFIP is largely *bound* by the nature and quality of its member organizations and by the delegates commissioned by them; IFIP can hardly be better than the sum or the average of its constituents. IFIP has lost less time and effort by fruitless political discussions than any other similar organization I know. It would be a good thing to cut down on its administration and to have fewer non-scientific and more scientific and technical meetings. But it is easier to propose such a reduction than to realize it without any damage to positive work. The people who installed IFIP, and Professor van Wijngaarden is one of them, knew very well to balance administrative needs and technical work and to build up a high level and a climate of mutual confidence which are not easy to improve. In a universe of increasing diversification of information processing, of reduced resources in funds and manpower, of less support for events and travelling, it is not easy to maintain the standard of the past, when increasing duties and more problems call for increasing scopes and achievements. IFIP needs the contributions and the sympathy of everyone in the field. Professor van Wijngaarden is an admirable example for all of us; in a seafaring country like Holland you might be reminded of a ship's figurehead, a smiling, mythical beauty who is constantly ahead of the crew and the passengers buried in the entrails of the ship.

Professor van Wijngaarden was not simply the representative of The Netherlands in the IFIP Council and later in the General Assembly. In the early years of IFIP he assumed almost all possible positions and participated in nearly all events, not with the intention to obtain fame and

Table 1
Professor van Wijngaarden in IFIP

ICIP 59: Congress Vicepresident

IFIP COUNCIL/GENERAL ASSEMBLY: Member 1960–1971
IFIP Vice-President: 1962–1964
IFIP Trustee (elected COUNCIL member): 1967–1970
CHAIRMAN TC 1: 1967–1974 → hibernated
Member WG 1.1: 1967–1974 → hibernated
Member TC 2: 1962–1971 → Koffeman
Member WG 2.1: since 1962
Member WG 2.2: since 1965
CHAIRMAN Future Policy Committee: 1963–1967
CHAIRMAN Publications Committee: 1965–1969
Finance Committee, Member: 1961–1962
Statutes and Bylaws Committee: 1969–1971
Member Congress Programme Committee: 1962
Member Working Conference Organizing Committee 1963/64
Chairman and Organizer IFIP 10 Years Anniversary 1969/70
SILVERCORE Recipient 1974

Table 2
Professor van Wijngaarden at IFIP events (and before)
Explanations:
[1] Paper read and published; see literature.
[76] Report of the Mathematisch Centrum, distributed before or at WG 2.1 meetings; see literature.
[::] Paper read but not published.
[Ω] Excused at that meeting – only 4 meetings!

OCT	1955	DARMSTADT	GAMM-NTG-Tagung	[1]
APR	1957	CAMBRIDGE MA	Aiken Conference	[2]
FEB	1959	COPENHAGEN	ALGOL Conference	
AUG	1959	PARIS	ICIP 59	
NOV	1959	PARIS	ALGOL Conference	
JAN	1960	PARIS	ALGOL 60 Conference	
JUN	1960	ROME	1st IFIP COUNCIL	[Ω]
FEB	1961	DARMSTADT	2nd IFIP COUNCIL	
OCT	1961	COPENHAGEN	3rd IFIP COUNCIL	
FEB	1962	SUNNYVALE CA	Aiken Conference	[3]
MAR	1962	MUNICH-FELDAFING	1st TC 2 Meeting	
MAR	1962	MUNICH-FELDAFING	4th IFIP COUNCIL	
MAR	1962	ROME	ICC Conference	[4]
AUG	1962	MUNICH	1st WG 2.1 Meeting	
AUG	1962	MUNICH	2nd TC 2 Meeting	

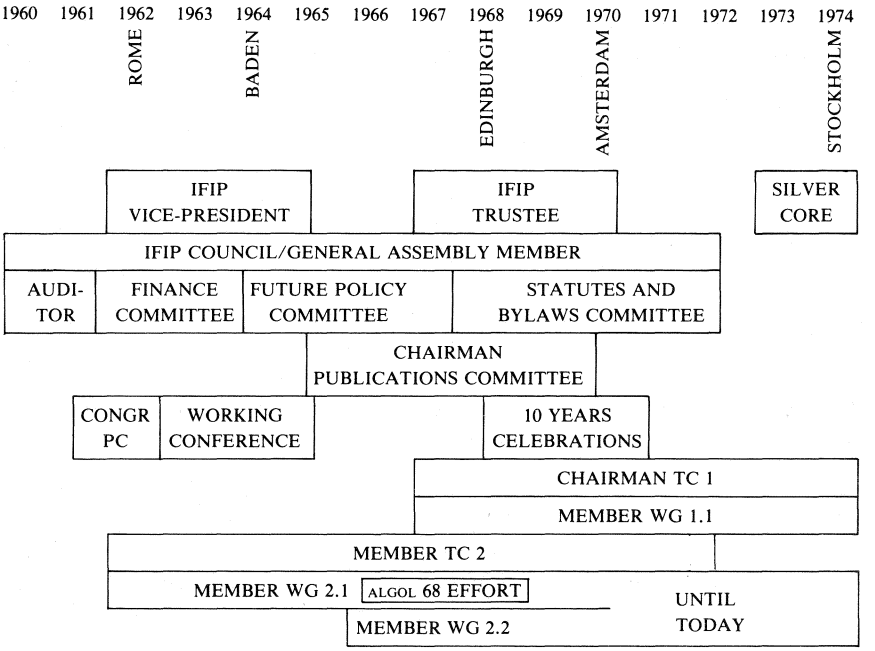
AUG	1962	MUNICH	5th IFIP COUNCIL	
AUG	1962	MUNICH	2nd IFIP CONGRESS	
SEP	1963	DELFT	2nd IFIP WG 2.1	
SEP	1963	OSLO	3rd TC 2 MEETING	
SEP	1963	OSLO-GOLA	6th IFIP COUNCIL	
MAR	1964	MUNICH-TUTZING	3rd WG 2.1 Meeting	
MAY	1964	PRAGUE-LIBLICE	4th TC 2 Meeting	
MAY	1964	PRAGUE-LIBLICE	7th IFIP COUNCIL	
SEP	1964	VIENNA-BADEN	4th WG 2.1 Meeting	
SEP	1964	VIENNA-BADEN	1st IFIP WORKING CONFERENCE	[5]
NOV	1964	ROME	8th IFIP COUNCIL	
MAY	1965	PRINCETON NJ	5th WG 2.1 Meeting	
MAY	1965	NEW YORK CITY	5th TC 2 Meeting	
MAY	1965	NEW YORK CITY	9th IFIP COUNCIL	
MAY	1965	NEW YORK CITY	3rd IFIP CONGRESS	
OCT	1965	St. PIERRE	6th WG 2.1 Meeting	[76]
NOV	1965	NICE	10th IFIP COUNCIL/GENERAL ASSEMBLY	
APR	1966	KOOTWIJK	Subcommittee Meeting	
APR	1966	LONDON	7th TC 2 Meeting	[Ω]
APR	1966	LONDON	10.5 IFIP COUNCIL	[Ω]
JUN	1966	PISA	2nd IFIP WORKING CONFERENCE	
OCT	1966	WARSAW	7th WG 2.1 Meeting	
NOV	1966	JERUSALEM	11th IFIP GENERAL ASSEMBLY	
APR	1967	MADRID	11.5 IFIP COUNCIL	
APR	1967	ZANDVOORT	8th WG 2.1 Meeting	[88]
MAY	1967	OSLO	8th TC 2 Meeting	
MAY	1967	OSLO	3rd TC 2 WORKING CONFERENCE	
SEP	1967	ALGHERO SARDINIA	1st WG 2.2 Meeting	
OCT	1967	MEXICO CITY	12th IFIP GENERAL ASSEMBLY	[Ω]
APR	1968	TBILISI USSR	12.5 IFIP COUNCIL	
JUN	1968	ZURICH	ALGOL 10 YEARS Anniversary	
JUN	1968	PISA-TIRRENIA	9th WG 2.1 Meeting	[93]
JUL	1968	COPENHAGEN-VEDBAEK	2nd WG 2.2 Meeting	
JUL	1968	NORTH BERWICK	10th WG 2.1 Meeting	[95]
AUG	1968	EDINBURGH	2nd WG 1.1 Meeting	
AUG	1968	EDINBURGH	1st TC 1 Meeting	
AUG	1968	EDINBURGH	9th TC 2 Meeting	
AUG	1968	EDINBURGH	13th IFIP GENERAL ASSEMBLY	
AUG	1968	EDINBURGH	4th IFIP CONGRESS	[::]
DEC	1968	MUNICH	11th WG 2.1 Meeting	[100]
JAN	1969	LONDON-GUILDFORD	10th TC 2 Meeting	
MAR	1969	BRUSSELS	13.5 IFIP COUNCIL	[Ω]
APR	1969	HILVERSUM	3rd WG 1.1 Meeting	
APR	1969	HILVERSUM	2nd TC 1 Meeting	
SEP	1969	CALGARY-BANFF	13th WG 2.1 Meeting	
OCT	1969	PRAGUE	11th TC 2 Meeting	
OCT	1969	PRAGUE	14th IFIP GENERAL ASSEMBLY	
JAN	1970	LONDON	4th WG 1.1 Meeting	
MAY	1970	ATLANTIC CITY NJ	14.5 IFIP COUNCIL	
JUN	1970	MUNICH	4th TC 2 WORKING CONFERENCE ALGOL 68	

JUL	1970	HABAY-LA-NEUVE	13th WG 2.1 Meeting	
AUG	1970	EINDHOVEN	12th TC 2 Meeting	
SEP	1970	NEW HAVEN	5th WG 2.2 Meeting	
OCT	1970	AMSTERDAM	15th IFIP GENERAL ASSEMBLY	
OCT	1970	AMSTERDAM	IFIP 10 YEARS Celebrations	[::]

Table 3
25 Years of Professor van Wijngaarden: 1955–1980

1955	DARMSTADT	GAMM-NTG-Fachtagung	[1]
1956			
1957	CAMBRIDGE MA	Aiken Conference	[2]
1958			
1959	PARIS	ICIP 59	
1960	ROME	1st IFIP COUNCIL	
1961		1st TC 2 Meeting	
1962		1st WG 2.1 Meeting, IFIP Vice-President	
	ROME	Paper on Generalized ALGOL	[4]
1963	GOLA	Chairman of Future Policy Committee	
1964	VIENNA-BADEN	1st IFIP Working Conference	[5]
1965	PRINCETON	ALGOL X begins	
	NEW YORK CITY	Chairman of Publications Committee till 1968: hard development work	
1966			
1967		Chairman TC 1	
1968	EDINBURGH	ALGOL 68 lecture at 4th IFIP Congress	
1969		Chairman of Statutes and Bylaws Committee	
1970	AMSTERDAM	10 YEARS ANNIVERSARY CELEBRATIONS	
1971		Resignation from General Assembly and TC 2	
1972			
1973		Resignation from TC 1 and WG 1.1	
1974	STOCKHOLM	SILVERCORE at 6th IFIP Congress	
1975		Revised ALGOL 68 Report	
1976			
1977			
1978			
1979	URGENCH	Lecture at Symposium on Algorithms	
1980			
1981	AMSTERDAM	Honored by Symposium	

Table 4
Professor van Wijngaarden and IFIP



honours, but working hard to make his contributions worthwhile. Tables 1 and 2 show the quasi syntactical size of his efforts in the form of a list of positions and a list of events in which he participated. The semantical size of his contributions is not so easy to show, but I will try. When I wrote this paper, I realized very soon that I should have started a year ago on a full research project including interviews with people all over the world; thus I might have done a really good job. But I doubt that Professor van Wijngaarden would like such an enterprise and I hope that he prefers my imperfect achievements and will forgive me for everything I do not know or forget to mention.

Professor van Wijngaarden was IFIP Vicepresident from 1962 to 1964, IFIP Trustee (i.e. an elected Council member) from 1967 to 1970, and he served on many IFIP committees. His first job was that of an auditor for the first IFIP accounts, and his second was in the IFIP Finance Committee. He chaired the first IFIP Future Policy Committee, then called Committee

for Future Operations and Policies, and there he laid the foundation for all future planning activities.

In those early days the IFIP family was much smaller and each national representative was a kind of general-purpose officer. The programme for IFIP Congress 62 was made up much along the same lines as it is being done today, but the Programme Committee consisted mainly of Council members. Since I had also been included – although Austria was not yet an IFIP member – Van Wijngaarden and I met in Copenhagen in October 1961, where the final programme was established, and we met of course at the Munich IFIP Congress 62. This was the first real IFIP congress, but still got the number '2' (the ICIP congress was considered number 1). This made it possible to go in parallel with our sister organizations – IFAC, IFORS, IMEKO and (then) AICA, which were later coordinated by FIACC, the Five International Organizations Coordinating Committee – which all accepted the 3-year cycle and have the same counting within one cycle as IFIP. Naturally we met again at the congresses in New York City in 1965 and in Edinburgh 1968 – the General Assembly is always held in the week before the congress and there are often committee meetings arranged at the same time in order to save on travel expenses.

1962: Rome and TC 2

This is the point to turn back to the stream of ALGOL events, since 1962 was a key year for both ALGOL and Professor van Wijngaarden. That year we first met in Sunnyvale, CA, where Howard Aiken had organized a conference on *Switching Theory in Space Technology* – but actually it had not too much to do with space travelling, Aiken had simply found a way to gather computer people in California with the remarkable support of the local industry. Professor van Wijngaarden read a paper on *Switching and programming* [3] which began as follows:

In switching theory much attention has been paid to the analysis and simplification of circuits and systems, and to properties of networks. The objective has been to provide network structures using rather simple components.

In the programs for automatic computers, similar structures are found, although on another scale. These programs consist of sequences of statements performing certain operations and are connected by transfers

for control into a complicated network. Executing the statement means moving along the paths of the circuits, seemingly completely different structures may be more or less the same functionally, and the problem of simplification arises immediately.

This was not simply an argumentation to make a paper on programming fit into a conference on switching, this was the indication of a path and the discovery of an equivalence the use and advantages of which have not yet been fully recognized today. We are all too preoccupied with daily work to dig deeper into such proposals and so were we in those days.

Already one month later we met again in Feldafing near Munich in order to start IFIP TC 2.

ALGOL, as I have already mentioned, was originally an ACM-GAMM creation, but after the publication of the Preliminary Report, the interest went up very steeply. Professor van Wijngaarden joined the enterprise in 1959, after an, in ALGOL 68 terminology, lengthened to **long** stay in Scotland. After the Copenhagen meeting in February there was another one in Paris in December, and after the ICIP Congress in Paris the last preparations were made for the Paris Conference in January 1960, where the *Report on the Algorithmic Language ALGOL* 60 [11] worked out by a committee originally planned to consist of seven ACM and seven GAMM members, but since William Turanski was killed in a car accident shortly before the conference, the number of 13 ALGOL fathers emerged: J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur (editor), A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden and M. Woodger. Thus Professor van Wijngaarden is one of the 13 ALGOL fathers and Peter Naur will describe his contributions to ALGOL 60 in the course of this symposium.

The best way to follow the development is to study the ALGOL Bulletin, which was founded by Peter Naur at the Paris conference in February 1959 and was later taken over with ALGOL under the IFIP umbrella. Professor van Wijngaarden, by the way, not only supported the Bulletin over long periods in general and by special contributions, but also gave substantial aid to its production and distribution.

Practically all the ALGOL authors (fathers) who were interested in the continuation of the work suggested to transfer the responsibility for the language to IFIP, which means to the Federation of National Computer Societies. And it was clear that the work should continue. To make this possible, IFIP had to create the necessary structure. After many

discussions the idea was presented and then realized in order to better match the ALGOL crew with its rather unequal national composition to the IFIP Council which necessarily was nationally structured. A two-level solution was found: a Technical Committee, into which each member society, i.e. each nation, could delegate one and only one member, and a Working Group, formally reporting to the Technical Committee, where membership was personal, only based on competency and the interest to cooperate, but accepted only in concordance with the TC, if necessary by a vote.

Naturally, there were also personal difficulties – the nomination or election of the two chairmen was a delicate problem. The solution was a diplomatic compromise. It was proposed that I chair TC 2, even if Austria was not yet an IFIP member, and Professor W.L. van der Poel was to chair WG 2.1. Thus the two bodies started work, TC 2 in Feldafing near Munich in March 1962, and WG 2.1 in Munich in August 1962; Professor van Wijngaarden was a member of both. TC 2 and WG 2.1 not only fulfilled their ALGOL 60 duties by producing and forwarding to ISO (which had also requested them) one proposal for ALGOL 60 Input/Output and one proposal for an ALGOL 60 subset, both published in 1964 [12]. A revised ALGOL 60 report was passed and published in 1963 [13]. Then work on the successor language was started. The working names were ALGOL X for the future programming language and ALGOL Y for the meta language. I will come back to this development a little later.

A few days after the March meeting in Germany, the IFIP programming language crew met again in Rome, where the International Computing Center – today the Intergovernmental Bureau for Informatics, IBI – had organized a symposium on *Symbolic Languages in Data Processing*. There Professor van Wijngaarden presented his famous paper on *Generalized ALGOL* [4], which contained most of the basic ideas he later incorporated in ALGOL X, which became ALGOL 68. Let me quote a paragraph of the introduction to this paper, a paragraph which those people who criticized ALGOL 68 later on – although they had been members of WG 2.1 – should have read more carefully. It is a kind of scientific programme of Professor van Wijngaarden's language work, his philosophy of programming, implemented by ALGOL 68 and crowned by his US paper 1981 [18].

The title "Generalized ALGOL" of this paper needs an explanation. The word ALGOL is used because of the fact that many of the concepts of the language to be described can be found, partially at least, in ALGOL. On the

other hand, the generalization goes to such an extent that the connection with ALGOL can only be appreciated by those who know ALGOL quite well.

Thus a certain alienation is clearly announced and declared to belong to the development programme. The introduction continues:

The main idea in constructing a general language, I think, is that the language should not be burdened by syntactical rules which define meaningful texts. On the contrary, the definition of the language should be the description of an automatism, a set of axioms, a machine or whatever one likes to call it, that reads and interprets a text or a program, any text for that matter, i.e. produces during the reading another text, called the value of the text so far read. This value is a text which changes continuously during the process of reading and intermediate states are just as important to know as the final value. Indeed this final value may be empty.

In order that such a language be powerful and elegant, it should not contain many concepts and it should not be defined with many words. On the contrary, by saying less one can say more, at least say more general things. Each definition in the language may restrict the set of meaningful texts. Without any definitions, however, one can only be silent in full generality. Of course, some compromise must be made in practice. This compromise has been made in ALGOL in a certain way. There are other ways, however, by which a better defined and more general language can be obtained using fewer concepts.

The paper continues with a discussion of the description of *such a syntax-free language.* It is seen as a machine M0 the working of which is described on the lid of the machine so that the user can easily find out how the language is used. If he should have doubts, he can open the machine and inspect its precise working. To his surprise, he finds that there are actually two machines inside, a preprocessor P1 and a more basic machine M1 – and so it goes on. Each machine P1 and M1 may again be made up of a preprocessor and a processor. This continues until the user finally finds a machine that cannot be opened, which is the most primitive machine for which there is no better explanation than the wording on the lid.

It is a systems theory of programming languages, elegant, general and powerful, but obviously for a certain price. Not everyone is ready to pay this price, as the course of history has shown.

1963 and 1964

In 1963, there was only one IFIP Council meeting which took place in Norway, but no spring meeting. TC 2 met in downtown Oslo, but the Council took place in the country at Gola, a typical Norwegian summer and winter resort. After a reception in Oslo the delegates went by train via Lillehammer to Harpefoss and continued by bus to the meeting place. Our Norwegian delegate, Jan Garwick, had come in his own Citroen car and took Professor van Wijngaarden, Academician Dorodnicyn and me for a ride through the beautiful, slightly rough countryside. When you compare Norway to Austria you will find that a mountain region of a certain character that might be placed, say, at 2000 m in Austria, will be found in Norway 1000 m lower, though the gulf stream makes up for much of the northern latitude. We enjoyed our ride thoroughly and had an amusing adventure.

As may happen to the best driver when he gives a lot of explanations instead of concentrating on the way he is going, Jan Garwick got lost. Since we could not loose too much time in order to reach our group again, Jan stopped at the first person we saw – there are not many in that region – and asked how we could best get back to the road to Oslo. It was not difficult to understand that obvious question in Norwegian. "You go straight ahead for a mile and then turn right," said the farmer – and pointed with his finger to the left. None of us doubted that left was the right direction. One easily says the opposite word to the one you want to say, but one rarely makes the opposite gesture. I like to tell this story to all those computer enthusiasts who propose to turn to oral input without making sure that the computer also registers the accompanying gestures.

In that year 1963, Professor van Wijngaarden joined me in a venture which should become the most frequently used model in IFIP. On the instigation of TC 2 the IFIP Council of Gola had approved the first Working Conference on *Formal Language Definition Languages.* The model envisaged that a TC should work out a list of some 50 to 80 specialists working in a field that was still new and yet developed enough for many people to work in it and to make it possible for discussions and working conferences to bring progress and consensus. In order to establish the vocabulary and to base the discussions on solid ground, there should be about 20 invited papers, distributed to the participants before or during the meeting, which constituted the essence of the proceedings. At this first

conference we also included the publication of the discussions. For this purpose there were a number of portable tape recorders in addition to the master tape on which the speakers were recorded; whoever wanted to contribute to the discussion had to wait for one of the conference assistants to come up with the recorder. That assistant pronounced the name of the speaker so that all names were recorded without exception. The auxiliary tapes were then copied onto the master tape which was then sent to the Rand Corporation in Santa Monica, where Tom Steel Jr. headed the job of transcription and editing. The proceedings appeared in 1966 and a large number was sold.

This proves the success of this first IFIP Working Conference. It is not easy to judge how much the participants profited from it. For the collaborators of the Vienna IBM Laboratory it was, however, a magnificent opportunity to meet all the people active in the field of formal definition. The contents of the papers (of course some more than others) were the basis for the development of the Vienna Definition Method to be applied for the formal definition of PL/I, not only the syntax, but also the semantics.

Professor van Wijngaarden's paper at the first IFIP Working Conference had the title *Recursive definition of syntax and semantics* [3]. Recursion was a key issue at that time and we teased him by proposing to him the title and official address *His high recursivity Professor van Wijngaarden.* Actually, the paper did not once use the word *recursive* except in the title. The paper was a kind of elaboration of an aspect of the Rome Paper on *Generalized* ALGOL and its notion of an interpreting machine consisting of preprocessor and processor, an investigation and a closer definition of their properties and their power to reduce the many concepts usually included in ALGOL-like languages to a few basic ones. *ALGOL-like*, by the way, was also a word that became a fashion at and through this conference with the culminating proposal or joke – the distinction between proposal and joke was not always clear in WG 2.1 and TC 2 – that *ALGOL was not an ALGOL-like language.*

A characteristic trait of the mood and spirit of WG 2.1 was the famous extension of the voting possibilities – I am of course not submitting that it was Professor van Wijngaarden's invention – from yes, no and abstention to a fourth choice: *I did not understand the question,* the semantics of which was essentially that the voting member for tactical reasons pretended not to understand the subject of the vote.

WG 2.1 and TC 2 were both a crew of old friends and enemies who enjoyed meeting and fighting and who gained, everyone from everyone, a lot from the official and inofficial discussions. You have only to read Duncan Fraser's closing banquet talk of the Working Conference which the editor, Tom Steel, very appropriately included in the proceedings. It had the title: *Our ultimate metalanguage*, which was a quotation from a paper by Peter Naur. This ultimate metalanguage is of course English, the computer language and the IFIP language. The Fraser talk was composed of a series of witty remarks on the subject and on the conference, out of which I quote only one sentence: *"Is your Chomsky really necessary?"*

From 1965 to 1968 the main work of both WG 2.1 and TC 2 was the development of the ALGOL successor language, first called ALGOL X, once ALGOL 67 [88], and finally ALGOL 68. It is not my intention to treat here the history of ALGOL 68. Let me proceed in comfortable disorder.

Princeton and St. Pierre

This summer the chairman of a TC 3 Working Conference in Vienna explained that they choose their meeting places according to certain parameters of which the most important were culture and food. Looking at the list of TC 2 and WG 2.1 meetings I find retrospectively that Professor van der Poel and I must have used similar parameters – restricted later by the Van Wijngaarden principle (a principle which he had submitted in IFIP several times and which said that there should never be a meeting in a place more than one hour's driving away from the next international airport). Maybe it was Princeton that he found too far away, maybe it was St. Pierre de Chartreuse, the two WG 2.1 places of 1965. For many other parameters they were fine places. Princeton recommended itself by its University and the Institute for Advanced Studies, while St. Pierre offered the opportunity to visit the distillery of the Chartreuse monks where we learned, among other things, that only four monks were introduced at one time into the secret of which and how many plants to use in the production of the Chartreuse essence from which the yellow, the green and the 72-degree Chartreuse liqueurs are made.

St. Pierre was also the starting point for another adventure with Professor van Wijngaarden.

The St. Pierre meeting was immediately prior to the last old-style

Council (from then on the spring IFIP meeting was only the Council meeting, i.e. Executive Body plus a number of trustees, while in autumn both the Council and the General Assembly had their meetings). The General Assembly was scheduled for Nice – and St. Pierre certainly did not correspond to the van Wijngaarden principle. I turned the disadvantage into an advantage: I flew to Nice and rented a car of the make I have owned since I first got a car – a Citroen. In that car I drove from Nice to Grenoble and spent 2 days with vacationing and sightseeing; I visited the Dames Coiffees, bizarre rocks, and the small town of Barcelonette and took in much of the landscape described by the French writer Jean Giono, which is the valley of the Durance. I stayed in a hotel down in Grenoble and drove up to St. Pierre several times. This, of course, was noticed by some WG 2.1 members and Aad van Wijngaarden and Fritz Bauer proposed to me to go together from St. Pierre to Nice to the General Assembly. I told them that I wanted to visit Avignon, the city of the popes, which I had never seen before. They quite agreed to this and said they would come along, if only we went together to Nice. Can you resist such a cordial invitation? No, you cannot. And with two mathematicians you cannot start off at six in the morning, as I had intended, but at 9:30, which is the proper time, and not in the middle of the night. Thus I picked them up at St. Pierre on October 30, the Saturday before a long weekend – November 1 (which was a Monday) being a holiday in France, which will be important for my story – and we headed for Avignon.

We went down the main road to the Rhone valley and again and again passed signposts indicating the roads to passes which are called 'col' in Southern France. "Let us go up to one col," Bauer and Van Wijngaarden said. "I want to go to Avignon," I answered, "and a detour will cost a lot of time." "Alright, alright," they tried to calm me, "but a little detour will not take that much time." They consulted a map and saw that one of the next cols would permit us to continue our way to Avignon in a relatively straight line. Who was I to point out that the map did not show the minor details such as bends and gradients? We turned left and mounted to the col. The weather was fine, the air was clear, the view was splendid. We collected alpine plants and had a coffee after we had passed the tunnel at the top.

But at the first bend on our way downwards a red light appeared on the dashboard of the Citroen: hydraulic trouble. It disappeared, but reappeared again after some time. When we had negotiated half the way

down it was more often on than off and steering became harder and harder. Being in France, the hope of finding a Citroen repairshop was a logical one, and indeed we saw a sign directing us to a repairshop in a town called Die – which was not really in our direction, but was it not better to aim for the nearest mechanic? The red light was on all the time, but our luck held and we not only found Die but also the repairshop immediately. "It can't be anything serious, please help us as fast as you can, because we want to reach Avignon in time," we asked him. The face of the man indicated delay. At that moment Aad gave a cry: he had seen the hydraulic liquid escape in a stream as thick as a finger. "No chance," said the mechanic. "And there is a long weekend to come. My son has already gone and I will close in five minutes. We will start on the car on Tuesday morning." All our entreaties did not help. We left the car at the shop and started looking for a hotel room. I must explain that Die owes its fame to the single fact that it is the place where Hannibal started out on his treck across the Alps. Nothing spectacular has happened since then and thus the hotel situation is somewhat unlike Grenoble or Nice; the few inns we found were practically sold out. Only by extraordinary good luck and with the help of the mechanic we finally got a single and a double room. Can you imagine how happy I was? No more hope to see Avignon, and perhaps we would even be too late for a part of the meetings. I was furious and apathetic at the same time. This was the moment when Van Wijngaarden showed his strength. He gave me a three-sentence lecture after which I was neither furious nor apathetic any more – all the three of us were ready for a nice weekend in Die. We visited the ruins dating back to Hannibal's time, drank wine called Clairette de Die, and had a fine dinner. The next morning, Professor van Wijngaarden developed the algorithm for the Fly and the Spider on the paper cover of the breakfast table – a copy is shown on the next page.

Then we walked back to our mechanic and with a lot of good words we could convince him to start working on the car despite the holiday and without his son.

Avignon was lost for me, and I have not seen it to the present day, but we drove gaily down to Nice, that is with the exception of one incident. Bauer – being also a Citroen fan – wanted to drive for a while, not to Van Wijngaarden's pleasure, by the way. Suddenly Bauer was stopped by a policeman who wanted to give him a ticket; he said that Bauer would have passed another car, hadn't he seen the gendarme at the very last moment.

Die, 1 nov 1965.

```
begin integer n; n := read;
  begin integer array move [1:n, 1:n];
    boolean array win, lose, path [1:n, 1:n];
    integer s, f, t, g; Boolean ready;
    for s := 1 step 1 until n do
    for f := 1 step 1 until n do
    begin path [s,f] := read = 1; move [s,f] := if s=f then s else 0;
      win [s,f] := lose [s,f] := s=f
    end;
spider: ready := true;
    for s := 1 step 1 until n do
    for f := 1 step 1 until n do
  begin if ¬ win [s,f] then
      for t := 1 step 1 until n do
  if path [s,t] ∧ lose [t,f] then
        begin ready := false;
          win [s,f] := true;
          move [s,f] := t; goto hurray
        end
      hurray:
    end;
    if ready then goto printout;
fly: ready := true;
    for s := 1 step 1 until n do
    for f := 1 step 1 until n do
  begin if ¬ lose [s,f] then
      for g := 1 step 1 until n do
      if path [f,g] ∧ ¬ win [s,g] then goto thanks heaven;
      lose [s,f] := true; ready := false; thanks heaven:
    end;
    if ¬ ready then goto spider;
printout: for s := 1 step 1 until n do
      for f := 1 step 1 until n do
      print (move [s,f])
end
```

Then he started to grumble over the car papers. This was the point where I joined the discussion. "You shut up", I was told by the gendarme. "But it is I who has rented the car," I retorted, and with carefully selected Austrian arguments I managed to convince him in my very best French to forget the ticket. And so we arrived in Nice in due time for the first evening gathering.

1965 to 1969: ALGOL 68

I must repeat: it is not the intention of this paper to give a technical history of ALGOL 68. This would be a scientific project of quite some extent – a job somebody should undertake, however, before it is too late to collect the material completely (I invite you to submit a comprehensive paper for the Annals for the History of Computing).

Professor Turski will revisit ALGOL 68 in his closing lecture and he will certainly do more than only paraphrase the thin skeleton of the development I intend to sketch here.

The intensive development work of ALGOL 68 extended over the years from 1965 to 1969. At the Princeton meeting of WG 2.1 in May 1965, an invitation for written descriptions of a language proposal was extended. At the meeting in St. Pierre three full descriptions were presented, by Niklaus Wirth, by Gerhard Seegmüller and by Professor van Wijngaarden [76]. Tony Hoare and Peter Naur presented significant papers. A four-man subcommittee consisting of Professor van Wijngaarden, Tony Hoare, Gerhard Seegmüller and Peter Naur was charged to bring the proposal into one common shape. The subcommittee met at Kootwijk in April and WG 2.1 in Warsaw in October, but the balance they had wanted was not achieved. From 1967 onwards it became clear that the Amsterdam group was gaining the absolute leadership, with one of the reasons being the amount of work they were investing into the new language. They had prepared a draft proposal for the May meeting in Zandvoort [88] which was followed by a next version distributed in November [92]. 1968 brought the culmination both of the work and of the number of meetings. The June meeting in Tirrenia near Pisa had an Amsterdam draft of January [93], the July meeting in North Berwick its follower from July [95], and in October the Mathematisch Centrum issued already the next version [99]; on the table of the December meeting in Munich the final version was presented

[100]. Only those who participated in this giant effort can judge the intensity and strain of the work. But at the same time criticism and tension spread, there was more fighting than agreement, and it was easy to predict that the December meeting in Munich would be a decisive and shaken event. In a circular letter to TC 2 and WG 2.1 of October 18 I tried to point out very clearly the situation and the responsibilities of the two bodies. I indicated the choices I saw for the Munich meeting:

(1) The language produced and described in the MRs would have to be the next ALGOL, or else WG 2.1 would have to decide that the editor and the authors had essentially failed to carry out their commission;

(2) WG 2.1 might decide that the editor and the authors had carried out their commission, but that the whole enterprise had become a failure;

(3) WG 2.1 might decide that the content of the language was acceptable, but that its description was unacceptable. In that case, another description would have to be produced;

(4) WG 2.1 might decide that the final document of the editor was a first edition and that a further edition should appear;

(5) WG 2.1 might decide that the final document of the editor was without any restriction the report on the new ALGOL.

These choices indicate the controversy within WG 2.1 and the criticism from outside. WG 2.1 in Munich accepted ALGOL 68, but there was an opposing minority report and TC 2 presented the language to IFIP for acceptance with an extremely carefully worded cover letter. This letter appreciated the magnitude and difficulty of the task, but mentioned the minority report and added that the language was submitted to IFIP for publication as one of the possible approaches to the subject rather than a final answer; it said, however, that the work had reached the proper stage for submission to the crucial tests of implementation and subsequent use by the computing community. With this cover letter ALGOL 68 became official, but the group split. WG 2.1 continued to take care of ALGOL 68. In June 1970 TC 2 and WG 2.1 set up a Working Conference on ALGOL 68 Implementation [7], and a few year later WG 2.1 presented a revised edition of the ALGOL 68 Report [8] it had produced, again under the leadership of Professor van Wijngaarden. The WG 2.1 dissidents formed, in response to an invitation of TC 2, Working Group 2.3, constituted under the chairmanship of Mike Woodger in 1969 with the name of *Programming Methodology* and with the scope *Support and Tools for Program Composition.*

At the spring Council a paper on ALGOL 68 was invited for the Edinburgh Congress 68, and this lecture by Professor van Wijngaarden found so much interest that more people had to go away than found room in the lecture hall.

What had happened in the late fifties, namely that the programming community was split into the FORTRAN and ALGOL cultures, and later in addition into the COBOL culture – in a simplified manner one might speak of the industrial, academic and commercial subfields of programming in spite of the considerable overlaps – was repeated within the university community and today we have ALGOL 60, ALGOL 68 and PASCAL in parallel (with unequal shares).

The story of the Babylonian language confusion is as contemporary as can be. It is a basic law of human thinking that giant enterprises – and computer programming is a giant enterprise – develop different mentalities which in turn and in feedback lead to the development of different languages. This multitude is a fact of life. We must accept the diversity. Every language must be judged by its merits. It is beyond doubt that ALGOL 68 has, in certain aspects, more power than any other language. Why ALGOL 68 did not have the impetus of ALGOL 60 will be judged – in appropriate distance – by history. The unique and outstanding role of Professor van Wijngaarden, the incredibly concentrated and immense amount of work done by him and his collaborators at the Mathematisch Centrum, with Professor Peck, Professor Mailloux and other Canadian 'guest workers', has already now filled many pages in the books of history.

1967: TC 1 and WG 1.1

In 1967 Professor van Wijngaarden took over TC 1, which – under the chairmanship of G. Tootill and A.R. Wilde – had tried since 1962 to carry out what had looked like a superidea of Ike Auerbach: the compilation of a multilingual glossary for information processing systems and related subjects. The proposal was that a collection of definitions and concepts and terms should be produced, the keywords being arranged in a kind of decimal classification. Then, for the different languages, it was simply necessary to establish the translations of the keywords and so the information processing community would soon and easily have a multilingual, well-defined dictionary. It would be sufficient to buy the

English glossary and the keyword translations for, say Spanish and Hungarian, to get the correct translation of technical texts [14].

It was indeed very astonishing that this idea should proceed so slowly, in particular in the early years, where the number of terms was not yet as large as it is today, and where not very many specialized dictionaries were on the market. The member societies obviously did not support the project strongly enough, only very few attempts of translations of keywords became known, and most of them did not follow the rules. North-Holland have only one non-English dictionary on their list, the German Fachwörterbuch of 1968 [15].

Thus it seemed a reasonable step that a General Assembly member that had chaired the Publications Committee should try to save the enterprise. Professor van Wijngaarden let himself be convinced to do it, although he realized how bad the situation was. He separated the Technical Committee and its general scope from the direct definition work and for the latter purpose created WG 1.1.

But in spite of his efforts, the situation did not improve. In 1973 he submitted the following letter of resignation to the General Assembly:

TC 1 *Terminology*

Since the General Assembly Meeting in Sofia in October 1972, the progress in the translation, by national groups, of the terms in the 2nd Volume of the IFIP Guide to Concepts and Terms in Data Processing has been regrettably small. Although a request has been sent out to all national representatives of IFIP and to all TC 1 members to supply TC 1 with the translations of the terms, so far only the translations into Finnish, Dutch, Swedish, Czech, French and Slovak have been received. Obviously, with languages as German, Italian, Russian, Spanish, and so on, missing, Volume 2 cannot appear.

Since the chairman of TC 1 obviously has failed to raise sufficient cooperation in IFIP circles he offers his resignation as such.

A. van Wijngaarden

In order to check on its own operations, IFIP had set up review committees for its various bodies, and at that time it happened that such a committee was reviewing TC 1. In its report to the next IFIP Council,

Professor van Wijngaarden's proposal to *hibernate* both TC 1 and WG 1.1, that is to inactivate them, but keep them in the lists so that they might be revived when needed, was brought forward. Unfortunately, the General Assembly in Stockholm did not follow this proposal; the bodies were discontinued. IFIP would now need a new edition of the glossary. If they had followed Professor van Wijngaarden's proposal they might simply dehibernate TC 1 and WG 1.1 and the work would probably proceed much faster.

Professor van Wijngaarden, just to mention this, was also in the Site Selection Committee for IFIP Congresses that recommended Stockholm as the site for Congress 74.

1970: 10 Years of IFIP

Dov Chevion considers it his duty to remind IFIP of its anniversaries; he brought up the proposal to celebrate the 10th anniversary and he put the 20th on the agenda. The real job, however, is to find somebody to organize the celebration, which is long and hard work. For the 20th anniversary IFIP failed to find a volunteer; and in view of the extraordinary event of 1980, namely the 8th Congress which was carried through as a Pacific event in two hemispheres, in two seasons, on two continents and in two big countries, it was decided not to insist too strongly on the anniversary idea and rather to celebrate the 25th anniversary. A volunteer has been found in the meantime: Professor Bauer in Munich.

The volunteer for 1970 was Professor van Wijngaarden, the location was Amsterdam, and the time a day during the General Assembly 1970.

What Professor van Wijngaarden achieved was an event of national and international importance, impressive and a model for the future. The programme included two opening addresses by representatives of the United Nations, Mr. Malecki of UNESCO and Mr. Gresford of UN New York, seven papers by active or past IFIP officers and two papers by representative managers of the industry.

Academician Dorodnicyn gave an overview over the first 10 years of IFIP and then announced the election of I.L. Auerbach, the first IFIP President, as IFIP Honorary Member.

The six other IFIP speakers were Professor Speiser, second IFIP President, Dov Chevion, Professor Bauer, E.L. Harder, Professor van

Wijngaarden and myself. The outside papers were by G.E. Jones, Senior Vicepresident of IBM, and by Professor Casimir of Philips.

Almost all the papers have been published in the volume *The Skyline of Information Processing* [16], so I need not describe their contents. But I want to add two remarks. One is a quotation from Professor Speiser's paper which seems to me as worth of being quoted as often as possible. He mentioned the blackout which in 1965 deprived the entire North-East of the United States of electricity for several hours. *The sequence of events which led to the disaster has been reconstructed with great accuracy. In the course of these studies it was learned that in systems of this high degree of complexity there can occur conditions of instability, even under perfectly normal operating circumstances, in which an almost arbitrarily small perturbation can have catastrophic effects* [16, p. 32]. The second remark is also concerned with a perturbation. Into my own contribution on *Some philosophical aspects* I should have invested a lot of the effort I put into it afterwards before the lecture. The main effect of such a state of affairs is, of course, that your manuscript becomes much too long: all of you know the excuse – I had not the time to write a short letter, so you get a long one. The General Assembly is more than a full-time job for the members of the Executive Body, and there was no chance to do in Amsterdam what I should have done in Vienna. Knowing all this, I fell into the second of the two alternatives that wait for the speaker: to fly above the manuscript, or to swim behind it. I did not only swim, I drowned. In doing so I lagged hopelessly behind the speaking time allotted to me, and upset Professor van Wijngaarden, his speech (which came next), and his time schedule completely.

Eleven years later, I apologize once more and regret my imperfection. And since I have embarked on apologies, I want to generalize them on behalf of IFIP: we are all imperfect and on many occasions we have made our distinguished member and friend Professor van Wijngaarden angry. This is the opportunity to present our apologies to him. But I am sure he will wave them aside. Not only because he realizes that he, too, has occasionally upset others, but mainly because he forgives immediately.

1979: Urgench

This account started with conferences outside the range of IFIP. Let me

begin also the last chapter with a conference outside IFIP, the last one where we met before this symposium. It was the meeting on *Algorithms in Modern Mathematics and its Applications, dedicated to al-Khorezmi*, in Urgench, the capital of Khorezm, a region in Uzbekistan. The place was chosen because the Arab mathematician al-Khorezmi, from whose name the term algorithm was derived, was of Khorezmian origin. You can be sure that it is only the absolute time limit that prevents me from summarizing my speech on al-Khorezmi and his country, which I gave in Urgench; the countryside is spectacular and the city of Khiva near Urgench is the most impressive and best-preserved Central Asian town (we saw it during an excursion in the course of the symposium). Fortunately, the proceedings of the Urgench conference will appear soon, and thus you cannot only read my paper but also that of Professor van Wijngaarden [17]* which he read at that conference and which I consider just as important as his paper on *Generalized* ALGOL, although it was more a sketch than a completed paper (which I hope to see in not too distant a future).

The basic idea of the Urgench paper was that by a further step of generalization one and the same, but highly general language structure permits not only, like Generalized ALGOL, the formulation of the problem and the *gestaltung* of the language in which one wants to formulate the problem – choosing, of course, the best formulation and language one can think of within the general structure, but also forming the automaton, the particular computing structure, on which the given problem is processed – again matched to the optimum.

In this latest step of intellectual development of a computer pioneer, one can recognize the superpower of generalization, but also the disadvantage by which one has to buy extreme generalization. The computer, whether we like it or not, has also the contrary tendency to particularize, to save the user from what a generalizer of the academic strength of Professor van Wijngaarden considers the essence of computing work: the narrowing down from the most general possibilities of the general purpose computer to the particular language, algorithm and computer, which finally carries out the job. In the daily life of today people expect the computer to even press the button for them which starts the execution of the job.

If ALGOL 60 was a programming language for computer professors, ALGOL 68 was a language for language professors and the latest proposal

* *Note by the editors*: The paper referred to here will not appear in ref. [17] but in ref. [18].

of highest generality is a language for generalization professors, a very small class, of which Professor van Wijngaarden is one of the most prominent representatives. Progress in science has never come from particularization, but from generalization, from the recognition of common and general properties and laws, from reduction to the ultimate invariables. All his life Professor van Wijngaarden has contributed to this progress, by hard work in many more fields than IFIP, of which I have described here only what I was able to see and remember.

Professor van Wijngaarden

I am extremely proud that the Silvercore, the symbol of recognition and service award of IFIP, bestowed on Professor van Wijngaarden in 1974, carries my signature. The plaque is certainly very modest, an all too modest sign compared to everything Professor van Wijngaarden has done for IFIP, for the examples he has set, for the model and challenge his presence and contributions in the many IFIP bodies have meant for all of us.

All abstraction and formalization that finally make up the body of science and technology separate themselves from the people who have created them. Maybe that a name remains attached to a law or a language – after less than one generation, the name is not much more than a keyword. The real importance, however, of human life and its incorporation in a field of science and technology, is not on the abstract and formal side, but in the personal style and accent, in the human and heartfelt involvement which distinguishes, for example, a teacher from a teaching machine. The next generation cannot find this dimension in the papers and programs, in the minutes and protocols. But the friends and students know it better than they can ever express: they are aware of a distributed monument of Professor van Wijngaarden which no sculpture in front of the Amsterdam railway station or the Schiphol airport building can bring to light.

This symposium is part of the distributed abstract monument just as well as the many documents and publications he has produced and by which he has influenced IFIP.

Professor van Wijngaarden can look back at a giant lifework extending far beyond the IFIP universe I have described. There are the mathematical contributions and there is the Mathematisch Centrum with its industrial

impact. There are the many people, students, friends and readers whose thoughts and achievements he has influenced, coined and sped on. All descriptions must remain behind reality, all words imperfect.

And yet it is appropriate to use this opportunity to express on behalf of IFIP as well as on my own behalf the infinite thanks and appreciation to a man of the first hour, and of 25 subsequent years in IFIP, to enumerate once more his contributions and to wish him a pleasant and successful evening of his life.

Retirement from a position or job for Professor van Wijngaarden has never been transition to inactivity, and will never be inactivity.

We are looking forward at this meeting to all the things by which he will continue to surprise us.

References

[1] A. van Wijngaarden, Moderne Rechenautomaten in den Niederlanden (Auszug aus dem Vortrag), Nachrichtentechnische Fachberichte 4 (1956) 60–61.

[2] A. van Wijngaarden, The state of computer circuits containing memory elements, in: H. Aiken (Ed.), Proc. Int. Symp. on the Theory of Switching, Cambridge, MA, 2–5 April 1957. Annals of the Computation Laboratory, Vol. XXX (Harvard University Press, Cambridge, MA, 1959) pp. 213–224.

[3] A. van Wijngaarden, Switching and programming, in: H. Aiken and W.F. Main (Eds.), Switching Theory in Space Technology (Stanford University Press, Stanford, CA, 1963) pp. 275–283.

[4] A. van Wijngaarden, Generalized ALGOL, in: Symbolic Languages in Data Processing. Proceedings of the Symposium organized and edited by the International Computation Center Rome, March 26–31, 1962 (Gordon and Breach, New York, 1962) pp. 409–419.

[5] A. van Wijngaarden, Recursive definition of syntax and semantics, in: T.B. Steel Jr. (Ed.), Formal Language Description Languages for Computer Programming, Proceedings of the IFIP Working Conference (North-Holland, Amsterdam, 1966) pp. 13–24.

[6] A. van Wijngaarden (Ed.), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Report on the algorithmic language ALGOL 68, Numer. Math. 14 (1969) 79–218.

[7] J.E.L. Peck (Ed.), ALGOL 68 implementation, Proceedings of the IFIP Working Conference in Munich, July 20–24, 1970 (North-Holland, Amsterdam, 1971) 375 pp.

[8] A. van Wijngaarden et al. (Eds.), Revised report on the algorithmic language ALGOL 68, Techn. Rep. TR 74-3. (March 1974) 226 pp. Dept of Computer Science, The University of Alberta, Edmonton, Alb.; Acta Informatica 5 (1975) in parts 1 through 3; ALGOL Bull. Supplement No. 36; as book: (Springer-Verlag, Berlin, 1976) 236 pp.

[9] C.H. Lindsey and S.G. van der Meulen, Informal Introduction to ALGOL 68 (North Holland, Amsterdam, 1971) 382 pp. (Revised Edition 1977).

[10] A.J. Perlis and K. Samelson (Eds.), Preliminary report – International algebraic language, Comm. ACM 1 (12) (1958) 8–22; Numer. Math. 1 (1959) 41–60.

[11] P. Naur (Ed.), Report on the algorithmic language ALGOL 60, ALGOL Bull. Suppl. No. 2 (March 1960), Numer. Math. 2 (1960) 106–136; Comm. ACM 3 (5) (1960) 299–314.

[12] Report on Subset ALGOL 60 (IFIP); Report on Input-Output-Procedure for ALGOL 60; both in: Comm. ACM 6 (1963) 626 ff; Numer. Math. 6 (1964) 454–462.

[13] P. Naur (Ed.): Revised Report on the Algorithmic Language ALGOL 60. –
In: Numerische Mathematik 4 (1963); pp. 420–453.
Comm. ACM 6 (1963), No. 1, pp. 1–17.
The Computer Journal 5 (1963), pp. 349–367.

[14] IFIP/ICC Vocabulary of Information Processing (North-Holland, Amsterdam, 1966) (Third Edition 1968) 208 pp. I.H. Gould (Ed.), IFIP Guide to Concepts and Terms in Data Processing (North-Holland, Amsterdam, 1971) 161 pp.

[15] IFIP Fachwörterbuch der Informationsverarbeitung (North-Holland, Amsterdam, 1968) 296 pp.

[16] H. Zemanek (Ed.), The skyline of information processing, Proceedings of the 10th Anniversary Celebrations of IFIP, Amsterdam, October 25, 1970 (North-Holland, Amsterdam, 1972) 146 pp.

[17] A.P. Ershov and D. Knuth (Eds.), Proc. Int. Symp. in Urgench, Khorezm (Uzbek SSR), September 1979, Springer Lecture Notes in Information Processing (Springer-Verlag, Heidelberg, 1981).

[18] A. van Wijngaarden, Languageless programming, in: Relationship between numerical computation and programming languages, Proc. IFIP WG 2.5, working conference, Boulder, CO, U.S.A. (August 1981), North-Holland, Amsterdam, to appear.

[76] A. van Wijngaarden, Orthogonal design and description of a formal language, Mathematisch Centrum, Amsterdam, MR 76 (October 1965).

[88] A. van Wijngaarden, B.J. Mailloux and J.E.L. Peck, A draft proposal for the algorithmic language ALGOL 67, Mathematisch Centrum, Amsterdam, MR 88 (May 1967).

[92] A. van Wijngaarden, B.J. Mailloux and J.E.L. Peck, A draft proposal for the algorithmic language ALGOL 68, Mathematisch Centrum, Amsterdam, MR 92 (November 1967).

[93] A. van Wijngaarden (Ed.), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Draft report on the algorithmic Language ALGOL 68, Supplement to ALGOL Bulletin 26, Mathematisch Centrum, Amsterdam, MR 93 (January 1968).

[95] A. van Wijngaarden (Ed.), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Working document on the algorithmic language ALGOL 68, Mathematisch Centrum, Amsterdam, MR 95 (July 1968).

[99] A. van Wijngaarden (Ed.), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Penultimate draft report on the algorithmic language ALGOL 68, Mathematisch Centrum, Amsterdam, MR 99 (October 1968).

[100] A. van Wijngaarden (Ed.), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Final draft report on the algorithmic language ALGOL 68, Mathematisch Centrum, Amsterdam, MR 100 (December 1968).

[101] A. van Wijngaarden (Ed.), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Report on the algorithmic language ALGOL 68, Mathematisch Centrum, Amsterdam, MR 101 (October 1969).

Algorithmic Languages, de Bakker/van Vliet (eds.)

PLAIN: An Algorithmic Language for Interactive Information Systems*

Anthony I. Wasserman

Medical Information Science, University of California, San Francisco, CA 94143, U.S.A.

Reind P. van de Riet and Martin L. Kersten

Wiskundig Seminarium, Vrije Universiteit, Amsterdam, The Netherlands

The programming language PLAIN has been designed to provide an effective tool for the systematic construction of interactive information systems. To achieve this goal, PLAIN started with a PASCAL-like framework and incorporated features for the construction of interactive programs, including string handling, pattern specification and matching, input/output, exception handling, and relational data base definition and management. Additional features have also been incorporated to support a systematic approach to programming, with particular attention given to issues of modularity and data abstraction. This paper describes some of the innovative aspects of PLAIN, shows how they have been synthesized into the language, and illustrates how they are used in the creation of interactive information systems.

1. The Design Context of PLAIN

The User Software Engineering (USE) project [25,27,29] was undertaken in 1975 with the goal of providing application developers with a methodology and programming environment to support the systematic creation of interactive information systems. Interactive information systems may be characterized in the following way:

(1) the user repeatedly types some input, e.g., a command;

* This work was supported in part by National Science Foundation grant MCS78-26287 and by The Netherlands Organization for the Advancement of Pure Research (ZWO) (grant 00-62-139). Computing support for text preparation was provided by U.S. National Institutes of Health grant RR-1081 to the UCSF Computer Graphics Laboratory, Principal Investigator: Prof. Robert Langridge.

(2) this input is decoded and parsed; if it is incorrect, a diagnostic message is presented to the user, who then provides alternative input;

(3) the input is subjected to various semantic checks, which may also produce diagnostic messages;

(4) if the input is validated, then some program action is taken, typically an access to or modification of some item(s) in a database, during which time output messages may be provided to the user.

A study of languages and systems available for the construction of interactive programs [23] led to the conclusion that "the programming languages designed explicitly for interaction do not [have the structure] for creating modular, well-structured software". With that in mind, the programming language PLAIN (Programming LAnguage for INteraction) became the first tool to be designed in the USE environment.

The design of PLAIN was carried out in parallel with many other language designs, including CLU [11], ALPHARD [34], GYPSY [3], EUCLID [9], and ADA [8]. These languages all have similar objectives (though with differing emphases) of support for data abstraction, support for system modularity, support for program readability, support for testing and/or verification of programs, and the imposition of greater discipline on the programmer. In addition, each of these languages draws heavily on the ALGOL family of languages, particularly PASCAL [33], and on one another. Of these languages, though, only PLAIN addresses the application area of interactive programs and their need for database facilities.

2. PLAIN Design Goals and Features for Interactive Programs

From the outset, the contribution of PLAIN was seen to be not so much the introduction of new language features, but rather a synthesis of features whose *interrelationships* would lead to a useful tool for such application programs. The approach was to make innovations to support interactive programs and to adhere closely to well-understood approaches for other features.

Essential capabilities for the creation of interactive programs were identified, including:

(1) Data base management. The language must deal with data bases and with operations performed on data bases, as well as with more primitive file concepts.

(2) String handling. Interactive programs involve large amounts of text processing, particularly user-program dialogue.

(3) Exception handling. User errors must be expected, but the user should not be adversely affected.

(4) Pattern specification and matching. Many interactive programs depend on a specific text pattern, e.g., a command, to determine program action.

PLAIN provides these capabilities by synthesizing a PASCAL-like framework with necessary features for interactive programs, including the following:

(1) data of type relation and associated relational algebra-like operators that provide a data base management facility;

(2) data of type char and type string, providing for both fixed and variable length strings;

(3) procedure-oriented exception handling, including a time exception;

(4) pattern specification primitives and pattern matching operations;

(5) sequential and direct access files;

(6) input/output operations, possibly involving patterns and files;

(7) access to external objects, such as data bases.

Space limitations make it impossible to give a complete description of the language or even the above features. A complete language description may be found in the Revised Report [30], and explanations of various other aspects of PLAIN may be found in [26, 28, 31, 32]. In this paper, we wish to summarize the motivations behind the design of features for database management, string handling, pattern specification and matching, and exception handling, and then to show how they work together in the construction of interactive information systems.

3. Database Management in PLAIN

A key design goal for PLAIN was to support database management explicitly, rather than working with the lower-level concept of a file as it exists in many languages or relying on traditional approaches to programming languages/data base interfaces. Problems with embedded query languages and with host language interfaces were noted and the need for a *unified* approach to programming languages and data base management was emphasized, so that "it becomes possible to achieve a level of con-

sistency in syntax and semantics" and so that "both type checking and data independence can be achieved" [24].

Other efforts have been made to extend programming language with database notions [1, 2, 20, 21, 22], but these suffer from one or more of the unpleasant problems of language/data management interaction identified in [17], including the difficulty of performing type checking, the tradeoffs between interpretation and compilation, the need to support data abstractions in the database environment, and the unattractive nature of combining nonprocedural data management sublanguages in procedural programming languages.

Two key goals were established for the data definition and management facilities of PLAIN:

(1) Use existing language structures wherever possible. Uniformity of syntax is important so that the data management operations will blend cleanly with other language features. Thus, traditional programming concepts of types and variables should be applicable to database declarations, and the operations on databases should be procedural, in keeping with the procedural nature of the language.

(2) Minimize the number of features added to the language specifically for database management. This objective follows directly from the first objective. Instead of providing a large set of database operations, the decision was made to strive for a compact, yet complete, set of operations. This decision was made with the understanding that the price of the language simplicity would be an increase in the amount of text needed to express complicated data management operations.

These goals pointed clearly toward use of the relational model of data [5] as the basis for database management in PLAIN. From a syntax standpoint, it is possible to exploit the similarity in notation between records and tuples, as was also done by Schmidt [21]. From a language axiomatization standpoint, relations were also the best choice because of their mathematical foundations. Although it was recognized that the relational model is weak in specifying the semantics of the database, it seemed that the potential advantages of the model greatly outweighed the disadvantages for programming language design and implementation.

A data base type definition specifies a structure consisting of an arbitrary number of record occurrences (each called a 'tuple') where each tuple consists of a fixed number of components (called 'attributes'). PLAIN supports two kinds of data base type declarations: **relation** and **marking**. A

relation is a set of tuples and has the property that all tuples are unique; the definition of a relation includes the specification of a non-null set of **key** attributes that uniquely identifies a tuple. A **marking** is a set of referenced tuples from one or more relations. Markings are used to store intermediate results during operations on relations. They play much the same role in database management that temporary variables play in complicated arithmetic calculations. Thus, one may declare variables to be of type relation, using a syntax similar to that for records in PASCAL, or of type marking. Similarly, all attributes must be declared of some type; permissible types are simple types, including scalars, fixed length strings (**type** char[n]), and variable length strings (**type** string). Database management operations are provided at the item (attribute) level, the tuple level, and the relation level.

3.1. *Item level operations*

At the lowest level of relation access and manipulation, it is possible to name individual tuples within a relation through a *tuple designator.* If a relation of degree N (N attributes) has M key attributes, where $M \leq N$, specification of values for the M key attributes designates a unique tuple of the relation (or no tuple at all). The syntax for a tuple designator is of the form

relation–name [key-value-list]

which permits an attribute of a relation to be designated with the notation

relation–name [key-value-list] . attribute–name.

This mechanism provides two important benefits. First, it is an *associative* addressing mechanism for databases that can be used to obtain single tuples and single attribute values from relations, yielding a clean solution to the problem of converting objects from type **relation** to the underlying attribute type. This makes it possible to perform complete type checking on items in the database, since each attribute must be declared with a type.

Second, it achieves integration at the lowest level between language concepts and database concepts, since the attribute designator may be used in arbitrary expressions throughout a program. Information stored in a relation can be used to declare the dimensions of arrays, to provide a bound on the number of iterations in a loop, or to supply the text for an output message.

3.2. *Tuple operations*

At the tuple level, it is possible to insert tuples and to remove tuples one at a time from a relation. One may simply construct a new tuple by designating a record variable or by specifying values for the attributes of the tuple. The tuple insertion assignment is designated by ':+', while tuple deletion is given by ':–'.

One may also iterate over the tuples of a relation or marking by use of the **foreach** clause in a **loop** statement. The effect of the **foreach** is to permit access to individual tuples in much the same way that iteration is performed over other types of variables.

3.3. *Relation level operations*

High level operations on relations and markings permit the construction of database expressions and the assignment of the expression to a relation or marking variable. The operations supported are selection on a condition (**where**), projection (⇒), natural join on two attributes of the *same type* (**join**), and the set-oriented operations of intersection, union, and difference. The language syntax limits the complexity of database expressions, making it necessary to decompose complicated operations into several steps (perhaps creating markings). The rationalization for this approach is presented in detail in [19].

In summary, PLAIN makes a number of advances toward achieving an effective integration between modern notions of programming languages and facilities for database definition and manipulation. In particular, the procedurality of the operations, the ability to perform type checking on database objects, and the associative access feature are the principal unifying ideas.

4. String Handling and Pattern Matching

Features for string handling and pattern matching were also seen as essential for PLAIN. In addition to providing strings as a data type, it is also necessary to provide tools for checking the conformity of strings to predetermined patterns, particularly for user input. User input must be checked for conformity to the syntactic rules and must then be checked to see that it is meaningful in the context of the input. A numeric input might

fail not only for reasons of invalid characters, but also for arithmetic overflow, arithmetic underflow, or because the numeric value was not a meaningful value for the corresponding data element.

PLAIN provides for the built-in simple type char (as in PASCAL, ADA, and other similar languages) and for the built-in structured type string. Variables of type char or array of char permit *fixed length* string processing, while variables of type string provide for *variable length* strings. String concatenation is provided with the binary operator '++' returning an array of type char or a string, depending on the operands. String contains is provided with the operator '$'; for strings *a* and *b,* the value of *a*$*b* is true iff the string *b* is contained in *a*. String follows (lexical ordering) is provided with the operator '>>'; for strings *a* and *b*, the value of $a>>b$ is true iff the lexicographic order of *a* follows *b* in the ASCII collating sequence. The remaining string operations are provided through functions, including length, string searching, substring extraction, insertion, deletion, and replacement.

The key observation for successful handling of user input was to see user inputs as *languages* subject to various kinds of syntactic and semantic rules. In short, one can define a grammar that describes the valid syntax for a given user input.

From that point, it became possible to identify some goals for the inclusion of pattern processing mechanisms in PLAIN, including the following:

(1) simplicity, comparable to that of MUMPS patterns, rather than to the more powerful and general SNOBOL 4 patterns;

(2) the pattern facilities should simplify not only the syntactic checking of user input, but also any subsequent semantic checking;

(3) certain common patterns should be predefined, i.e., 'built into' the language;

(4) the pattern facilities should be usable for control of program output as well, so that it would not be necessary to include a totally separate output management mechanism;

(5) the power of the pattern specification and pattern matching should make it possible to recognize a large class of possible user inputs, such as specified by a context-free (Type II) grammar.

The key idea behind pattern specification and matching in PLAIN was to provide a simple mechanism whereby the programmer could define the grammar for a language, and then use built-in operators to determine the

match between a pattern and a string defined by the grammar.

The **pattern** declaration facility permits patterns and pattern sets to be declared. In a pattern, all elements are required for pattern matching, while in a pattern set, only one of an alternative list of patterns is required for matching. In both cases, the declarations are static and, unlike SNOBOL4, it is not possible to create patterns dynamically.

A pattern is composed of a list of pattern elements, which may be string literals, subranges of characters, or the name(s) of other patterns, including pattern sets. Each pattern element may be preceded by a repetition count, which may be definite (a positive integer), or indefinite. The indefinite cases are '*' for zero or more instances, and '.' for one or more instances. In the absence of a repetition count, a default count of one is assumed.

Many common pattern matching cases are covered by predefined patterns in PLAIN. These patterns include A for alphabetic characters, N for numerics, P for punctuation, I for a (signed) integer, X for blank, and S for string, which matches anything.

A simple example of a pattern definition is given by the patterns

```
bookid = (10N);
chkout = ('out', . X, bookid, '/', I)
```

they would match the string 'out 9023633407/12554'. Note that chkout contains the name of another pattern, bookid, as well as string literals and predefined pattern names.

Such pattern names can be combined into other patterns and pattern sets. Thus, the pattern chkout might be an alternative in the pattern set

```
command = [chkout, checkin, reserve, status, quit]
```

where each of the patterns represents the permissible user input for the various commands in the system. (If a string matches more than one pattern in the pattern set, the leftmost matching alternative is selected.)

A more complex example can be given by combining patterns and pattern sets to define a class of strings representing permissible ways to input a date, showing that patterns and pattern sets may be nested.

```
date = [form1, form2, form3];
form1 = (one-or-two, sep, one-or-two, sep, two-or-four);
form2 = (one-or-two, 1X, month, 1X, two-or-four);
form3 = (month, X, one-or-two, ',', X, 4N);
```

```
one-or-two = [1N, 2N];
two-or-four = [2N, 4N];
sep = ['/','-','.'];

month = [longenglish, shortenglish];

{intermediate months omitted in the next two pattern sets}
longenglish = ['January', 'February', ..., 'December'];
shortenglish = ['Jan', 'Feb', ..., 'Dec'];
```

Note that, from a syntactic standpoint, this pattern specification handles most of the forms of giving the date in the English language. Among the strings accepted by date are '2/2/1972' and '27.08.80', corresponding to form1, '4 July 1778' and '22 Nov 63', corresponding to form2, and 'June 6, 1944', corresponding to form3.

Two more observations may be made about this scheme:

(1) the availability of the built-in patterns and the ability to include string literals eliminates the need for a separate lexical analysis tool; primitive text units, i.e., tokens, can be placed within the patterns and pattern sets;

(2) the pattern declaration mechanism permits one to specify an arbitrary context free language, since patterns may contain arbitrarily many patterns and pattern sets with a completely recursive capability;

PLAIN contains two pattern matching operators: one for determining the *exact* match between a string and the pattern specification, and one for determining whether the pattern can be found anywhere in the string. Accordingly, two binary pattern matching operators, pattern match (?=) and pattern contains (?) were defined. The left-hand operand for each is a string; the right-hand operand is the name of a pattern. The pattern match operator '?=' returns true iff the pattern matches the entire string. The pattern contains operator '?' returns true iff the pattern matches a substring.

For example, if one uses the patterns form1 and form2 declared in conjunction with the date example above with the variables sa, sb, and sc as follows:

```
var sa, sb: string; sc: char[16];
```

with the following assignments

```
sa := '04/02/77';
sb := '27 Aug 72';
sc := 'Received 6-11-66';
```

then sa? = form1 is true, sc? = form1 is false, sb?form2 is true, sa? = form2 is false, and sc?form1 is true.

The binary operators **match** and **contains** are used with the **case** statement to allow branching based on pattern matching. These operations return a pattern name if the case expression, which must be of type string or array of char, is successfully found in the designated pattern set. The pattern name is then used as the case selector, as follows:

```
case input match month of {assume input declared of type string}
  when longenglish, shortenglish: english-date (display)
  when others: unknown-date (display)
end case
```

The remaining necessary capabilities are to be able to split a given string into its components and to combine two or more shorter strings into a longer string, based on patterns. The **split** and **combine** operations, respectively, provide these capabilities in PLAIN. The **split** operation apportions a string value to one or more variables, possibly discarding part of the string. The **combine** operation assembles two or more expressions into a single string value according to a specific pattern. The assembled string value is then assigned to a variable. A given string may be split or combined according to different patterns as necessary at any level of the pattern matching. Such a facility is particularly useful for processing of command languages.

With this set of pattern matching capabilities, it is possible to make effective use of the pattern facility in conjunction with the string handling features and to carry out the input/output and string processing that is essential to the effective construction of interactive programs. These string-handling and pattern matching features are described at greater length in [31].

5. Exception Handling

The ability to anticipate and to handle non-standard situations is essential to the construction of reliable systems. Thus, the specification for a system may provide not only for 'normal' conditions, such as proper operation of the hardware and meaningful user input, but also for abnormal conditions, such as hardware errors and arithmetic overflow,

describing the action to be taken if these conditions arise during system operation.

Accordingly, exception-handling mechanisms have been designed and implemented in many programming languages, including PL/I [14], MESA [16], CLU [12], and ADA [13]. Also, there have been proposals made for the inclusion of exception-handling mechanisms in languages and systems, and for the specification and implementation of exceptions [4, 7, 10, 15, 18].

The goals established for the exception-handling features of PLAIN are the following:

(1) Association of exceptions – it should be possible to associate exception handlers with specific exceptions and to bind this association at the statement level in the source program; it should also be possible to attach this association to a group of statements, such as a procedure body.

(2) Fielding of exceptions – it should be possible to pass an exception from the environment in which it was signalled to any previous level of invocation for handling.

(3) Orderliness – it should be possible to carry out normal shutdown procedures in the event of a fatal error, permitting, insofar as possible, the closure of open files, and the generation of messages.

(4) Grouping of exceptions – it should be possible to define a group of exceptions that are to be treated similarly under certain conditions.

(5) Programmer-defined vs. built-in exceptions – the exception-handling scheme should support both the handling of built-in exceptions and the definition, signalling, and handling of programmer-defined exceptions.

We designed a procedure-oriented approach to exception-handling for several reasons:

(1) the use of a call provides a constraint upon control flow, since control can return from the handler to its invocation point;

(2) the same handler can be invoked for several different exceptions or for several different instances of the same exception;

(3) the use of procedures serves to separate the exception-handling code from the remainder of the code;

(4) data coupling is made more visible through the parameter passing mechanism of procedure calls.

PLAIN provides built-in exceptions for commonly occurring exceptional program conditions, and permits the declaration of user-defined exceptions. Built-in exceptions are raised automatically by the runtime system, while user-defined exceptions must be explicitly raised. The **signal**

statement is used to signal a condition or event that needs special handling. The execution of a signal statement causes the program unit being executed to be immediately terminated at the point of the signal, with control returned to the invoker of the unit with the named exception as an active exception in the invoking context.

Program statements may optionally contain an exception part, which contains a list of exceptions and the names of associated exception-handling routines, called handlers. A **handler** is like a **procedure** in that it may be invoked from numerous places within a program and that standard parameter passing rules apply. Handlers are also like procedures in that there are no restrictions upon declarations or statement types; in other words, any type of computation may be performed within a handler.

The handler attempts to perform whatever actions are necessary to take care of the exception that caused it to be invoked and then returns to the point of invocation. There are four possible ways in which the computation may then proceed:

(1) the exception has been *cleared* and normal program execution may continue;

(2) the exception has not been handled completely and is then passed to the invoker of the routine in which the exception occurred, causing the termination of the routine;

(3) the exception has been *cleared* and the program segment (statement or compound statement) associated with the exception is *retried;*

(4) a different exception is returned to the location where the first exception occurred, which must be handled before handling of the first exception can be completed.

This mechanism permits exceptions to be passed up the activation chain and permits them to be handled at each level until they are cleared or until the absence of a programmer-defined handler causes the system-defined default handler to be invoked, thereby causing program termination.

The **clear** statement clears the exception that caused the invocation of the handler. The **retry** statement clears the active exception and then returns control to the beginning of the statement from which the handler was invoked, attempting to restore the environment which then existed. (Note that not all these effects, e.g., input/output and database updates, can be feasibly undone.) The **clear** and **retry** statements may only be used within a **handler**.

There are three built-in user-callable handlers that facilitate the use of this mechanism:

(1) abort, which signals the unclearable fail exception to the invoker of the currently executing routine;

(2) continue, which clears the active exception and results in continued execution of the currently executing routine;

(3) pass, which passes the active exception to the invoker of the currently executing routine.

Although this mechanism is more complex than some of those provided by other languages, it also provides some facilities that are not present in other exception-handling schemes, but that are important for interactive programs, including:

(1) exception handling is preemptive so that executions may be interrupted and stacked, making it possible to react to an exception while handling another;

(2) the pass handler makes it possible to pass exceptions through successive function/procedure invocation levels to a point at which the exception is meaningful in terms of the intended function; a low-level exception may or may not signify an error condition;

(3) the **retry** statement (see above) makes it very easy to program the common situation of asking the user to repeat input that does not conform to expected patterns.

These features may be illustrated by considering an example of user/program dialogue, such as asking the user to type in a valid bookid as defined above. In this example, the program reads a variable input according to the bookid pattern. An exception part is associated with the **read** statement to handle the various exceptional conditions that might arise. If the user transmits the break or the escape character, the handler break-message will be invoked. An exception can then arise while executing the **read** statement in break-message.[1]

```
var input: char[10];
  limit: integer;
  .
{limit is set to the number of tries we are willing to make}
  .
read[bookid]: input![ioerr: abort; patform: ask-again;
      break, escape: break-message];
```

[1] The exception parts shown in this example are intentionally thorough. In practice, the thoroughness of the exception parts would depend on the desired robustness of the program.

```
  {ask-again and break-message are user-defined handlers}
  .
  .
handler yes-or-no;
imports limit: modified;
begin
  if limit > 0 then
    write 'Please answer yes or no';
    limit := limit-1;
    retry {causes read in break-message to be repeated}
  end if;
  write 'The program is being terminated';
  signal fail;
end yes-or-no;
  .
  .
handler break-message;
var answer: string;
pattern yes-no = ['yes', 'no'];
begin
  write 'Do you wish to terminate the program? ...';
  read [yes-no]: answer![patform, time: yes-or-no];
  if answer = 'yes' then signal fail end if;
  retry {causes read in main program to be repeated}
end break-message;
  .
  .
handler ask-again;
begin
  write 'Invalid book number. Please try again.', \ n;
  retry {causes read in main program to be repeated}
end ask-again;
```

It can be seen from this example that a significant portion of the code in an interactive system must be devoted to management of the user/program dialogue, particularly if one wishes to create user-centered systems that are easy to learn and easy to use [29]. Because careful handling of user errors is critical in such an environment, the exception-handling mechanism is particularly important, and the exception handling features of PLAIN were designed with this requirement in mind.

6. Interactive Information Systems in PLAIN

The combination of database management, string handling, pattern matching, and exception handling within the framework of a language to support and encourage systematic programming is the most significant contribution made by PLAIN. These features work together most effectively in the construction of interactive information systems.

A program schema for the typical interactive information system characterized in the introduction is as follows:

```
program iisschema;
external {names of external objects used by program, such
    as databases and files}
var input: string;
    {other global declarations, including exceptions}
pattern cmdset = [com1, com2, com3, ..., comN, quit];
    com1 = (...);
    com2 = (...);
      .
      .
    comN = (...);
    quit = ('quit');
begin
  loop
    read input![ioerr: abort];
    {terminate on hardware I/0 error}
    case cmdset match input of
      when com1: action1 (...) {parameter list}
      when com2: action2 (...)
        .
        .
      when comN: actionN (...)
      when quit: exit
      when others: write 'illegal command' {pattern match failed}
    end case;
  repeat;
  write 'byebye'
end iisschema.
```

Each of the actions associated with the commands may perform additional decoding or analysis of the command, perhaps splitting the command string into substrings via the string functions or the split operation, and will then carry out the action implied by the user command.

Consider the example of a library information system using the pattern set command and the pattern chkout shown above. The procedure bookcheckout would include the following code:

```
procedure bookcheckout (input: string);
imports book, cardholder: readonly; checkout: modified;
  {book cardholder, and checkout defined external to bookcheckout
    as relations in library data base}
var booknum: char [10]; copyno: 1..100; datedue: char [4];
  oldcount, person: integer;
    .
  {handlers bad-book, bad-card, bad-copy, and dberr not shown}
    .
begin
  (#, #, booknum, #, person) := split [chkout]: input;
  {check validity of ISBN number and cardholder}
  assert book [booknum] in book ![assertion: bad-book];
  assert cardholder [person] in cardholder ![assertion: bad-card];
  write 'Copy number:';
  read copyno ![patform, range: bad-copy];
  write copyno;
  {compute due date and save in variable datedue}
    .
  {update set of checkouts}
  checkout :+ [⟨booknum, person, copyno, datedue⟩]![fail, duplicate:
    dberr];
end bookcheckout;
```

This brief example shows how these features combine to incorporate the facilities for interactive systems with such important features as assertion checking for semantic integrity of databases and powerful control structures. These features are easily used in a similar fashion for other similar kinds of examples and greatly simplify the problems of writing this class of programs.

Because of space limitations, we have omitted discussion of the PLAIN **module** facility, which provides facilities for data abstraction. The **module** facility is extremely useful in PLAIN, since it permits type extension of data-base types as well as other types. It is similar in most other respects to data abstraction facilities found in other modern languages, e.g., CLU.

7. Conclusion

The design of PLAIN combines modern programming language design concepts for creating well-structured programs with an integrated set of innovative features to support the implementation of interactive information systems.

Among the most significant aspects of these innovative features are:

(1) the associative addressing capability of relations, making it possible to access and modify individual data base items, to use data base items routinely throughout the program text, and to perform conversion between data base types and the underlying types of their attributes;

(2) the pattern and pattern set specification facility, making it possible to specify a context-free grammar, using the pattern-matching features to carry out the lexical and syntactic aspects of the text processing;

(3) the procedure-oriented exception-handling scheme, which makes it practical for the programmer to anticipate user errors and to build robust programs that handle these errors.

These features are largely orthogonal and do not interfere with one another in using or implementing the language, even though they are typically used together in practice.

Experience with PLAIN and with other modern programming languages indicates, subjectively at least, that it is much easier to implement interactive information systems with PLAIN than with any of the languages previously used for such applications or any of the other modern languages designed to support systematic programming. Work is continuing to use PLAIN to implement various application systems and software tools, as well as to develop implementations of PLAIN for a variety of execution environments.

References

[1] E. Allman, M.R. Stonebraker and G.D. Held, Embedding a relational data sublanguage in a general purpose programming language, Proc. Conf. on Data: Abstraction, Definition, and Structure, ACM SIGPLAN Notices 11 (Special Issue) (1976) 25–35.

[2] T. Amble, K. Bratsbergsengen and O. Risnes, ASTRAL: a structured and unified approach to data base design and manipulation, in: G. Bracchi and G.M. Nijssen (Eds.), Data Base Architecture (North-Holland, Amsterdam, 1979) pp. 257–274.

[3] A.L. Ambler et al., GYPSY: a language for specification and implementation of verifiable programs, Proc. ACM Conf. on Language Design for Reliable Software, ACM SIGPLAN Notices 12(3) (March 1977) 1–10.

[4] D.M. Berry, R.A. Kemmerer, A. von Staa and S. Yemini, Toward modular verifiable exception handling, Comput. Languages 5 (2) (1980).

[5] E.F. Codd, A relational model of data for shared data banks, Comm. ACM 13 (6) (June 1970) 377–387.

[6] E.F. Codd, Further normalization of the data base relational model, in: R. Rustin (Ed.), Data Base Systems, Courant Computer Science Series, Vol. 6 (Prentice-Hall, Englewood Cliffs, 1972) pp. 35–63.

[7] J.B. Goodenough, Exception handling: Issues and a proposed notation, Comm. ACM 18 (12) (December 1975) 683–696.

[8] J. Ichbiah et al., Reference manual for the ADA programming language, Advanced Research Projects Agency, U.S. Department of Defense (July 1980).

[9] B.W. Lampson et al., Report on the programming language EUCLID, ACM SIGPLAN Notices 12 (2) (February 1977) 1–79.

[10] R. Levin, Program structures for exceptional condition handling, Ph.D. Dissertation, Computer Science Department, Carnegie Institute of Technology, Pittsburgh, PA (1977).

[11] B. Liskov et al., CLU reference manual, Lecture Notes in Computer Science, Vol. 114, Springer, Berlin 1981.

[12] B. Liskov and A. Snyder, Exception handling in CLU, IEEE Trans. Software Engrg. SE-5 (6) (November 1979) 546–558.

[13] D.C. Luckham and W. Polak, ADA exception handling: an axiomatic approach, ACM Trans. Programming Languages and Systems 2 (2) (April 1980) 225–233.

[14] M.D. McLaren, Exception handling in PL/I, Proc. ACM Conf. on Language Design for Reliable Software, ACM SIGPLAN Notices 12 (3) (March 1977) 101–104.

[15] P.M. Melliar-Smith and B. Randell, Software reliability: the role of programmed exception-handling, Proc. ACM Conf. on Language Design for Reliable Software, ACM SIGPLAN Notices 12 (3) (March 1977) 95–100.

[16] J.G. Mitchell, W. Maybury and R. Sweet, MESA language manual, version 5.0, XEROX Palo Alto Research Center, Palo Alto, CA (1979).

[17] C.J. Prenner and L.A. Rowe, Programming languages for relational database management, Proc. AFIPS 1978 NCC, Vol. 47, pp. 849–855.

[18] B. Randell, System structure for software fault tolerance, IEEE Trans. on Software Engrg. SE-1, (2) (June 1975) 220–232.

[19] R.P. van de Riet, A.I. Wasserman, M.L. Kersten and W. de Jonge, High level programming features for improving the efficiency of a relational database system, ACM Trans. on Database Systems 6 (3) (1981), in press.

[20] L.A. Rowe and K. Shoens, Data abstraction, views, and updates in RIGEL, Proc. of ACM 1979 SIGMOD Conference, Boston, MA, pp. 71–81.

[21] J.W. Schmidt, Some high level constructs for data of type relation, ACM Trans. on Database Systems 2 (3) (September 1977) 247–261.

[22] J.E. Shopiro, THESEUS – a programming language for relational databases, ACM Trans. on Database Systems 4 (4) (December 1979) 493–517.

[23] A.I. Wasserman, Online programming systems and languages: a history and appraisal, Techn. Rep. No. 6, Laboratory of Medical Information Science, University of California, San Francisco CA (1974).

[24] A.I. Wasserman, Embedding database management operations in programming languages, Conference Digest – IEEE COMPCON Spring 1976, pp. 79–82.

[25] A.I. Wasserman, USE: a methodology for the design and development of interactive information systems, in: H.-J. Schneider (Ed.), Formal Models and Practical Tools for Information Systems Design (North-Holland, Amsterdam, 1979) pp. 31–50.

[26] A.I. Wasserman, The data management facilities of PLAIN, Proc. ACM 1979 SIGMOD Conference, Boston, MA, pp. 60–70.

[27] A.I. Wasserman, Software tools and the user software engineering project, in: W.E. Riddle and R.E. Fairley (Eds.), Software Development Tools (Springer Verlag, Heidelberg, 1980) pp. 93–113.

[28] A.I. Wasserman, The design of PLAIN – support for systematic programming, Proc. AFIPS 1980 NCC, Vol. 49, pp. 731–740.

[29] A.I. Wasserman, User software engineering and the design of interactive systems, Proc. 5th International Conference on Software Engineering, San Diego, 1981, pp. 387–393.

[30] A.I. Wasserman et al., Revised report on the programming language PLAIN, ACM SIGPLAN Notices 16 (5) (May 1981) 59–80.

[31] A.I. Wasserman and T. Booster, String handling and pattern matching in PLAIN, Techn. Rep. No. 50, Laboratory of Medical Information Science, University of California, San Francisco, CA (1981).

[32] A.I. Wasserman and M. Dippé, Design and evaluation of a procedure-oriented exception-handling mechanism, in preparation.

[33] N. Wirth, The programming language PASCAL, Acta Inform. 1 (1) (1971) 35–63.

[34] W.A. Wulf (ed.), An informal description of ALPHARD (preliminary), Techn. Rep. CMU-CS-78-105, Department of Computer Science, Carnegie-Mellon University (1978).

Algorithmic Languages, de Bakker/van Vliet (eds.)

PORTAL – A PASCAL-Based Real-Time Programming Language

Rudolf Schild

Central Research Laboratory, LGZ Landis & Gyr Zug AG, CH-6301 Zug, Switzerland

The high level programming language PORTAL was developed to alleviate the problems experienced in programming complex real-time process control systems. It is based on PASCAL and includes facilities for breaking a task up into modules (information hiding), for describing and synchronising parallel processes, and for handling peripheral devices and interrupts. It has been used in actual systems with excellent results.

1. Introduction

1.1. *Motivation*

The programming language PORTAL (for Process Oriented Real-Time Algorithmic Language) was developed for the efficient production of reliable real-time software.

In process control software parallel processes as well as real-time events play an eminent part. This, of course, makes the programming of such systems especially difficult. We know by now that the design of sequential programs is in itself a difficult task. The difficulties are compounded when several activities go on concurrently, but each at its own speed, and when asynchronous external events must be taken into account.

In view of the possibly severe costs of a failure in a real-time system, it is extremely important to detect errors as early in the development process as possible. If the manner in which the system is designed and the tools used for this task can prevent a number of errors from being committed in the first place then quite a lot has been gained.

If we are furthermore able to reduce the complexity of what we are dealing with at any given moment, so that we can comprehend the (sub)task and stay in control, then there is hope for building complex yet reliable systems.

Thus, to attack the problem of producing efficient and reliable systems we found these two means essential: structuring – to keep within manageable limits – and redundancy – to reduce the number of errors.

1.2. *Achieving the goal*

Originally we did not set out to produce a new language. The project started in 1974 with a study that was to find tools for proving real-time software correct. We did find that there were quite a number of efforts going in that direction, but we also found that the general consensus was: it is more difficult than we expected, and beyond the current state of the art.

Our next thrust – in 1975 – was in the direction of finding an existing programming language which would allow us to build our systems in such a way that we could be reasonably certain the job had been done well, and which would be amenable to program proofs if there ever was a way of automatically proving programs.

Again we did not find what we were looking for. To be sure there were a number of candidates, but some existed only on paper, while for others we could not obtain a compiler that would fill our needs.

Thus our final step was to design our own language. Throughout our effort we remained in contact with Niklaus Wirth and his collaborators at the Swiss Federal Institute of Technology (ETH Zurich), who were then developing Modula.

1.3. *Language design*

As a basis we chose PASCAL, augmented with constructs for parallel processing. Since the study of programming languages in the early seventies had pointed out the desirability or undesirability of a number of language constructs, we decided to make use of that knowledge, rather than try to invent new features.

2. The PORTAL Language

2.1. *Structuring*

In order to manage the complexity of a system, there are two ways of structuring it:

(a) The entire system is divided into several processes, each of which runs at its own speed, perhaps driven by some sort of external signal. Each of these processes can then be viewed separately, with occasional interaction with some of the others, and totally ignorant of the existence of the remainder of them.

(b) There is also a static division of the system into modules. Modules serve as information hiding devices; they can be used to implement certain structures or even entire subsystems. The internal implementation of a module need not be known to the outside, i.e. to the user of the module; access to it is solely through the interface.

2.2. *Synchronisation*

In order to make up a coherent system, the parallel processes must somehow be able to synchronise with one another.

We chose the well-known monitor and signal concept [1]. The monitor guarentees exclusive access to the routines and therefore to the data within it. A process that is active within a monitor may find that it is unable to proceed because some condition is not present or some event has not happened. In this case it frees the monitor by waiting for a signal. This signal must subsequently be given by another process, which must be active within the monitor to do so. It creates the condition desired by the waiting process and sends the signal.

It seemed important to us to assure that this condition is in fact still true when the first (the waiting) process receives control, since any kind of proof or even plausibility argument would have to be able to rely on that. For this reason we decided to switch processes immediately after the send statement. This means that the sending process is suspended until the waiting – and now reawakened – process frees the monitor again, either by leaving it for good, or by going into another wait state.

2.3. *Synchronisation with the clock*

A rather simple device was introduced to permit access to a clock. A process may wait for a signal and at the same time specify a maximum delay (in some predefined units such as ticks). This is often useful in connection with external interrupts (Section 2.4), which might be lost and thus keep a process waiting forever.

Normally, in accordance with Hoare's ideas, processes are unidentified,

which means that a process cannot tell who woke it up. But when the wait with delay is used, it may well be important to know that the delay has elapsed and the expected signal did not come, which might e.g. indicate some equipment malfunction. Therefore there is another parameter accessible if the wait is used with a delay, which indicates whether the clock or some other process sent the signal.

The abilily to recover from lost interrupts was in fact the principal reason for introducing the wait with delay. But it can also be used to wait for a dummy signal, which will never be sent by any other process and which therefore causes the waiting process to be awakened by the clock, after the specified delay has elapsed.

2.4. *Interrupts*

Real-time systems generally comprise a number of peripheral hardware devices that usually communicate via interrupt with the central processor. It seemed important to us that the system designer be able to handle interrupts without having to take recourse to assembler language programming.

The method we chose was to represent interrupts as signals given by a hidden process. Once a signal is defined as belonging to an interrupt, it is then used in exactly the same way as any other signal.

2.5. *Safety*

Since our goal included preventing errors from being committed (if possible) the criteria for inclusion of a certain construct in the language were:

- it must aid the designer in constructing reliable software,
- it should not be inherently dangerous to use,
- it must be implementable in a clear and fairly straightforward way. .

According to these criteria we did not include pointers or record variants (dangerous) nor exception handling (difficult).

It was felt, however, that thereby we restricted ourselves too much, and we introduced different, safer versions of pointers and record variants. (We do not yet have an answer to the problem of exceptions.)

Pointers for linked lists, say, can be had by our index type. This is always tied to a specific array type, can only be used with an existing variable, and thus avoids the problem of dangling references at least partially.

Pointers to an element of a data structure, say from inside a module, can be realised with the resource function, which in effect returns a pointer. The syntax ensures that this result pointer is only used while access to the structure, and in particular to that element, is guaranteed.

The case type represents a restricted form of the record variants, similar to ALGOL 68's union. The syntax was chosen such that the specific variant being processed must be indicated at compile time, either as a constant if it is known, or in a case statement if it must be selected. Either way the language guarantees that in fact the fields for the actual variant are being accessed and none other.

3. Implementation

3.1. *Synchronisation*

The methods used to implement monitors and signals are quite straightforward. At run time the PORTAL nucleus manages all the processes and their synchronisation. The state of each process is recorded in a process descriptor. Calls of monitor routines – i.e. routines which can be viewed as indivisible actions – are not executed directly but via the nucleus, which keeps track of the availability of each monitor and maintains the entry queues.

The signals are implemented as queues also, with the execution of a wait statement causing the process descriptor to be entered into that queue. The execution of a send statement puts the executing process's descriptor in a stack and removes the first item from the signal queue.

A process leaving a monitor will cause the send stack to be popped. If it is empty, i.e. if there are no more processes suspended because of sending a signal, then the first process from the entry queue (if there is one) enters the monitor.

Processes waiting with a maximum delay are additionally linked in a time-out list, since they may be removed from inside their respective signal queues.

Interrupts are caught by the nucleus, which then removes the first descriptor from the corresponding signal queue, just as if a signal had been sent by another process. 'Normal' send's, i.e. those executed by processes, may be performed on an empty queue, in which case no other process is

started. The sent signal is thus lost without any other effect. In case of an interrupt, however, there must be exactly one process waiting for it, otherwise a run-time error occurs. This seems to be a reasonable interpretation of interrupts as hidden processes.

3.2. *Priorities*

Each process is given, at compile time, a fixed priority under which it runs. The assignment of priorities is up to the programmer; if none is given the compiler uses a default value of zero (lowest).

In addition, monitors are also assigned priorities, also fixed at compile time (default 3). While a process is active within a monitor, it runs under the monitor's priority. To avoid the possibility of resulting confusion, processes may only enter monitors of at least the same priority as their own current priority. Thus it is possible to cell a routine in another monitor from within one, but only if this would not lower the process's priority.

Monitors whose routines deal with hardware interrupts must be assigned the priority of the hardware device the handle in order to run correctly.

3.3. *Checks*

3.3.1. *Run-time checks*

The usual run-time checks for overflow, assignment to subrange variables, access to array elements, selection of case statements, are of course included.

3.3.2. *Computation of the stack lengths*

Since all routines are reentrant, i.e. they can be executed by several processes at once, their local data are stored on a stack. Each process has its own stack whose size increases and decreases during the system's operation. In most process control applications it is considered quite unacceptable to have the system signal a memory overflow, and perhaps stop, simply because one of the process stacks has overflown. To avoid this, the PORTAL compiler contains a pass which computes the maximum stack length for each process, taking into account all routine calls. Each process is then assigned a portion of memory for its stack, and it can be guaranteed that this will suffice for any possible control flow, yet it will not be more than can actually be required.

If routines call each other recursively, this is no longer possible,

however, and the compiler issues a warning message. The programmer in this case has the option of introducing a special test with the recursive calls, allowing the program to handle imminent stack overflow itself.

The stack length computation has been found very useful, in particular by users who had had access to an earlier version with a fixed allocation of stack storage.

4. Examples

Two examples should serve to illustrate the experience made with the language in different areas. Details will not be considered.

4.1. *A process control example*

The system consists of two identical PDP-11/04's that work together as a master/standby system. The two CPU's are connected by a watchdog unit for changing the standby machine to master, and a single data line for transmission of certain operator input from one machine to the other, for updating purposes. Attached to each machine are a teletype, a magtape unit, and up to six communication interfaces connected to telephone lines for dialling, and for sending and receiving information from outlying stations.

Roughly, the specifications for the system are as follows. Every day at a specified time the master unit calls the outlying stations according to a list and requests information from them. This information is then stored on magnetic tape for later off-line processing. The standby unit listens in continuously and stores the data it receives on its own magnetic tape. Thus, if all is well, master and standby always have the same data on their respective tapes.

The time at which the calling sequence is to start, the list of stations, as well as a list of messages which the system may print out, can all be changed on-line by the operator. Furthermore, the operator is able to call up stations individually as well as retrieve selected information from either magtape.

The system was programmed entirely in PORTAL, with the exception of the nucleus (about 1 K bytes). In particular, all drivers were programmed without using assembly language.

Our experience with this system was extremely encouraging. The deadline for the acceptance test was easily met; and neither the acceptance test nor the subsequent operations (the system has been in continuous use since March 1979) have turned up any software errors.

4.2. *A simulation example*

In simulating a polyvalent heating system, full use could be made of the possibilities for parallelism as well as of the modular approach.

Physically, such a system consists of a number of elements such as heat stores, heat consumers and producers, etc., which are interconnected in a pipe network. For each element there exists a corresponding PORTAL module containing a process representing the dynamics of the element. A complete system is configured during a dialogue with the computer, determining the elements and the actual pipe-layout. This makes the program very flexible and easy to use. Each type of element constitutes a different simulation problem, but by this separation they are easy to handle. Once the framework of the program had been completed, different people were able to implement elements without any deep knowledge of the intricacies of the rest of the program; only the interface definitions had to be observed.

Furthermore, if desired, true concurrency can be achieved by running a number of modules on a second processor. Thus all the calculations for solving the differential equations for the elements are done on one machine, while the remainder of the program (the simulation control, the plotting, etc.) run on another processor. To achieve this separation, only a small number of quite localised changes have to be made, essentially stretching the interface between two modules across machine boundaries.

5. Programming and Debugging Support

Two programs to support the PORTAL system have been written in PORTAL: an editor and a post-mortem-dump analyser.

5.1. *The editor*

The editor provides the usual functions for editing files, with the user moving a cursor on the display screen to indicate the place where a change

is to be made. But in addition to the normal editing mode, this editor also checks the PORTAL program being edited for syntactical correctness (line by line).

If so desired, keywords may be entered by pressing just one functional key. When the program is displayed or listed, all keywords are (optionally) converted to lower case letters for better readability of the program.

For easier structuring, the program lines are automatically indented. Structures (statements, routines, modules) can be properly terminated by just pressing the special end-key.

5.2. *The post-mortem analyser*

The post-mortem-dump analyser lets the user request information about the values of variables, the status of processes and the calling sequence of routines, after a run-time error has occurred. After the error, the entire memory partition used by the program is dumped onto a file, e.g. on a floppy disk. This file, together with files generated during compilation, are then used to produce the required formation interactively.

It is important to note that the code of the program itself is in no way changed by the fact that a PMD analyser is being used. All the information the analyser needs to find names and types of variables, line-numbers of statements, etc. is contained in the files produced during compilation; they provide the connection between the source program and the machine representation.

Depending on the actual system configuration it is also possible to set triggers, referring to line numbers. Whenever control passes such a trigger, the system is stopped and may be analysed on-line, then execution can be resumed.

6. Conclusion

Great care has been taken in the design of PORTAL to produce a language that will provide the necessary tools and constructs for the efficient development of process control software with special emphasis on the reliability of the finished product. A great deal of thought and debate has gone into it, especially where compromises had to be made between 'pure' and 'practical', which sometimes seem to lie at opposite ends of the spectrum.

Whether it is for the production of a real-time system with physically concurrent processes, or for writing an essentially sequential program, experience has shown us that we are on the right track and that our new development tool is a useful one.

Acknowledgement

I would like to thank my friends and colleagues at Landis & Gyr for their work on PORTAL, which would not exist without them.

I would also like to thank Klaus Wirth for numerous fruitful discussions and for introducing me to compilers in the first place.

And finally I would like to thank the referees for their helpful suggestions on the first draft of this paper.

References

[1] C.A.R. Hoare, Monitors: An operating systems structuring concept. Comm. ACM 17 (10) (1974) 549–557.

Algorithmic Languages, de Bakker/van Vliet (eds.)

Naming by Colours: A Graph-Theoretic Approach to Distributed Structure

J.D. Roberts
University of Reading, Reading, U.K.

The use of relative or 'local' naming which is already significant in many programming languages is developed further by the concept of a directed graph (called a 'name-graph') with labelled nodes and coloured arcs. Such name-graphs enable remote objects to be named by 'colour' rather than by globally valid labels, and their use is illustrated by systems incorporating both active and passive components. Various ways of generating name-graphs are explored, as finally also is their application to specifying type and visibility.

1. Introduction

In the 1960's Dijkstra [2] suggested the possibility "that a confrontation with the intricacies of Multiprogramming [could] give us a clearer understanding of what Uniprogramming is all about". Another question which we might ask in the 1980's is whether a confrontation with Distributed Computing might give us a clearer understanding of some of the fundamental problems of data structure and type.

In the exploration which follows answers are sought to the following questions.

> Should a programming language allow communication structure to be described independently of the other details of the program?
>
> Should it be possible for a process to reference neighbours other than through global identifiers or formal parameters?
>
> Can regular but non-rectangular communication structures be described concisely by generally applicable methods?
>
> How can synonymous references be prohibited?

This study has also been motivated by a longer term wish to understand various fundamental problems of how sharing and recursivity in data structures should be handled and of how far references can be removed by abstraction.

Eventually perhaps the study will be presented starting with passive data structures and working towards distributed activities; but here the questions are taken in the opposite order, for the simple reason that this is how the exploration has so far taken place.

2. Naming by Colours

The basic principle developed here is that the naming of objects in some localized context needs only to distinguish between members of that set of objects to which reference needs to be made. As such, the principle is already well established in programming languages. In the context of an ALGOL 60 block, a variable is specified uniquely by its identifier, whereas in a global context it would be necessary to specify somehow which *instance* of the block was intended. Although it is not to be found in ALGOL 60, such extra-contextual reference to local names *is* an important feature of more recent languages exemplified by the **'inspect'** feature of Simula or the **'with'** construction of PASCAL. What does *not* seem to be found currently is the use of local or relative naming in a situation where a fairly large number of (active or passive) objects coexist at the same scope level, but where in the context of each object, reference needs to be made to only a very small number of other objects; yet in distributed computing this seems to be a highly realistic requirement. The reasons for this lie in:

(i) the economic desirability of localizing communication,

(ii) the fact that many problems do admit solutions having such structure, and

(iii) the simplicity and uniformity of programs which provide such solutions.

In graph-theoretic terminology, the information required to specify such a relative or local naming scheme corresponds to a *directed graph with labelled nodes and coloured arcs.* Such a graph we shall call a 'name-graph'. The nodes correspond to the objects, and their labels correspond to global identifiers. An arc is directed from an object A to an object B if and only if reference has to be made to B in the context of A, and the *colour* of

the arc corresponds to the local name by which reference is made. The labels (so-called because they have to be all different) attached to the nodes are all strings of symbols over some alphabet, and the developments in the later sections will actually use this structure. This is perhaps the only not entirely standard graph-theoretic concept. The colours (so-called because the same colour may be used on many arcs) are treated as symbols with no internal structure. As it is *sometimes* quite natural for an object to be called by the same name by more than one process, there is *no* requirement for name-graphs to be *properly in-coloured* (i.e. have at most one arc of any colour directed *into* any node); but they will normally be *properly out-coloured.* To violate the latter restriction would lead to non-deterministic references to variables which we shall, provisionally at least, avoid. Most ways of expressing communication between two activities seem to require each to be able to reference the other. For this reason name-graphs related to systems of active objects are typically *symmetric* digraphs but this is not an essential property.

The use of relative names of this kind is analogous to the use of pronouns in natural language (see Fig. 1), or the use of logical numbers of devices in computing systems.

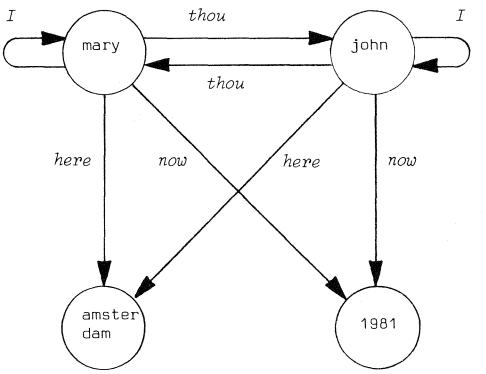


Fig. 1. Hypothetical application of relative names in natural language.

3. Two Examples

The applicability of the 'name-graph' concept is not confined to any particular communication or synchronization mechanism. To illustrate this

fact two examples of asynchronous processing have been chosen which use contrasting primitive mechanisms. Both have been described previously using global names and here the original mechanism of each is represented but in terms of relative names (i.e. the colours of the arcs in a name-graph). They are:

(1) matrix × vector multiplication pipeline [6], using the handshaking communication primitives '?' (input) and '!' (output), and

(2) the dining philosophers [4] using *'p' 'v'* operations on semaphores and a subroutine calling mechanism.

3.1. *Matrix × vector multiplication*

In this example a stream of 3-vectors is multiplied by a constant matrix using a system of 21 processors which are arranged on a rectangular array

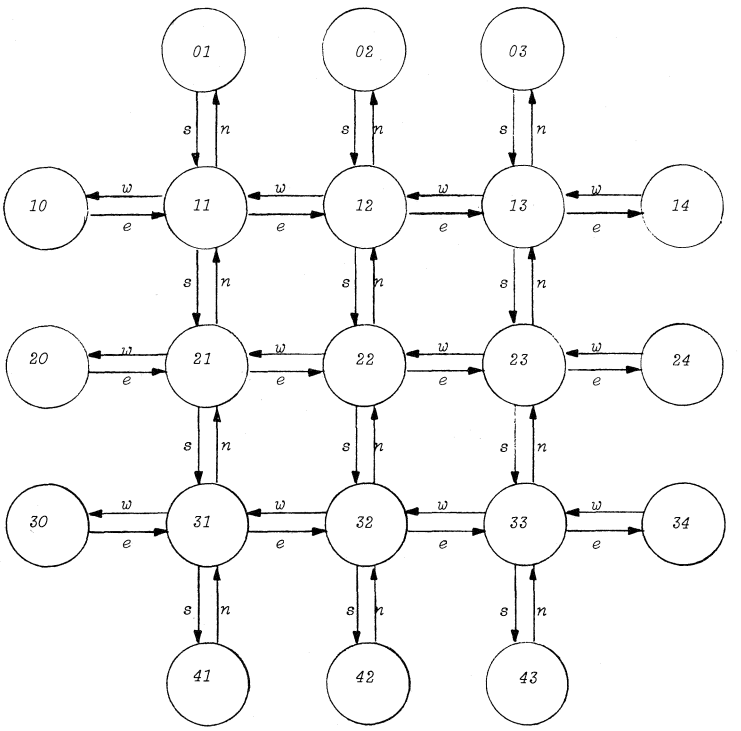


Fig. 2. Name-graphs applied to distributed computing. (a) Matrix × vector pipeline.

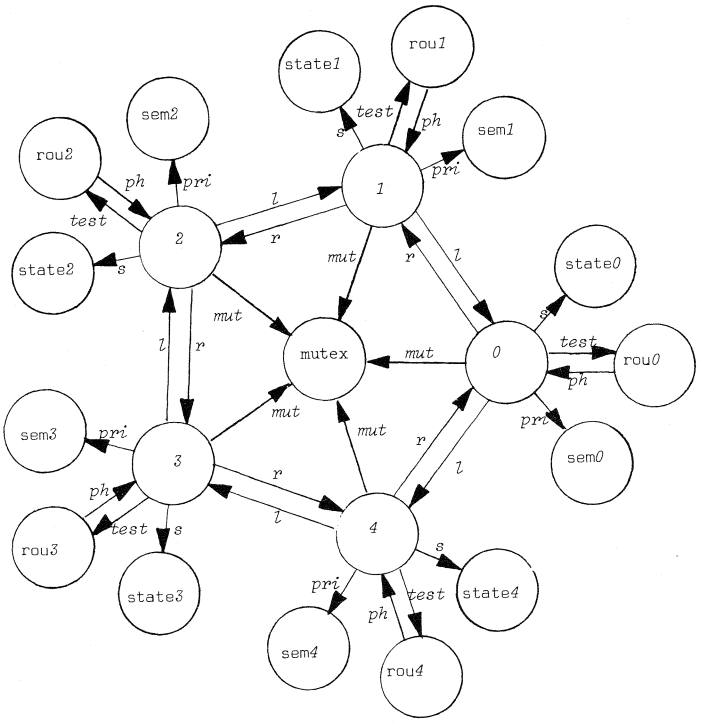


Fig. 2 (continued). (b) The dining philosophers.

and the programs which they execute are divided into 5 classes, instances of which are located respectively at the north, south, east and west borders and centre square of the system.

In the original description, the definition of each process class referred to its neighbouring processes by use of globally recognizable pairs of subscripts; but in the context of the name-graph shown in Fig. 2a those processes with which direct communication takes place can be identified more simply by their *relative* positions which are denoted by the arc colours *n, s, e, w*. Each node is labelled with a string of length 2 of symbols taken from $0, \ldots, 4$, which serves to determine the class of process attached.

The program below shows how this system could be represented in a form which relates to this name-graph but which follows Hoare's original version in other details.

```
"matrix × vector multiplication":
begin
    "Fig. 2a";
      {This brings into the scope of this block the colours and labels of
      Fig. 2a and the connections between them and also provides the
      environment for defining classes of activity at each of the labelled
      nodes}
    formal j, k: 1, ..., 3;
      {This restricts the classes of symbol for which j, k may stand when
      used as name parameters delimited by the meta-symbols ( , )}
    at '0(k)'    : "north"; at '4(k)' : "south(k)";
    at '(j)4'    : "east"; at 'j(0)' : "west";
    at '(j, k)'  : "centre";
    "north"      : *[true→s!0]
    "south(k)" : "process consuming output(k)"
    "east"       : *[x: real; w?x→skip]
    "west(j)"  : "process supplying input(j)";
    "centre(j, k)" : [a: real; "initialize a depending on (j, k)";
                        {which can be performed prior to run-time
                      possibly even when the hardware is being built}
                          *[x: real; w?x→e!x;
                          sum: real; n?sum; s!(a*x+sum)
                    ]]
end
```

3.2. *The dining philosophers*

In this classical example the sustenance of each philosopher depends on the non-eating state of his left and right neighbours. By using the name-graph shown in Fig. 2b, these neighbours can be denoted by *l, r* in identical routines for each philosopher, and the original identities 0, 1, 2, 3, 4 cease to serve any purpose. The solution give here, closely follows that recommended in Dijkstra's original discussion [4]. This made use of a routine named *test(i)* for testing and if appropriate stimulating a particular philosopher *i* into eating. In the distributed representation, each potential activation of this routine is realized as a separate activity connected directly to the philosopher (denoted relatively by the colour *ph*) being tested. A specified test activation is invoked by an occurrence in the program text of

a statement of the form '*call* (⟨routine activation name⟩)'. It may be conceptually useful to regard this as a coroutine call; but in practice the simplest of subroutine calling mechanisms will suffice, since the use of the global mutual exclusion semaphore in this example ensures that no already active routine will be re-entered.

```
"5 dining philosophers":
begin
    "Fig. 2b";
    formal i: 0, ..., 4;
    at '(i)': "philosopher";
    at 'rou(i)': "test routine";
    "philosopher": *[true
                    →p(mut); s := hungry; call(test);
                      v(mut); p(pri);
                      "eat";
                      p(mut); s := thinking;
                        call(l . test); call(r . test);
                      v(mut);
                      "think"
                    ];
    "test routine": [ph . s = hungry and eating ∉ {ph . l . s, ph . r . s}
                    →ph . s := eating; v(ph . pri)
                    ▯ ... {else-condition} ...
                    →skip
                    ]
end
```

4. Description of name-graphs

Although each of the name-graphs illustrated in Fig. 2 is intended to be an essential part of a computer program, we have not yet proposed any notation which would be acceptable to a computer for describing them; nor is there any fundamentally urgent reason to do so. I suggest in principle, that Figs. 2a, b be considered provisionally as examples of a perfectly acceptable 'publication language', and that any sequential text containing equivalent information be regarded as analogous to what in the ALGOL 60

report was called a 'hardware representation'. Nevertheless, the systematic and repetitive structure of the examples does suggest that it would be of value (and indeed necessary in larger scale examples) to identify principles for describing large regular structures concisely. This is in fact one of the distinctive features of Hoare's proposal [6] which is intended as a neutral description suitable for realizing either on distributed systems or by sequential operations on vectors; but this particular notation is restricted to rectangular arrays and we would require different notations to describe non-rectangular structures.

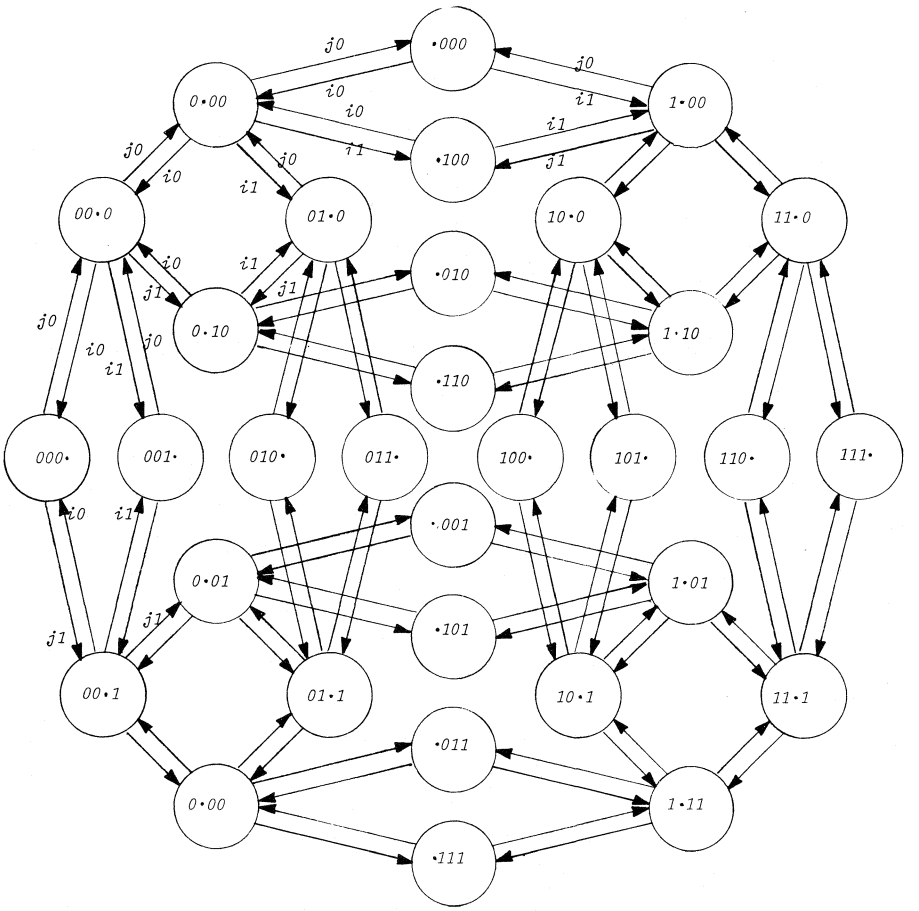


Fig. 3. Fast discrete fourier transform pipeline.

One example of a regular but non-rectangular structure has already been given in Fig. 2b. Another pattern is exhibited by the fast discrete fourier transform pipeline illustrated in Fig. 3. Further examples of communication structure are offered by the hexagonal mesh systems and the hierarchical configurations (which map onto Peano-like curves) of Kung [8]. Some applications may require the description of structures which combine a varied collection of symmetric substructures in this way. One example of this would be the use of asynchronous machines to implement finite element methods (as described by Loendorf [9]), which would require the specification of name-graphs corresponding in structure to the finite element decomposition; and these, as has been illustrated by Zienkiewicz [13], can be highly complex and irregular. These represent a very wide variety indeed and indicate the need for a fundamental approach which does not favour any particular type of structure.

Three basic methods will be explored for describing name-graphs:

(i) enumeration of nodes and arcs (with their labels and colours),

(ii) functional description of the maps (over the domain of nodes) defined by each colour,

(iii) generation from smaller name-graphs by means of the operations $\cup$ (union) and $\times$ (Cartesian product) supplemented by appropriate joining and contracting principles.

4.1. *Enumerative and functional descriptions*

Simple name graphs are readily describable by enumeration of the pairs of nodes connected by arcs of each colour e.g.

"Fig. 1":
name graph
nodes '1981', 'amsterdam', 'john', 'mary';
arcs *I* : 'john' → 'john', 'mary' → 'mary';
thou : 'john' → 'mary', 'mary' → 'john';
here : 'john', 'mary' → 'amsterdam';
now : 'john', 'mary' → '1981'
end

The description of a systematically constructed larger graph can

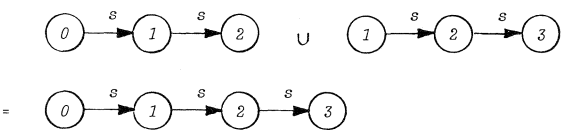
alternatively be achieved by defining the underlying functions which it describes. This requires some means of indexing whole families of nodes with formal parameters, as has already been used viz:

"Fig. 2a":
name graph
formal $i: 0,\ldots,3$; $j: 1,\ldots,3$; $k: 1,\ldots,4$; $l: 0,\ldots,4$;
nodes '0(j)', '4(j)', '(j,l)';
arcs n: '(k,j)' → '($k-1,j$)';
s: '(i,j)' → '($i+1,j$)';
e: '(j,i)' → '($j,i+1$)';
w: '(j,k)' → '($j,k-1$)'
end

4.2. *Composition from simpler graphs*

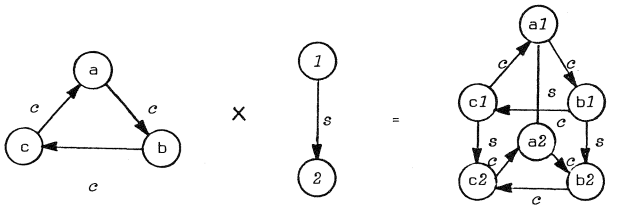
Three composition operations on name-graphs are proposed namely: *union* (denoted by ∪), *Cartesian product* (denoted by ×) and *directed coloured join* (denoted by $\overset{c}{\leftarrow}\!\!+$ where c is a colour); and these are supplemented by *colour contraction* operation (denoted by **mod**).

The union $\mathscr{A} \cup \mathscr{B}$ of two name-graphs $\mathscr{A}$ and $\mathscr{B}$ is simply the union of the nodes and arcs of $\mathscr{A}$ and $\mathscr{B}$ with similarly labelled vertices contracted to a single vertex and redundant repeated edges with same name removed. e.g.



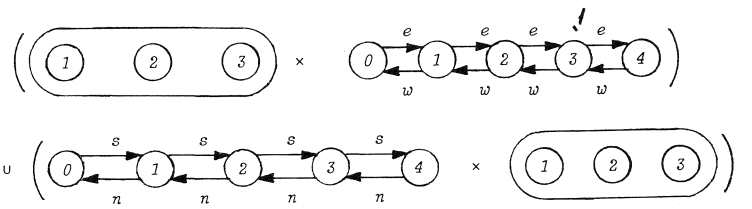
If none of the labels in $\mathscr{A}$ occur in $\mathscr{B}$, then the structure of the union thus defined is equivalent to the conventional union of two unlabelled graphs.

The product $\mathscr{A} \times \mathscr{B}$ is formed by constructing the conventional Cartesian product and labelling each vertex with a string formed by concatenating the strings used to label the corresponding vertices of the original graphs. All the edges retain their original colours. E.g.



The pair of operators ∪ and × satisfy the normal associative and distributive laws of the algebra of sets.

Using these composition operators, the name-graph of Fig. 2a can be expressed as

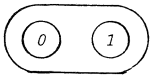


Here the factor graphs could quite reasonably be regarded as 'standard' graphs or alternatively they could be decomposed by further recursive formulae to even simpler forms.

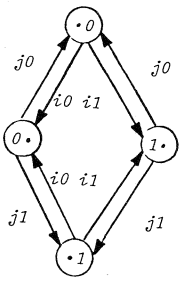
An example of a regular but non-rectangular structure is offered by a hypothetical pipeline for the fast discrete fourier transform. This was used by Dijkstra [3] as an example of how to 'build elephants out of mosquitoes' and further considered by the writer [11]. A name graph which leads to concise and uniform coding at the nodes is shown in Fig. 3 for the case of order 8. The nodes are labelled by strings of symbols taken from the alphabet $\{0, 1, \cdot\}$. This order-8 graph contains two order-4 graphs as subgraphs (on the left and right of the picture) in which the vertex labels are prefixed by 0 and 1 respectively. These are joined by four order-2 graphs which can be generated respectively by appending the strings 00 01 10 11 to all the node labels of the basic order-2 fourier transform graph (shown below). The order-4 graph is similarly decomposable and more generally the name-graph $\mathscr{F}_n$ for the transform pipeline of order 2^n can be generated by the recursive formula

$$\mathscr{F}_n = (\mathscr{E} \times \mathscr{F}_{n-1}) \cup (\mathscr{F}_1 \times \mathscr{E}^{n-1}) \quad (n > 1)$$

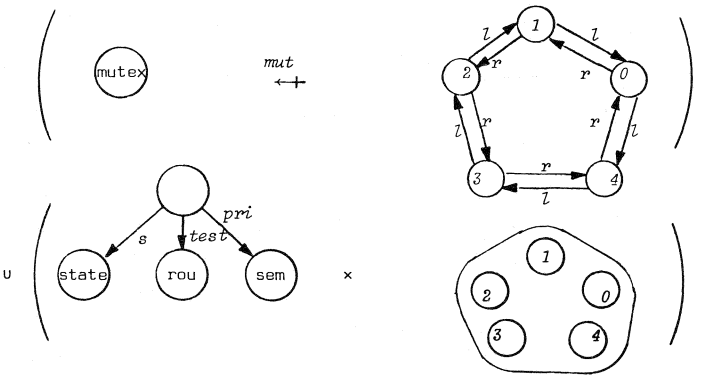
where

$\mathscr{E} \triangleq$ 

and

$\mathscr{F}_1 \triangleq$ 

To describe the kind of name-graph shown in Fig. 2b (for the dining philosophers) we introduce the directed coloured join $\mathscr{A} \overset{c}{\longleftarrow\!+} \mathscr{B}$ which is generated by augmenting the union $\mathscr{A} \cup \mathscr{B}$ with c-coloured arcs directed from every node of $\mathscr{B}$ to every node of $\mathscr{A}$. Fig. 2b can thus be expressed as follows.

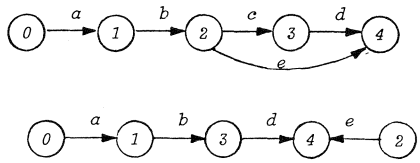


As with the operation $\cup$, if $\mathscr{A}$ and $\mathscr{B}$ share no labels in common, the structure of the graph $\mathscr{A} \overset{c}{\longleftarrow\!+} \mathscr{B}$ is that of the ordinary undirected join $\mathscr{A} + \mathscr{B}$. In the most general case the join so defined could contain loops and multiple edges; but we neither use nor (for the time being) formally prohibit such constructions.

The fact that we are working with *labelled* graphs actually enhances the versatility of the union and join operations. In fact the graph obtained by

removing all labels colours and edge directions from Fig. 2b has the kind of structure described by Akiyama and Harary [1] in terms of a *ternary* composition operation based on the ordinary *join* (denoted by '+') already mentioned and a more complex *corona* operation (denoted by '$\circ$').

Finally, we show how the union operation on labelled graphs plays an important role in building systems by connecting prefabricated modules; but for this purpose we also need a contraction operation. To construct the graph denoted by $\mathscr{A}$ **mod** c (where c is a colour) we proceed as follows. For each arc α in $\mathscr{A}$ of colour c we note the originating node α_0 and the destination node α_1 and redirect all arcs pointing to α_0 to point to α_1; then we remove the c-coloured arcs. E.g.

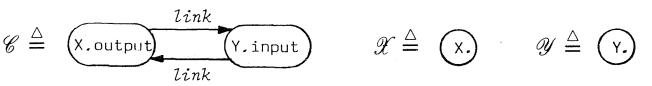


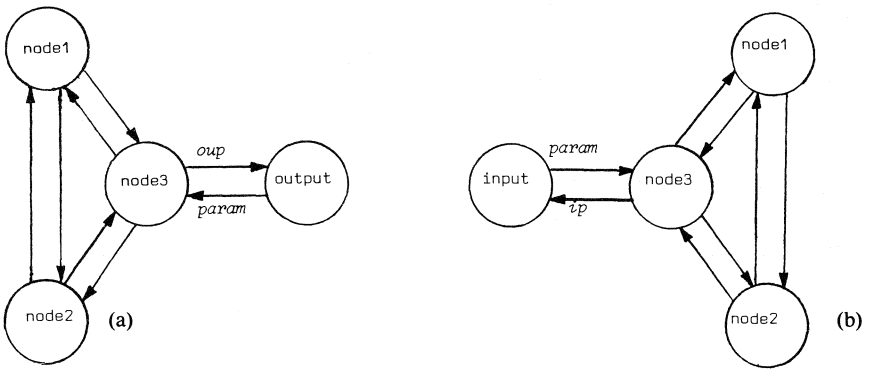
mod c

For the application to the linking of modules we refer to Fig. 4. The graphs $\mathscr{A}$, $\mathscr{B}$ shown in Figs. 4a and 4b represent classes of activity with unspecified input and output. The graph shown in Fig. 4c represents two communicating instances of $\mathscr{A}$ and $\mathscr{B}$ (respectively named X and Y) and can be generated by the formula

$$((\mathscr{X} \times \mathscr{A}) \cup (\mathscr{Y} \times \mathscr{B}) \cup \mathscr{C}) \textbf{ mod } \textit{link}) \textbf{ mod } \textit{param}$$

where





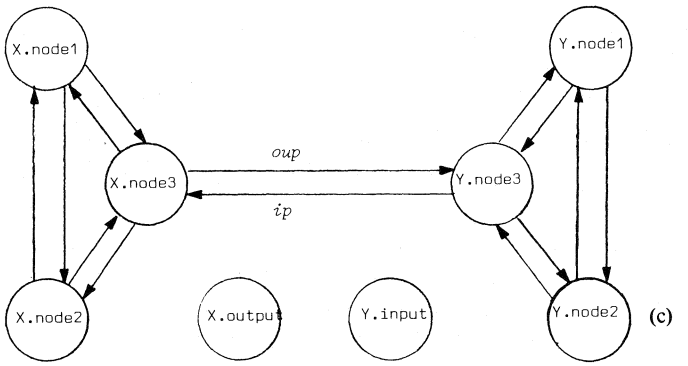


Fig. 4. (a), (b) Graphs showing classes of activity $\mathscr{A}$ and $\mathscr{B}$. (c) Graph of program formed by linking instances of $\mathscr{A}$ and $\mathscr{B}$.

5. Some Problems Concerning 'Type'

5.1. *Prohibition of synonyms*

Research into reasoning processes which underlies the composition of programs indicates that it is important to avoid any situation in which the same object can be called by more than one name. The formal reason for this is found in the axiomatic description of the semantics of the assignment statement; for whether this be expressed in terms of post-conditions implied by pre-conditions [7] or in terms of weakest pre-conditions required for given post-conditions [5] the description involves the concept of a 'predicate transformer' which substitutes an expression for every occurrence of a variable with a given name. If the possibility exists of a variable being called by more than one name then the reasoning process becomes complicated and prone to error, and it would be difficult for example to see the way to automating the verification of assertions under such conditions.

Not every programming language has been designed to meet this criterion. To achieve it even in sequential programming involves restriction, which is why in the design of PASCAL [7] the substitution of actual **var** parameters is carefully restricted, and variables accessible via pointers are segregated from those which are declared 'directly'. These precautions indicate the importance which has been attached to the principle of prohibiting synonymous names.

Unfortunately, name-graphs as described so far do allow multiple naming. The problem is *not* that different processes use different names for the same object (as reasoning about a parallel processing environment is bound in any case to extend beyond the application of simple predicate transformers to sequential sub-programs): it is the more serious problem that the same object can be accessed by the same process via different naming routes. In the code which drives a dining philosopher, for example, the names

$$pri, \qquad l.r.pri, \qquad l.l.l.l.l.pri$$

would be synonymous, and to attain the criterion of unique naming it is necessary to prohibit all but one of these. In the general case we need some principle restricting the use of the name-graph just sufficiently to provide one and only naming route to each object to which access is required.

5.2. *Visibility of attributes*

The kind of restriction required on the use of names is comparable to the control of visibility of the attributes of an object which is already established and manifest in several well-known programming languages, such as SIMULA (in later versions), PASCAL PLUS and ADA. These languages allow the programmer to select which attributes should be accessible from outside and in some cases to discriminate between read access and write access and even (with enumerated types) which constant values of a given type may be used; but to prohibit synonyms, such visibility control would in general need to be specified *individually* for each accessing process. The collection of attributes of and operations acting upon an object as seen through such an individually restricted view is what we shall call the *apparent type* of the object (i.e. how its type appears to the accessing process).

In the dining philosophers configuration for example (Fig. 2b) the main code for each philosopher needs only to access the attributes *s* and *test* of its neighbours, but the 'test' activity requires access to the *l* and *r* attributes of the philosopher with which it is associated and these neighbours should have an apparent type which allows access to the attributes *s, pri, l.s, r.s* but in a way which prevents synonyms like *l.r.s* from being formed. To achieve all the objectives, three different apparent types of philosopher need to be distinguished, namely:

philosopher = (*s*: state; *pri*: sem; *mut*: mutex; *test*: test routine;
l, *r*: neighbour);
neighbour = (*test*: test routine; *s*: state)
{prohibiting synonyms like *l*.*mut* (for *mut*),
l.*r*.*s* (for *s*) etc.};
tested phil = (*s*: state, *pri*: sem; *l*, *r*: neighbour)
{describing how the type of a philosopher
appears from inside its test routine and
prohibiting *ph*.*test*.*ph* as a synonym for
ph}.

5.3. *Graphs of apparent type*

It is convenient to embed visibility restrictions of the kind just described in a separate name graph called a *graph of apparent type.* In such a graph, each node is labelled with either a class name or the name of an apparent type and the colours of the arcs are taken from those of the main name-graph. Its use is to check the naming route for every reference in a class body (at compile-time) by following it through the graph of apparent type starting at the node labelled with the class name. The examples shown in Figs. 5a and 5b prohibit synonyms while providing all naming needs for the programs associated with Figs. 2a and 2b respectively.

A further use for a type-name-graph is to combine it with the main name-graph by directing arcs coloured to denote *class* from each class

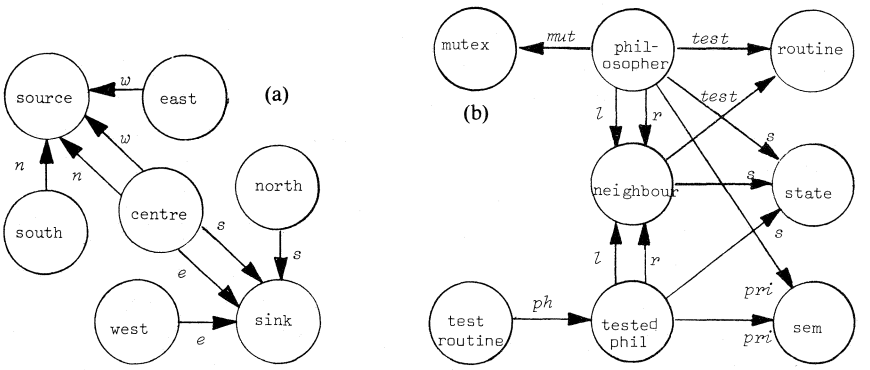


Fig. 5. Graphs of 'apparent type'. (a) Matrix × vector multiplication. (b) Dining philosophers.

instance to the node in the type graph with the appropriate class label. Such a combined graph would then include the information contained in the **'at'** statements in the program examples of Sections 3.1 and 3.2.

Finally we could ask the question: "Is it always possible to prohibit unwanted synonyms by using a suitable graph of apparent type?" and this can be answered almost trivially 'yes'; for as a last resort we could construct a 'forest' of trees where each tree would describe a set of unique naming routes for each node in the original name graph. Indeed, we *could* resolve the problem of synonym prohibition by constructing any spanning tree of the main name-graph; but in the examples already considered this would lead to close neighbours being addressed by circuitous routes and to totally unsystematic and irregular coding. The main questions here are pragmatic rather than graph-theoretic.

6. Conclusion

Affirmative answers to the first three questions posed are, I believe, indicated by the examples studied; for although the first two questions were subjective in character, it has been shown that the use of name-graphs has allowed subscripted references to give way to simple names and moreover to do so in a way which narrows the gap between language and machine. The operations defined on name-graphs of *union, Cartesian product, coloured directed join* and *colour contraction* seem to be useful and versatile tools for describing name-graphs. The question of whether proof techniques using global invariants will lend themselves to 'naming by colour' remains an open topic for further study.

The last question also seems to be answered by the use of name-graphs to define apparent type. In this way it seems practicable to describe and study a degree of finely selective visibility which is sufficient to avoid the possibility of 'synonyms'.

Acknowledgement

I would like to thank Frank Harary for pointing me in helpful graph-theoretic directions.

References

[1] J. Akiyama and F. Harary, A graph and its complement with specified properties, IV, J. Graph Theory 5 (1981) 103–107.

[2] E.W. Dijkstra, Cooperating sequential processes, in: F. Genuys (Ed.), Programming Languages (Academic Press, New York, 1968).

[3] E.W. Dijkstra, Asynchronous systems, conservation laws, and convergence to a steady state, Joint IBM University of Newcastle Seminar on Computers and Communications (4–7 September 1973).

[4] E.W. Dijkstra, Hierarchical ordering of sequential processes, Acta Informat. 1 (1971) 115–138.

[5] E.W. Dijkstra, Chapter 4, in: A Discipline of Programming (Prentice Hall, Englewood Cliffs, NJ. 1976).

[6] C.A.R. Hoare, Communicating sequential processes, Comm. ACM 21 (8) (August 1978) 666–677.

[7] C.A.R. Hoare and N. Wirth, An axiomatic definition of the programming language PASCAL, Acta Informat. 2 (1973) 335–355.

[8] H.T. Kung and C.E. Leiserson, Algorithms for VLSI processor arrays, in: C. Mead and L. Conway, Introduction to VLSI Systems (Addison-Wesley, Reading, MA, 1980).

[9] D. Loendorf, The finite element machine: an array of asynchronous microprocessors, ICASE Workshop (April 1980).

[10] J. Olszewski, A machine-oriented version of PASCAL: A proposal, Univ. of Reading, Dept. of Computer Science Internal Report, RCS 100 (April 1978).

[11] J.D. Roberts, A fast discrete fourier transform algorithm suitable for a pipeline vector processor, Univ. of Reading, Dept. of Computer Science Internal Report, RCS 82 (December 1977).

[12] J.D. Roberts, The construction of inherently tame asynchronous programs, Univ. of Reading, Dept. of Computer Science Internal Report, RCS 106 (August 1978).

[13] O.C. Zienkiewicz, The Finite Element Method (McGraw-Hill, New York, 1977).

Algorithmic Languages, de Bakker/van Vliet (eds.)

Optimization of Inductive Assertions

Henry S. Warren Jr.

IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, U.S.A.

Inductive assertions are assertions placed in the loops of a program, primarily to aid a mechanical correctness prover. Here we assume that the assertions in a program are executed along with the program. That is, the predicate expression of each assertion is evaluated when encountered during program execution, to verify that its value is *true*.

Inductive assertions are particularly expensive to execute. This is not only because they are in loops, but also because they are frequently themselves loops (quantified expressions). Thus executing them can slow a program's execution by a factor that can be indefinitely large.

We investigate the possibility of optimizing such quantified inductive assertions by substantially reducing the range of quantification. Many inductive assertions encountered in practice fall into a simple pattern in which the quantifier may, essentially, be removed. This restores the execution time of the program to the same order of magnitude that it would have been if the inductive assertions were not executed.

We emphasize methods that are no more costly in compiler size and execution time than conventional global optimization techniques.

1. Introduction

This paper explores ways to optimize inductive assertions in computer programs. The orientation is toward conventional high level languages (PL/I, ALGOL, PASCAL, etc.) that have been augmented to include an 'assertion' statement. The assertion statement allows simple bounded quantifiers over the predicates of the base language. Such quantifiers are the minimal equipment necessary to make significant statements about the facts alleged to hold at various points in a program, e.g., "array A is sorted," "x is the greatest common divisor of y," etc.

Inductive assertions are, of course, contained in loops. Frequently they are quantified expressions, and thus the quantified expression is a loop

within one or more containing loops. It is this pattern that we seek to optimize: a quantified expression contained in a loop. We are not really concerned with whether or not the quantified expression forms part of an inductive proof. We will show that little is gained by considering *only* assertions, and, if quantifiers are allowed in contexts other than in assertions, much would be lost.

We emphasize methods paralleling those found in conventional globally optimizing compilers. We restrict our attention to the use of transitive closure-like processes such as are found in data flow analysis and strength reduction problems, and we avoid 'theorem proving' methods, such as proving the equivalence or non-equivalence of expressions, and inventing inductive proofs. Although what constitutes a 'theorem proving' method is ill-defined, we mean to minimize complicated (long) algorithms unless they are likely to be in the compiler anyway, and we mean to completely avoid algorithms with exponential worst-case running time.

We first discuss two simple examples. Then we make necessary qualifying remarks about the assertion language. Next we discuss the important issue of safety, and lastly we give two methods for doing the optimization, and discuss their relative merits.

This paper is a summary of work done as a thesis at New York University's Courant Institute of Mathematical Sciences. It stems from work done there on algorithmic differentiation [2, 3]. The interested reader is referred to the thesis [4] for a more complete development.

2. Basic Examples

The optimization we are looking for is illustrated by the program below. This program searches a vector A, of length n, for the first component equal to a given item x. If found, it returns its index, and otherwise it returns zero.

```
        function search1(A, n, x);

        do i = 1 to n;
          if A(i) = x then go to out;
          end;
        return 0;                    "Not found."
out:    return i;                    "Found."
        end search1;
```

The main assertions for this program are (1) at the 'not found' exit, all components of A are not equal to x, and (2) at the 'found' exit, i indexes the first component of A that is equal to x. The inductive assertion asserts that all components checked thus far are not equal to x. No entry assertion is necessary, but for completeness we write 'assert **true**' at the entry point. The fully annotated program is shown below.

```
      function search1(A, n, x);
A1:   assert true;
      do i = 1 to n;
        if A(i) = x then go to A4;
A2:     assert 1 ≤ ∀k ≤ i: A(k) ≠ x;
        end;
A3:   assert 1 ≤ ∀k ≤ n: A(k) ≠ x;
      return 0;                    "Not found."
A4:   assert (1 ≤ ∀k < i: A(k) ≠ x) & A(i) = x;
      return i;                    "Found."
      end search1;
```

A2′: assert $A(i) \neq x$;

We wish to optimize assertion A2, as it is iterative and is in a loop. Observe that the only variable occurring in this assertion that varies in the loop is i: the upper limit of the quantifier. Furthermore, observe that i increases by one each time around the loop. Therefore, if the assertion was **true** on one pass around the loop, then on the next pass it is certainly true that

$$1 \leq \forall k \leq i-1: \quad A(k) \neq x,$$

and so the assertion will be **true** iff $A(i) \neq x$. Hence the assertion may be replaced by A2′ shown in the box below the program. Replacing A2 by A2′ restores the program from $O(n^2)$ back to the $O(n)$ execution time characteristic that the program would have without the assertions.

This is the basic pattern that is studied here. We will speak of this transformation as 'differentiating' the program; A2′ is the *derivative* of the original assertion with respect to the change $i \leftarrow i+1$.

Although there are several unstated assumptions in the above reasoning (e.g., we have used the fact that $i \geq 1$ in the loop, and have assumed that execution terminates if the assertion ever evaluates to **false**), this basic

pattern occurs in many programs. The pattern is:

(1) there is a quantified expression in a loop,

(2) the range of quantification increases or decreases monotonically, and

(3) the free variables in the quantified expression are loop constants.

For our methods to be practical, it is also necessary that we can easily detect when these conditions are satisfied. It is desirable and possible to relax condition (3) slightly, by allowing indexed array assignments in the loop.

Now let us consider another equally simple example, but one that uses $\exists$ quantifiers rather than $\forall$ (although $\forall$ quantifiers seem to be more common than $\exists$, with the expressions in prenex normal form). The program below is a somewhat contrived variation of *search1* for which the assertions would most naturally be writen with $\exists$ quantifiers. It searches a vector A for an arbitrary occurrence of a component equal to a given item x, and it is given the fact that A is certain to contain some component equal to x.

```
      function search2(A, n, x);
A1:   assert 1 ≤ ∃k ≤ n: A(k) = x;
      do i = 1 to n;
        if A(i) = x then go to A4;
A2:     assert i + 1 ≤ ∃k ≤ n: A(k) = x;
      end;
A3:   assert false;
      return 0;                    "Not found."
A4:   assert A(i) = x;
      return i;                    "Found."
      end search2;
```

```
A2′:  if i = 1 ∨ A(i) = x then
      assert i + 1 ≤ ∃k ≤ n: A(k) = x;
```

The program itself is identical to *search1*. The assertions, however, are entirely different. The input assertion specifies that some component of A is equal to x. The inductive assertion says that some component in the unexamined portion of A is equal to x.

Loop fall-through should never occur; 'assert **false**' is appropriate there.

The assertion we wish to optimize is A2. Notice that the range of the quantifier, $i+1$ to n, gets smaller and smaller as control flows around the

loop. One would expect this for existential quantifiers, and the opposite for universal quantifiers, as this makes them get stronger and stronger with successive loop iterations. (However, this is not *always* the situation.)

To optimize A2, consider the usual case in which A2 is executed, i.e., the case in which it has been executed before. If $A(i) \neq x$, then there is no need to check the assertion: its truth follows from the previous iteration. On the other hand, if $A(i) = x$, then the assertion must be checked for the whole range from $i+1$ to n. (Of course if $A(i) = x$, assertion A2 is not reached, but our optimizer would not know this. Furthermore, in more general situations, particularly when the program has bugs or incomplete assertions, the assertion must be fully checked.) Thus for the usual loop iteration, assertion A2 can be replaced by:

$$\text{if } A(i) = x \text{ then assert } i+1 \leq \exists k \leq n\colon \quad A(k) = x;$$

However, if that were all that were done, the revised program would not be the same as the original on first time through the loop. To properly handle this case, the assertion must be executed on first time through. The correct replacement for A2 is shown as A2′ above.

Our 'optimization' has, unfortunately, made the program larger. However, it is a bona fide optimization in time, because the quantified part of the assertion is only executed on first time through the loop. One would expect that in more general situations, also, the quantifier would not usually be executed.

The two methods to be described transform the program quite differently from what was just illustrated. However, they retain the property that the range of quantification executed is substantially reduced.

3. The Assertion Language

We require that the evaluation of assertions be without side effects, and this is an unpoliced rule. (The program must function in the same way with or without correct assertions being compiled with it.)

Bound variables are of strictly local significance, i.e., in:

$k = 0;$

assert $1 < \forall k < n$ ...

print k;

the value printed will be zero.

It will be helpful for the optimizer to know what happens if an assertion evaluates to **false**: Does the program terminate or does it continue to run? Must all the variables be available for dumping with the values set as if no optimization had been done? The greatest optimization is possible if false assertions result in termination, as will be seen. However, we do *not* make this assumption. The reason is that we wish to be able to optimize assertions of forms such as "assert $P \vee Q$"; after such an assertion it is not necessarily the case that P is **true**, or that Q is **true**.

We assume that the order of evaluation of $\forall$ and $\exists$ may be reversed, i.e., $m \geq \forall k \geq n$ may be replaced with $n \leq \forall k \leq m$. This is not essential but is a convenience for the optimization algorithms. It is used to bring quantifier expressions into a 'standard form', which is described in Section 5.

We also assume that the implementation of the language is free to evaluate quantifier expressions over their entire range of quantification, even if the value of the expression is apparent before the range is exhausted.

Lastly, we assume the implementation is free to evaluate all subexpressions of a quantified expression (i.e., the 'left-to-right rule' for Boolean expression is not used). This assumption is not essential, but it permits breaking up a quantifier expression to consider its parts separately for optimization. For example, in "assert $P \,\&\, Q$", it may be that no optimization can be done, because variables of P are assigned to in the region of optimization. We allow breaking this up (if necessary) into $t_1 = P$; $t_2 = Q$; assert $t_1 \,\&\, t_2$, and then consider separately P and Q for optimization. This 'breaking up' is not discussed here (see [4]).

4. Safety

There is no generally accepted and unambigous definition of 'safe' transformation. We take a fairly conservative position, and consider any of the following ill-defined operations to be unsafe:

(a) division by zero ('divide check'),

(b) fixed point overflow,

(c) subscript or substring range check, and

(d) use of an uninitialized value.

Following Kennedy [1], we allow the optimized program to execute

fewer ill-defined operations, and to execute them in different places and in a different order, but it must not execute any *new* ill-defined operations.

In searching for reasonably general and simple ways to optimize inductive assertions, problems of safety were frequently encountered. For example, let us see what happens if we apply the 'strength reduction' technique to the *search1* program. This technique will be reviewed in a later section; suffice it to say here that the technique is to maintain the current value of the expression being optimized in a variable t. The variable t is initialized at loop entry and is updated wherever the value of the expression being optimized might change. This transforms *search1* to the code below, where we explicitly show the loop control steps.

```
      function search1(A, n, x);

      i = 1;
      t = 1 ≤ ∀k ≤ i: A(k) ≠ x;   "Inserted".
L1:   if i > n then go to L2;
      if A(i) = x then go to out;
A1:   assert t;
      i = i + 1;
      t = t & (A(i) ≠ x);          "Inserted."
      go to L1;
L2:   return 0;
out:  return i;
      end search1;
```

A safety problem should be suspected, because we have inserted code at points which are not necessarily executed either before or after the point (A1) from which the code came. In fact *both* inserted evaluations of "$A(i) \neq x$" are unsafe. A subscript range check could occur at the first if $n = 0$. The second will evaluate $A(n+1)$, which could also cause a subscript range check.

These safety problems in *search1* are relatively easy to fix, because they merely involve illegal computations that "don't matter". That is, if they don't cause an interrupt (because of insufficient checking on the machine), then the program will still work correctly. If they do cause an interrupt, the proper fix is to simply ignore it. (This is not entirely trivial, because the language implementation must discern *which* interrupts can safely be ignored.)

However, there are more difficult situations. Since we are manipulating iterative expressions, we must take care not to inadvertently insert a near-infinite iteration. This could happen in connection with interchanging the order of quantifiers, which is sometimes desirable to do to optimize nested quantifiers [4]. Suppose we change the expression

$$1 \le \forall i \le m: \quad [P(i) \vee 1 \le \forall j \le n: Q(i,j)] \tag{1}$$

where $m, P,$ and Q are loop invariant, but n is varying, to:

$$1 \le \forall j \le n: \qquad 1 \le \forall i \le m: \quad [P(i) \vee Q(i,j)] \tag{2}$$

to simplify differentiating it with respect to changes in n. Then the possibility exists that when $m = 0, n$ is undefined (uninitialized). An attempt to evaluate (2) could then cause an enormous value to be used for n, which would cause the 'optimized' program to loop nearly indefinitely. The proper fix for this depends upon the details of the optimizing transformation being used. For example, in the strength reduction technique, we should insert (1) at loop entry, not (2), even though it is (2) that we are differentiating. However, the point being made is that no system that detects errors by means of interrupts can fix this type of safety problem (we assume that the language implementation does not include a check for uses of uninitialized data, which is usually the case with conventional HLL's).

We develop two methods of optimizing quantified expressions in loops, which observe the safety requirement in different ways. In the first, the 'min-k' method, we avoid safety problems by limiting ourselves to transformations that adhere to the principle that for every expression e inserted into the program in the course of optimizing it, the original program would evaluate e, either before or after the point at which we inserted e, for the same values of the arguments of e. Thus if the optimized program interrupts or loops indefinitely in evaluating e, the original program would also, although possibly sooner or later than the optimized program.

We will, however, find it necessary to allow minor violations of this, for example introducing an evaluation of $e+1$ in the neighborhood of an evaluation of e in the original program.

The second method uses the technique of strength reduction. We avoid this method's safety problems by (1) avoiding the use of uninitialized variables, and (2) by monitoring interrupts via a mechanism such as the PL/I 'ON' statement. If an interrupt occurs in an expression that was inserted by the optimizer, then a switch variable is set that causes re-

evaluation of the expression being optimized when it is reached at its original position in the program.

5. The 'min-k' Method

In this method we keep track of the index k_f of the least **false** value of $P(k)$ in the expression being optimized,

$$q = m \leq \forall k \leq n: \quad P(k), \tag{3}$$

or equivalently the first **true** value in the corresponding existential predicate. If (3) is **true**, we record this fact with a special value of k_f.

Reduced to its bare essentials, the method is to transform the program:

```
do while ...;
  ...
  n = n + n1;
  ...
  q = m ≤ ∀k ≤ n: P(k);
  ...
  end;
```

where m and $P(k)$ are loop-invariant, to:

```
kf = m;
do while ...;
  ...
  n = n + n1;
  ...
  kf = kf ≤ min k ≤ n: ¬ P(k);
  q = (kf > n);
  ...
  end;
```

The value of

$$m \leq \textbf{min}\ k \leq n: \quad P(k)$$

is the least k in the range m to n inclusive for which $P(k)$ is **true**, if such a k exists, or $n+1$ if $m \leq n$ and all $P(k)$ in the indicated range are **false**, or m if $m > n$.

This method has the following virtues:

(1) It does not require any code to be inserted at the points where n is assigned to, and thus avoids the safety problems that this would bring on.

(2) It gives the correct result and the evaluation of $P(k)$ is safe regardless of the value of n_1, i.e., regardless of how n varies with each loop iteration.

(3) It gives the correct result if initially $m > n$ (simpler methods break down in this case).

(4) Since no code is inserted at the assignments to n, these may be in a strongly connected region strictly contained in the strongly connected region containing the assignment to q, and the transformation is still an optimization.

(5) The method is correct and there is no safety problem if the assignment to q is conditionally executed in the loop.

(6) The optimization is particularly 'strong' in that if $n_1 \gg 1$ and q is **false**, $P(k)$ is evaluated only once per iteration.

However, the method should be extended to handle indexed assignments to arrays that occur in the quantified expression. A more complete description of the min-k method, that allows such array assignments, follows. This description is simplified from that in [4], in that here we assume that the quantifier being optimized is the outermost position (is not nested), and here we do not independently consider for optimization P and Q in the form "assert P & Q".

The optimization may be applied when the following conditions are met:

(1) A quantifier expression Q occurs in a strongly connected region R. Q is of one of the sixteen forms:

$$m \le \forall k \le n\colon \quad P(k),$$

$$m \le \exists k \le n\colon \quad P(k),$$

or

$$m > \forall k \ge n\colon \quad P(k),$$

etc., where m and n are integer-valued expressions independent of k, and P is a predicate expression possibly dependent on k.

(2) Q is free of side effects, and the order of evaluation of the quantifier may be reversed.

(3) At most one of the range limits m and n varies in R. ([4] shows how to remove this restriction, at some cost in compiler complexity.)

(4) All the free variables of P that are assigned to in R must be array variables, and the assignments must be array assignments satisfying:

(a) Each such assignment is of the form $A(x_i)=\ldots$, where x_i is an arbitrary integer-values expression (possibly involving A), except $x_i \neq -\infty$, the maximum negative number of the machine, and A is a vector (one-dimensional array) occurring in P only in the form $A(\pm k + c_j)$, where k is the bound variable of Q, and each c_j is invariant in R.

(b) The control flow is such that an assignment to $A(x_i)$ cannot be executed twice without an intervening execution of Q.

Observe that there are few control flow restrictions. We require that Q be in an SCR, but the structure of the SCR is arbitrary. The assignment to the variables of m or n, if any, may be in an SCR that is properly contained in R and does not contain Q. However, restriction (4(b)) implies that any relevant array assignments are not in such an SCR.

Restriction (4) implies that P does not contain a free occurrence of the varying range limit.

If the above conditions are met, the quantifier Q is optimized by the transformations given below.

(1) Convert the quantifier to 'standard form' as follows:

(a) If the quantifier is $\exists$, replace it with $\neg \forall \neg$.

(b) Replace 'decrementing' quantifiers with 'incrementing' ones, e.g., replace "$m > \forall k \geq n$" with "$n \leq \forall k < m$", etc.

(c) Replace "$m < \forall k < n$" with "$m+1 \leq \forall k \leq n-1$", etc., so that only '$\leq$' remains.

(d) If the lower limit m varies (while the upper limit n is constant), replace "$m \leq \forall k \leq n$: $P(k)$" with "$-n \leq \forall k \leq -m$: $P(-k)$".

The quantifier is now in the form

$$\mathrm{m} \leq \forall k \leq n: \quad P(k),$$

with m invariant in R. We will refer to this 'standard form' quantifier as Q_s.

(2) At each entry to R, insert the code (outside of R):

$$k_f = -\infty;$$
$$t_1 = -\infty;$$
$$\ldots$$
$$t_r = -\infty;$$

where there are r array element assignments in R as described in restriction

(4) above, and $-\infty$ denotes the maximum negative number representable on the machine.

(3) Replace each such assignment $A(x_i) = \ldots$ with

$$t_i = x_i;$$

$$A(t_i) = \ldots;$$

(This is simply to capture the subscript value in the variable t_i.)

(4) Replace the quantifier in standard form,

$$q = m \le \forall k \le n: \quad P(k),$$

with the following code:

```
if m ≤ n then do;
   if k_f = -∞ then k_f = m;
┌  if t_i ≠ -∞ then do;
│  ┌ k_d = ±(t_i - c_j);
│  │ if m ≤ k_d < k_f & k_d ≤ n
│  └ then if ¬P(k_d) then k_f = k_d;
│    t_i = -∞;
└    end if t_i;
   k_f = k_f ≤ min k ≤ n; ¬P(k);
   q = (k_f > n);
   end;
else q = true;
```

where the code indicated by the inner bracket is repeated for each $j = 1, 2, \ldots, s$, where s is the number of occurrences of $A(\pm k + c_j)$ in P, and the code indicated by the outer bracket is repeated for each $i = 1, 2, \ldots, r$ (giving $(3s+3)r$ lines of code represented by the six bracketed lines). In the assignment "$k_d = \pm(t_i - c_j)$", the $\pm$ is to be taken in the same sense as the $\pm$ in the corresponding term $A(\pm k + c_j)$ occurring in Q_s.

The main idea of this transformation is that whenever an array assignment

$$A(x) = \ldots;$$

occurs that can affect $P(k)$ in Q_s, we reduce the value of k_f, if necessary, to

maintain the truth of:

$$k_f \geq m \;\&\; m \leq \forall k \leq k_f - 1: \quad P(k),$$

which is the invariant that makes the method work. If $P(k)$ contains a reference to $A(k+c_j)$, the assignment to $A(x)$ might affect the value of $P(k)$ for $x = k + c_j$, i.e., for $k = x - c_j$. Therefore, before evaluating the **min** k functional that replaces Q_s, we reduce k_f to $x - c_j$ if (1) $m \leq x - c_j < k_f$, (2) $x - c_j \leq n$ (for reasons of safety) and (3) $P(x - c_j)$ is **false**.

We do not attempt to 'correct' Q_s at the point of the assignment $A(x_i) = \ldots$; this would lead to safety problems. Instead, at the point of the assignment we save the value of the subscript x_i, and use it to correct Q_s at the point where Q appeared in the original program. Since we have introduced only one temporary t_i for each assignment $A(x) = \ldots$, it is necessary that this temporary be used (if the quantification ranges are such that it will be used) to update Q_s before the assignment is reached again. That is the reason we require that the control flow be such that an assignment to $A(x_i)$ cannot be executed twice without an intervening execution of Q (restriction (4(b))).

The test "if $m \leq n \ldots$" may seem unnecessary. However, it provides a valuable safety check. Without it, there would be a possibility of evaluating c_j when the original program would not. This would cause trouble in a quantifier expression such as

$$1 \leq \forall k \leq n: \quad \ldots A(k + 1/c) \ldots$$

The optimized program without the "if $m \leq n \ldots$" test would cause a divide check if conditions are such that when $n < 1, c = 0$.

Reference [4] contains a proof that the transformation is correct, and is safe within limits that are defined there.

Below is a sample program, the 'insertion sort', before and after optimization by the min-k method. Assertion A1 is optimized. Assertion A2 is not optimizable because an unbounded number of elements of A get assigned to between two successive times that control reaches A2 (restriction (4(b)) is not satisfied for any SCR that contains A2).

```
        procedure insort(A, n);
        do j = 2 to n;
          i = j - 1;
          x = A(j);
    L:      if x < A(i) then do;
              A(i + 1) = A(i);
    A1:       assert i < ∀k ≤ j: [A(k - 1) ≤ A(k) & x < A(k)];
              i = i - 1;
              if i > 0 then go to L;
              end if x;
          A(i + 1) = x;
    A2:   assert 1 ≤ ∀k < j: A(k) ≤ A(k + 1);
          end do j;
        end insort;
```

```
 1.     procedure insort(A, n);
 2.     do j = 2 to n;
 3.       i = j - 1;
 4.       x = A(j);
 5.       k_f = -∞;
 6.       t_1 = -∞;
 7.   L:    if x < A(i) then do;
 8.           t_1 = i + 1;
 9.           A(t_1) = A(i);
10.   A1:     if i < j then do;
11.             if k_f = -∞ then k_f = -j;
12.             if t_1 ≠ -∞ then do;
13.               k_d = -(t_1 + 1);
14.               if -j ≤ k_d ≤ k_f & k_d ≤ -(i + 1) then if
15.               ¬[A(-k_d - 1) ≤ A(-k_d) & x < A(-k_d)] then k_f = k_d;
16.               k_d = -t_1;
17.               if -j ≤ k_d < k_f & k_d ≤ -(i + 1) then if
18.               ¬[A(-k_d - 1) ≤ A(-k_d) & x < A(-k_d)] then k_f = k_d;
19.               t_1 = -∞;
20.               end if t_1;
21.             k_f = k_f ≤ min k ≤ -(i + 1);
22.             ¬[A(-k - 1) ≤ A(-k) & x < A(-k)]
23.             q = (k_f > -(i + 1);
```

```
24.                 end;
25.               else q = true;
26.               assert q;
27.               i = i − 1;
28.               if i > 0 then go to L;
29.               end if x;
30.           A(i + 1) = x;
31.   A2:     assert 1 ≤ ∀k < j: A(k) ≤ A(k + 1);
32.           end do j;
33.       end insort;
```

6. The Strength Reduction Method

In this section we show how to apply the strength reduction, or 'formal differentiation', technique of Paige and Schwartz [3] to the optimization of inductive assertions.

We will replace $\forall$ and $\exists$ with arithmetic summation, with the quantified expression having **true** treated as 1 and **false** treated as 0. The assignment:

$$q = m \leq \forall k \leq n: \quad P(k);$$

is replaced with:

$$C = \sum_{k=m}^{n} \neg P(k);$$

$$q = (C = 0);$$

where C is a compiler-generated integer-valued temporary. We transform $\exists$-expressions similarly. The expression to optimize now is the summation.

We make this transformation for two reasons. First, it permits differentiation with respect to both increasing and decreasing changes in the range of quantification. Second, it is necessary to handle differentiating with respect to array element assignments by the Paige–Schwartz method, which requires a subtractive correction followed by an additive correction to the expression being differentiated.

The optimization in a typical situation is illustrated in Table 1.

The assignment "$C = C - [A(i) < x]$" is evaluated by evaluating the predicate, converting **true** or **false** to 1 or 0, respectively, and subtracting the 1 or 0 from C.

Table 1

Original	*Optimized*
do $i = 1$ to n;	$i = 1$; $C = \sum_{k=1}^{i} [A(k) < x]$; L: if $i > n$ then go to out;
$A(i) = \ldots$;	$C = C - [A(i) < x]$; $A(i) = \ldots$; $C = C + [A(i) < x]$;
$C = \sum_{k=1}^{i} [A(k) < x]$;	(no code).
end;	$C = C + [A(i+1) < x]$; $i = i + 1$; go to L; out:...

The transformation has several safety problems:

(1) If $n \leq 0$, the optimized program evaluates "$A(1) < x$" (second line), whereas the original program does not. Possibly in this case neither A nor x is defined, e.g., they might have the PL/I BASED storage attribute, and the base pointers may not be initialized. Then the attempt to evaluate "$A(1) < x$" can cause an addressing exception. Also, if $n \leq 0$, array A might be of zero extent, in which case the reference to $A(1)$ would cause a subscript range check. If these interruptions do not occur, there might be an overflow interruption if '<' is implemented by subtraction (because of the reference to an undefined quantity).

(2) If $n > 0$, the optimized program evaluates $A(n+1)$ (in the term $[A(i+1) < x]$, on the last loop iteration), whereas the original program does not. This could cause a subscript range check or overflow interruption.

(3) Even if the reference to $A(n+1)$ does not in itself cause an interruption (e.g., if subscript range checking is disabled), it can cause another problem. It causes the optimized program to calculate a slightly larger maximum value of C than the original. This is not a serious problem for the above program (surely $C < 2^{31}$), but there are probably analogous situations involving other functionals, such as the $\sum$ of real numbers, where this problem (possibility of overflow) cannot be ignored.

(4) $A(i)$ in the fourth line, and also $A(i+1)$ in the term $[A(i+1) < x]$, may be undefined (for *any* i), as the statement "$A(i) = \ldots$" may be initializing the array. These references to an undefined quantity might cause an overflow interruption, etc.

These safety problems may be solved as follows. Problem (1) is caused by evaluating the expression being optimized at loop entry, when the loop may in fact not be executed at all. The interrupt cannot be simply ignored, because it may be a 'legitimate' one that would have occurred in the original program (if the loop *is* executed one or more times). We solve this problem by leaving the expression being optimized where it was in the original program, but conditionally executing it under control of a Boolean switch variable *sw*. At loop entry, we set *sw* to **true**, which causes C to be initialized when it is first encountered in the loop. Then, *sw* is set to **false**, so that subsequently C will be calculated in the more efficient, strength reduced, way.

The second problem, that the optimized program evaluates $A(n+1)$ on the last loop iteration, is solved in a similar way (note that a 'legitimate' interrupt could also occur at this point). We replace the code

$$i = i + 1,$$

with the following:

on error *sw* = **true**;
$C = C + [A(i+1) < x]$;
revert error;
$i = i + 1$;

The 'on error' statement signals the operating system to execute the code "*sw* = **true**;" if any type of interrupt should occur after execution of the 'on error' statement. After applying the differential correction to C, the program executes 'revert error'. This signals the operating system to cancel the last executed 'on error', and to revert to the previous error action, whatever it was. If no interrupt occurs, C is differentially updated. If an interrupt occurs, the value of C is undefined, but *sw* is set to **true**. If control never reaches the point of the original expression being optimized, then the interrupt is in effect ignored, as it should be. If control does reach the point of the original expression, then the expression will be reevaluated in its original form, and the interrupt will occur, as it should.

Problem 3 above (that the optimized program calculates a slightly larger value of C than the original) is also solved by this technique.

Problem 4, that the first reference to $A(i)$ in the code:

$C = C - [A(i) < x]$;
$A(i) = \ldots$;
$C = C + [A(i) < x]$;

may be undefined, is solved in a similar way. We surround the first and last of the three statements above with 'on error – revert error.'

The complete transformation of our skeletal example is shown in Table 2.

Table 2

Original	*Optimized*
do $i = 1$ to n;	$i = 1$; sw = **true**; L: if $i > n$ then go to out;
$A(i) = \ldots$;	on error sw = **true**; $C = C - [A(i) < x]$; revert error; $A(i) = \ldots$; on error sw = **true**; $C = C + [A(i) < x]$; revert error;
$C = \sum_{k=1}^{i} [A(k) < x]$;	if sw then do; $C = \sum_{k=1}^{i} [A(k) < x]$; sw = **false**; end;
end;	on error sw = **true**; $C = C + [A(i+1) < x]$; revert error; $i = i + 1$; go to L; out:...

The 'on' and 'revert' statements may be implemented in a way that has practically no cost in execution time, as long as interrupts do not occur. The technique involves the creation of tables that define the beginning and ending addresses of the machine code that is bracketed by 'on' and 'revert'; see [4] for details.

The reader is also referred to [4] for a detailed description of when this transformation may be applied, exactly how to do it, a proof that it is correct, and a proof that it is safe, with minor qualifications.

Below we show the 'insertion sort' program after optimization by the strength reduction method.

```
1.      procedure insort(A, n);
2.      do j = 2 to n;
3.          i = j − 1;
4.          x = A(j);
5.          sw = true;
6.      L:      if x < A(i) then do;
7.                  on error sw = true;
8.                  if i + 1 ≤ (i + 1) − (−1) ≤ j then C = C −
9.                      ¬[A(((i + 1) − (−1)) − 1) ≤ A((i + 1) − (−1))
                                                & x < A((i + 1) − (−1))];
10.                 if i + 1 ≤ (i + 1) − 0 ≤ j then C = C −
11.                     ¬[A(((i + 1) − 0) − 1) ≤ A((i + 1) − 0)
                                                & x < A((i + 1) − 0)];
12.                 revert error;
13.                 A(i + 1) = A(i),
14.                 on error sw = true;
15.                 if i + 1 ≤ (i + 1) − (−1) ≤ j then C = C +
16.                     ¬[A(((i + 1) − (−1)) − 1) ≤ A((i + 1) − (−1))
                                                & x < A((i + 1) − (−1))];
17.                 if i + 1 ≤ (i + 1) − 0 ≤ j then C = C +
18.                     ¬[A(((i + 1) − 0) − 1) ≤ A((i + 1) − 0)
                                                & x < A((i + 1) − 0)];
19.                 revert error;
20.     A1:         if i < j then do;
21.                     t = i + 1;
22.                     on error sw = true;
23.                     C = C + ∑_{k=t}^{min(m−1, j)} ¬[A(k − 1) ≤ A(k) & x < A(k)];
24.                     revert error;
25.                     m = t;
26.                     if sw then do;
27.                         C = ∑_{k=m}^{j} ¬[A(k − 1) ≤ A(k) & x < A(k)];
28.                       sw = false;
29.                       end;
30.                   q = (C = 0);
31.                   end;
32.               else q = true;
33.               assert q;
34.               i = i − 1;
```

```
35.              if i > 0 then go to L;
36.              end if x;
37.          A(i + 1) = x;
38.    A2:   assert 1 ≤ ∀k < j: A(k) ≤ A(k + 1);
39.          end do j;
40.       end insort;
```

7. Summary

We have studied a number of examples of inductive assertions. We have observed that they frequently involve quantifier expressions, and that there are many opportunities to substantially optimize these quantifier expressions. In many cases these optimizations can be done by straightforward extensions of the facilities that are normally found in globally optimizing compilers.

We believe the optimization methods described fit in well with a conventional globally optimizing compiler. The main functions that are normally included in such a compiler, and that are used by our optimizing methods, are control flow analysis and data flow analysis. Either interval analysis or SCR analysis would be adequate for our purposes. In the 'min-k' method, we use data flow analysis only to the minor extent of detecting which variables and expressions are invariant in each region. In the strengh reduction method, we use data flow analysis for this and also for a map that gives all the 'definition' (assignment) points that reach a given use.

A sampling of programs was obtained from various sources in the literature, which contained a total of 38 inductive assertions. In this sample, it was found that about 70% of the inductive assertions could be optimized by some algorithmic differentiation technique. The methods we have described get about 70% of these, which amounts to about 50% of all the quantifiers appearing in loops. When the optimization does apply, it frequently gives an 'order of magnitude' improvement in execution speed.

The table below compares the two methods of optimizing inductive assertions that were given. The methods are compared as regards range of applicability, output code volume, and complexity added to the compiler. Both methods are approximately equal in output code execution time.

In Table 3, R is the strongly connected region with respect to which

Table 3

Min-k	Strength reduction
Easily handles only one varying range limit	Both range limits may vary
No restriction on placement of assignments to range limits in R	No assignment to a range limit may be in an SCR that is wholly contained in R and that does not contain Q
All array references in Q that undergo assignments in R must be of the form $A(\pm k + c_j)$ with c_j invariant in R	All array references in Q that undergo assignments in R must be of the form $A(\pm k + c_j)$ with c_j a constant (known at compile-time) and all signs of k the same
Arbitrary assignments to range limits are allowed	Range limits must be recursively additive or monotonically increasing (upper limit) or decreasing (lower limit)
Optimized code does not generate interrupts	Optimized code may generate interrupts; some sort of interface with the operating system, such as 'on units', is required to allow execution to continue
Inserted code volume is large at the point of Q, small at other points	Inserted code volume is large at assignments to range limits and to arrays that appear in Q, small at other points
Complexity added to compiler is 509 SETL source lines of code	Complexity added to compiler is 687 SETL source lines of code

optimization is being done, Q is the quantified expression being optimized, and A is an array that is referenced in Q and that is assigned to in R.

Acknowledgements

I am indebted to Professor Schwartz for having first observed that this optimization might be useful, and for guidance during its development. I have also benefitted from technical discussions with Robert Paige.

References

[1] K. Kennedy, Saftey of code motion, Int. J. Comput. Math. Section A 3 (1972) 117–130.

[2] R. Paige, Expression continuity and the formal differentiation of algorithms, Thesis, in: Data Structure Choice/Formal Differentiation, Two Papers on Very High Level Program Optimization, Courant Computer Science Report No. 15, Courant Institute of Mathematical Sciences, New York University (September 1979).

[3] R. Paige and J.T. Schwartz, Expression continuity and the formal differentiation of algorithms, Conf. Rec. 4th ACM Symp. on Principles of Programming Languages (1977) 58–71.

[4] H.S. Warren Jr., Optimization of inductive assertions, PhD Thesis, New York University, Courant Institute of Mathematical Sciences (October 1980).

Algorithmic Languages, de Bakker/van Vliet (eds.)

The Design of Vector Programs

Alain Bossavit and Bertrand Meyer

Direction des Etudes et Recherches, Electricité de France, Clamart, France

Current vector computers such as the Cray-1, Cyber 205 S1, DAP or BSP pose a special challenge to the software designer as the available software tools and techniques are far behind the hardware developments, and the goals of efficient vector programming seem to conflict with some of the basic principles of good software engineering. After studying some properties of these computers, with particular emphasis on the Cray-1, we purport to show that a systematic approach to vector programming is possible and fruitful; the proposed methods are applied to the systematic, proof-oriented derivation of several vector algorithms. Language aspects are also considered.

1. Introduction

The advent of 'second-generation' vector processors [8] such as the Cray-1, CDC Cyber 205, Lawrence Livermore Laboratory S1, ICL DAP and Burroughs BSP, is one more piece of evidence for the fact that software lags far behind hardware as far as practical industrial usage is concerned. These computers, built with the latest LSI or VLSI technology in highly optimized architectures, are capable of achieving speeds which were unheard of before: for example, a Cray-1 computer will in good conditions carry out more than 100 million 'actual' operations, excluding control, per second. On the other hand, a look at the software provided with these 'super-computers' will show them to be what may be called **Fortran machines:** even though processors for other languages may exist, these computers are obviously tailored to a philosophy of programming which has the static array as its only data structure and the DO-loop as its main control structure. Recipes given for writing efficient programs in that

framework [6], seem at first glance to be very far from modern ideas about programming, if not incompatible with them.

Vector programming thus appears as a challenge for the software specialist. Areas where advances are needed include the following interrelated topics:

(1) algorithmics (algorithms for vector processing, and methods for finding such algorithms);

(2) program design (how to find program and data structures which will lead to efficient use of supercomputers while ensuring other program qualities such as reliability, clarity, portability, modularity, etc.);

(3) program transformation (methods for adapting existing programs to efficient execution on vector computers);

(4) languages for vector programming;

(5) proof methods.

The aim of this paper is to lay some foundations for a systematic treatment of vector programming. It is mostly concerned with (1) and (2), with a brief discussion of (4).

The particular machine which motivated this study is the Cray-1 computer, which seems to be the most widely available among the 'second generation' vector machines, and is quoted as the fastest currently available computer, even in scalar mode [4, 8]. Most of the discussion is, however, also valid for the other machines.

In Section 2, we give a software interpretation of the rules which must be obeyed by a computation in order to be able to use the vectorization capabilities of the hardware. In Section 3, we give a more abstract interpretation of these rules in terms of the data types involved. Section 4 discusses language problems. Section 5 is devoted to a study of systematic program construction techniques applied to vector programming; several algorithms, in particular a 'vector Cholesky', are derived.

2. Rules for Vectorization

Vector machines require that a program satisfy certain conditions in order to be vectorizable, i.e. amenable to processing in vector, as opposed to scalar, mode. The study of these conditions is particularly interesting in the case of vector computers such as the Cray-1 or BSP which accept standard FORTRAN, so that vectorization rests with the compiler rather

than the programmer. Abstracting from machine peculiarities, five basic conditions appear as necessary and sufficient:
- repetitive series of operations;
- primitive operations only;
- regularity;
- no backward dependency;
- no cross dependency.

These conditions are studied in [12] for the Cray case. We shall outline them here in general terms.

2.1. *Repetitive series of operations*

The only sequences amenable to vectorization are loops, and, more precisely, ***for*** loops, i.e. counter loops with a number of executions known at the outset. The ***for*** loop control structure, associated with the array data structure, is the software representative of the so-called SIMD (*Single Instruction stream, Multiple Data stream*) mode of restricted parallelism.

2.2. *Primitive operations only*

With some slight extensions, only assignments and numerical or boolean operations are allowed in a vector loop. This precludes in particular jumps, thence conditional statements other than conditional assignments. The Cray-1 Fortran compiler (CFT) will also inhibit vectorization of a loop containing a subprogram call (except the subprogram is known to CFT as having a vector version) or another loop (thus restricting vectorization to the innermost loops).

2.3. *Regularity*

For a loop to be vectorizable, it must involve only 'regular' array elements, i.e. elements whose indices follow a strictly defined pattern, so that they can be fetched in advance for vector operations. On the Cyber 205, the only regular elements are those which are stored contiguously; on the Cray-1, a sequence is regular iff the distance between successive elements is constant (but not necessarily 1). Thus only certain types of subarrays may be processed in vector mode.

2.4. *No backward dependency*

Let a loop with i as a counter contain the following array element assignment:

$$a[f_0(i)] := \textbf{\textit{op}}(b_1[f_1(i)], b_2[f_2(i)], \ldots, b_m[f_m(i)])$$

where ALGOL-like brackets are used for array elements, ***op*** is some numerical or logical operation, the f_k's are linear functions (from the regularity rule), and all arrays are considered as one-dimensional (which is always possible on a machine with a linear store).

This assignment has a backward dependency, which will inhibit vectorization, iff for some k ($1 \leq k \leq m$) b_k is a, and for some pair of values p, q in the range of i, the following holds:

$$p < q \textbf{ and } f_k(p) = f_0(q).$$

In other words, the computation of $a[f_0(q)]$ will use the value of another element of a, which was fetched for updating in some previous iteration. For example, the assignment $a[i] := a[i-1] + 1$ introduces a backward dependency.

The reason for this rule is that the vector interpretation of such a computation would use the old value of the array element, not the new one as in the standard (sequential) interpretation of the loop.

Note that the vector interpretation makes perfect sense; it is only different from the sequential one.

On the Cray-1 the condition is less stringent; a backward dependency will actually arise only if the above condition holds together with

$$q - 64 < p$$

where 64 is the length of the vector registers, which on the Cray must be used for the operands and results of vector operations (in contrast, the Cyber 205 and BSP work directly on vectors stored in memory). Vector processing on the Cray-1 may be considered, for all practical purposes, as successive processing of 64-element vector slices, all elements in a slice being processed in parallel.

An important case of backward dependency occurs when the dependency affects a simple variable (which may be considered as a one-element array, whose index is constant through the loop), i.e. when the loop contains an assignment of the form

$$x := \textbf{\textit{op}}(x, b_1[f_1(i)], b_2[f_2(i)], \ldots).$$

Such an operation is called a **reduction;** it is particularly unfortunate that it should not vectorize, since it corresponds to the very common case of accumulating a result into a variable, as in the computation of the sum of the elements of a vector, or of the scalar (inner) product of two vectors. In practice, techniques exist for reducing the loss of efficiency of reductions as compared to truly vectorizable operations; reductions may thus be thought of as 'pseudo-vectorizable' operations who execute more slowly than vectorizable operations but faster than scalar ones.

2.5. *No cross dependency*

Let a loop contain the following assignments:

$$a[f_0(i)] := \boldsymbol{op}(\ldots);$$

$$c[g_0(i)] := \boldsymbol{op}'(\ldots, a[g_1(i)], \ldots).$$

They induce a cross dependency, which will inhibit vectorization, iff for some pair of values p, q in the range of i, the following holds:

$$g_1(p) = f_0(q)$$

with $|q-p| < 64$ (on the Cray-1).

For example, the following statements in a loop on i will cause a cross dependency:

$$a[i] := 1; \qquad c[i] := a[i+1].$$

The rule stems from the fact that, due to the limited size of the instruction buffers, long loops may have to be split into several shorter ones in order to be vectorized (by slices of 64 on the Cray); thus the two assignments might end up in two different loops, giving a different semantics for the program. In our example, assuming a was initially all 0, then c would receive the previous null values in the sequential case and the new unity values in the vector case.

3. Basic Thoughts for a Vector Programming Methodology

Considering the preceding rules, even though they do not include many details which may be found in manufacturers' documentation, it is quite tempting to dismiss them as too low-level and machine-dependent, and

assert that vector programming is just programming with objects of data type 'vector'. Although we will use this definition as the basis for our approach to vector program construction, it should be pointed out that it is not quite sufficient and that the previous rules, especially the last ones on dependency, must also be taken into account for practical purposes.

Let us illustrate this point with an important vector algorithm: matrix multiplication. Assume c is initialized to zero; a, b, c have dimensions (m,n), (n,p) and (m,p) respectively. The ordinary algorithm will not vectorize (notations are mostly taken from [11]):

```
for i in 1, ..., m do
    for j in 1, ..., p do
        for k in 1, ..., n do
            c[i,j] := c[i,j] + a[i,k] * b[k,j]
```
(3.1)

In terms of the preceding rules, we may say that $c[i,j]$ has a backward dependency on itself (the last line is a reduction). Now if we reverse the loops on j and k, the program becomes vectorizable. This in fact means that instead of the 'element' formula which forms the basis for algorithm (3.1):

$$c[i,j] = \sum_{k=1}^{n} a[i,k] * b[k,j]$$

one relies on the 'vector' formula

$$c[i,*] = \sum_{k=1}^{n} a[i,k] * b[k,*]$$

(where $x[i,*]$ and $x[*,j]$ respectively denote the ith line and jth column of matrix x).

However, if we applied a purely functional view of vector programming, i.e. obtained a program directly from an 'abstract data type' specification of matrix multiplication, the initial version of our program, as deduced from the last formula, would require, for each line i, n vector variables:

$$c_1[i,*] := a[i,1] * b[1,*];$$
$$c_2[i,*] := a[i,2] * b[2,*] + c_1[i,*];$$
$$\ldots$$
$$c_m[i,*] := a[i,m] * b[m,*] + c_{m-1}[i,*];$$
$$c[i,*] := c_m[i,*].$$

For practical reasons (storage) this is excluded; the same variable $c[i, *]$ has to be used all along. This programming simplification is correct because it does not conflict with the no backward dependency rule, as every operation of the form

$$c[i, *] := op(c[i, *])$$

will be implemented as a counter loop whose body is $c[i,j] := op(c[i,j])$ without any reference to $c[i, l]$ for $l \neq j$ (note that the loop counter here is j). This condition guarantees that the vectorized form of the new version (i.e. the standard program where loops on j and k have been interchanged) is indeed semantically equivalent to the standard program.

Such a condition, which is more restrictive but conceptually simpler than the no backward dependency rule, may be used as a replacement for it in a systematic approach. It can be formalized in the following way, inspired from the presentation of sequences in the specification language *Z* [1]. Let $VEC\ X[(n)]$, for $n \in \mathbb{N}$ (the set of n-vectors of elements of X) be defined as the set of all total functions from $1, \ldots, n$ to X. Let & be the functional binary operator such that, if f and g are two functions with the same domain Y, then $f \& g$ is the function h such that, for any $y \in Y$, $h(y)$ is the pair $(f(y), g(y))$. Then for any binary operation p on X ($p : X \times X \to Z$ for some Z) we may define a vector **extension** of p, $ext(p) : VEC[X](n) \times VEC[X](n) \to VEC[Z](n)$, whose value for any two vectors v and w in $vec[X](n)$ is

$$ext(p)(v, w) = p \circ (v \& w)$$

where $\circ$ is functional composition; in other words, for any $i \in 1, \ldots, n$,

$$ext(p)(v, w)(i) = p(v(i), w(i)).$$

It is possible to define in the same way (at least if p is associative) a vector **reduction** of functionality

$$red(p) : VEC[X] \to X$$

where $red(+) = \sum$, etc.

We shall interpret the rules of Section 2 as implying that, in designing programs for vector computers, one should work on objects of data type vector, restricting oneself to extension operations as much as possible. When an extension operation cannot be applied, a reduction will still be preferable to operations which would perform arbitrary shifting of indices

(e.g. $p \circ ((v \circ pred) \& w)$, where *pred* is the predecessor function on integers, which would give $p(v(i-1), w(i))$ for any i); such operations would introduce hopeless backward dependencies.

The situation may be depicted using a hierarchy of abstract machines (Fig. 1). At the matrix level, machine MAT offers the operations of matrix algebra: multiplication, inversion, etc. At the vector level, several machines are available to implement these operations: the extension machine EXT, the reduction machine RED, and others. Choosing one of them will lead to a definite algorithm , the scalar machine SCAL, which corresponds to conventional programming languages. It is clear that the standard matrix multiplication algorithm given above (3.1) stems from the RED machine, while its vectorizable counterpart will come out naturally if one uses the EXT machine.

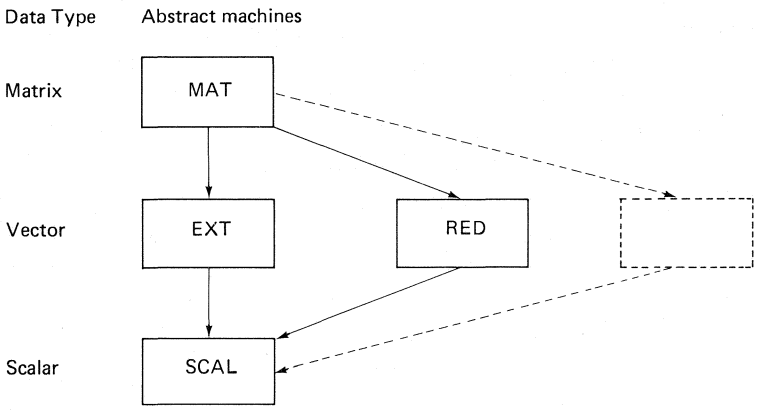


Fig. 1. Hierarchy of types and virtual machines.

Using the above approach, we will derive vector algorithms by working on vector objects from the beginning. This should lead to programs which are both properly structured and efficient on a vector processor. This should be contrasted with the results obtained through more 'ad hoc' methods. For example Higbie [6], in a paper on how to write code which will vectorize on the Cray, warns that 'overly modular or structured programs' will not be vectorizable (because of the rule which we called 'primitive operations only', precluding subprogram calls inside a vectorizable loop). If this were true, the situation might be considered quite sad for the programmer, forced to choose between structure and vectorization. On

the other hand, if one agrees that a program is 'structured' at least as much from its proper adequation of control structure to data structure as from its observance of rules regarding control structure only (e.g. many subprograms, etc.), then the answer is clear: rather than in-line expansion of subprogram calls in loop bodies, one should strive to write subprograms working on entire arrays (to use expressions found in Cray publications, "put the loop in the subroutine rather than the subroutine in the loop"). This will, in effect, implement the 'vector' data type abstraction. If the program is indeed vectorizable, i.e. if it does have vectors as its principal objects, there is a good chance that the version thus 'vectorized' will be clearer and better 'structured' independently of any machine consideration.

4. Language Considerations

Before we turn to the derivation of a few vector algorithms, we must pay some attention to language issues. The Cray approach uses a standard language, FORTRAN, and places the task of detecting vectorizable portions of code upon the compiler. The BSP also has a 'vectorizer' for standard FORTRAN code (an introduction to the techniques used for such program transformations may be found in [10]). Other methods have been used or suggested (see [9] or [14] for a survey); for example, the Cyber 205 supercomputer only vectorizes calls to special array processing subroutines. Perrott [14, 15, 16] has argued repeatedly in favor of using a language designed specifically for vector programming; he describes such a language, ACTUS, based on PASCAL. This approach can be justified on several grounds:

- In the Cray and BSP approach to optimization, the programmer has to present his code in a 'favorable' way so that the compiler will be able to detect vectorizable pieces of code; he thus has to know the compiler's idiosyncracies in this respect. This, however, has to be balanced with the considerations on program structuring expressed above.
- The search for vectorizable code amounts to de-compilation (reconstructing higher-level vector constructs, such as they might be expressed in ALGOL 68, PL/I or APL, from lower-level FORTRAN scalar operations), which is a rather silly activity;
- It is quite natural to specify the amount of allowable parallelism in

connection with the data structure definition rather than with the description of the operations performed on it.

On the other hand, the 'vector language' approach seems extremely difficult to implement in the context of a large scientific computing center (the typical target for supercomputers), where it is not realistic to imagine that programmers will turn to a new language for every new kind of application and every new machine – especially at a time when concerns for portability are at last making their way into the scientific programming community.

Given the failures experienced by all previous efforts to impose languages other than FORTRAN to this community, it is doubtful that a proposal applying to vector computers would succeed. In view of the current state of the art, the Cray approach seems sensible as far as program **coding** is concerned. Languages such as ACTUS may, however, be very useful as intermediary notations for vector program **design,** and we shall use similar ways of expression in the examples which follow.

5. Examples of Systematic Vector Program Construction

We turn now to the application of the principles expounded in Section 3 to the construction of some practical programs. We shall use a method and set of heuristics for constructing programs from specifications which were exposed in [13]. A similar approach was applied to classical (scalar) numerical algorithms in [2].

The following notation will be used in addition to the ones defined in Section 3:

- $VEC(n)$ stands for $VEC[REAL](n)$, the set of vectors of n real elements;
- $MTR(m, n)$ is the set of (m, n) real matrices;
- $P_l v$, where $v \in VEC(n)$ and $l \leq n$, is the projection of v on $VEC(l)$.

For a matrix $s \in MTR(m, n)$, if $i \leq m$ and $j \leq n$, we will consider line $s[i, *]$ and column $s[*, j]$ as vectors in $VEC(m)$ and $VEC(n)$ respectively.

5.1. *Triangular systems*

We saw in Section 3 a vector algorithm for matrix multiplication. Let us proceed with the inverse operation: solving linear systems. We first examine triangular systems. This will be a simple example of top-down synthesis of a numerical algorithm.

The first step in the design of the program (called *trisolv*) is to express it as a matrix algorithm (which could run on the virtual machine MAT):

in s: $MTR(n,n)$, b: $VEC(n)$; ***out*** x: $VEC(n)$;
(P) $\{1 \leq i \leq n \Rightarrow P_{i-1}s[*,i] = 0$ ***and*** $s[i,i] \neq 0\}$
trisolv
(Q) $\{sx = b$, i.e. $\sum_{k=1}^{n} s[*,k] * x[k] = b\}$

We must refine *trisolv* into a predicate transformer (on the vector machine EXT) from the precondition (P) to the postcondition (Q). Let us try twice the heuristic called 'uncoupling' [13], i.e. add an auxiliary vector variable y, and an integer one l, noticing that

$$(Q) \Leftrightarrow b = \sum s[*,k] * x[k] \Leftrightarrow (y + \sum_{k \leq n} s[*,k] * x[k] = b \textbf{ and } y = 0)$$
$$\Leftrightarrow (y + \sum_{k \leq l} s[*,k] * x[k] = b \textbf{ and } P_l y = 0)$$
$$\textbf{and } l = n.$$

So $(Q) \Leftrightarrow (I(l)$ ***and*** $l = n)$ if we set $I(l) =$ the first term of the ***and*** above. Here, $I(l)$ is a 'weakening' of the exit condition (Q) (which is $I(n)$). We notice that $I(0)$ can be trivially obtained. Thus a refinement of *trisolv*, using $I(l)$ as an invariant and $l = n$ as the goal (exit condition) will be:

var l: Integer;
$l := 0$; $y := b\{I(l)\}$
while $l < n$ ***do***
  $l := l + 1$;
  reestablish $I(l)$;
$\{l = n$ ***and*** $I(l)\}$

This program is correct (by construction): $I(l)$ being a loop invariant, it is true after the completion of the loop, and the exit condition $l = n$ is also true, hence $I(n)$. The statement *reestablish* is now (just as *trisolv* was, one step backwards) a specification for what is to be done.

Next step: develop *reestablish*. One must go from $I(l-1)$, i.e.

$$y + \sum_{k<l} s[*,k] * x[k] = b \textbf{ and } P_{l-1} y = 0$$

to $I(l)$, i.e.

$$y + \sum_{k<l} s[*,k] * x[k] = b - s[*,l] * x[l] \textbf{ and } P_l y = 0.$$

Without modifying b, which is part of the input, we must use the assignment $y := y - s[*, l] * x[l]$ after an $x[l]$ such that $P_l(y - s[*, l] * x[l]) = 0$ has been found. But $P_{l-1}s[*, l] = 0$ by hypothesis, and $P_{l-1}y = 0$ also. The equation thus becomes $y[l] - s[*, l] * x[l] = 0$, thence $x[l]$. The final version of the program is:

$l := 0$; $c := b$; $I(0)$
while $l < n$ **do**
| $l := l + 1$;
| {*reestablish* $I(l)$:}
| | $x[l] := y[*, l]/s[l, l]$
| | $y := y - s[*, l] * x[l]$

Starting from a matrix specification and aiming at the EXT vector target machine, we have just synthesized a program which must be, by construction, vectorizable.

5.2. *Vectorized Choleski*

We shall now introduce a more difficult algorithm, Choleski factorization: given a symmetric positive-definite matrix A, find a lower triangular S such that $SS^t = A$ (in view of the resolution in two easy steps, using e.g. the above program, of the linear system $Ax = b$). What follows is also valid for the LU factorization.

We again apply systematic top-down synthesis. Here are the successive steps. First the specification, expressed in terms of MAT objects:

in a: $MTR(n, n)$; **out** s: $MTR(n, n)$;
(R) {*symmetric*(a) **and** *positive-definite*(a)}
| *Choleski*
| $\{1 \le i \le n \Rightarrow P_{i-1}[*, i] = 0\}$
(S) | $\{A = SS^t$, i.e. $a = \sum_{k \le n} s[*, k] * s[*, k]\}$

As before, we uncouple (S), after introducing the auxiliary variable c of type $MTR(n, n)$:

$$(S) \Leftrightarrow ((c + \textstyle\sum_{k \le l} s[*, k] * s[*, k] = a \textbf{ and } P_l c = 0) \textbf{ and } l = n)$$

$$\Leftrightarrow \qquad (I(l) \textbf{ and } l = n).$$

The next refinement is, quite naturally:

$l := 0$; $c := a$; $\{I(0)\}$
while $l < n$ ***do***
| $l := l + 1$; $\{c + \sum_{k<l} = a$ ***and*** $P_{l-1}c = 0\}$
| *reestablish* $I(l)$;
| $\{c + \sum_{k<l} = a - s[*, l] * s[*, l]$ ***and*** $P_l c = 0\}$.

To *reestablish* $I(l)$, one must perform the assignment $c := c - s[*, l] * s[*, l]$ once an $s[*, l]$ such that $P_{l-1}s^l = 0$ and

$$P_l(c - s[*, l] * s[*, l]) = 0$$

has been found. As $P_{l-1}c = 0$, row l is the only one concerned, and must satisfy l-*column*$(c - s[*, l] * s[*, l]) = 0$, that is to say $c[l, *] - s[l, l] * s[*, l] = 0$, which implies ($l$ component)

$$c[l, l] = (s[l, l])^2.$$

Thence the two instructions for *reestablish* $I(l)$:

$$s[l, l] := sqrt(c[l, l])); \; s[*, l] := c[l, *]/s[l, l].$$

As c is symmetric (this fact is itself a loop invariant), $P_{l-1}c[*, l] = 0$ implies $P_{l-1}c[l, *] = 0$, therefore $P_{l-1}s[*, l] = 0$.

The final version will thus be:

$l := 0$; $c := a$;
while $l < n$ ***do***
| $l := l + 1$;
| $pivot := sqrt(c[l, l])$;
| $s[*, l] := c[l, *]/pivot$;
| $c := c - s[*, l] * s[*, l]$

A FORTRAN translation appears on Fig. 2 and 3. It exhibits some of the nice properties of programs resulting from top-down design (high-level built-in documentation, etc.) and the safety guaranteed by the systematic synthesis method.

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      SUBROUTINE
     "                        C H O V E C
     "                      (N, A, S, NDP)
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                              C
C                              PURPOSE:                                        C
C                                                                              C
C     FACTORIZATION OF A SYMMETRIC MATRIX, VECTORIZABLE VERSION.               C
C                                                                              C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C     INPUT
C
            INTEGER    N
C                                  Order of the matrix A
            REAL       A   ( 1 )
C                                  Array of the entries of A. Aij is at
C                                  the position ((J - 1)(2N - J) + 2I)/2
C                                  ("column-symmetric storage mode")
C     OUTPUT
C
            REAL       S   ( 1 )
C                                  Array of the entries of A. Aij is at
C                                  the position ((J - 1)(2N - J) + 2I)/2
C                                  On exit, if NDP = N, A = S tr(S)RT
            INTEGER    NDP
C                                  Number of columns actually taken into
C                                  account during the factorization.
C                                  If NDP < N, a non-positive radix ap-
C                                  peared in the treatment of column
C                                  NDP + 1
C------------------------------------------------------------------------------
C
C     LOCAL VARIABLES:
C
            INTEGER L, NNP1S2, ADRLL, ADRJL, ADRJJ, I, J, LP1
            REAL PIVOT, MUL, RADIC
C
C     ARITHMETIC FUNCTION:
C
            INTEGER ADDRESS
            ADRESS(I, J) = ((J - 1)*(2*N - J) + 2*I)/2
C
```

Fig. 2. Head of the vectorizable Choleski program (FORTRAN).

6. Conclusion

The field of numerical and scientific programming, although the oldest and one of the best established among the application domains of computers, has shown strong resistance to the practical implementation of software research and advances in programming methodology. With the

```
C-----------------------------------------------------------------------------
C
           NDP = 0
C      i <--- 0 ;
           L = 0
C      C <--- A ;
           NNP1S2 = (N*(N + 1))/2
           DO 1 I = 1, NNP1S2
1               S(I) = A(I)
C               -- The array S contains both C and A.
C      while i < n do
2      IF (L .GE. N) GOTO 7
C          i <--- i + 1 ;
               L = L + 1
               ADRLL = ADRESS(L, L)
C          pivot <--- sqrt(C11) ;
               RADIC = S(ADRLL)
               IF (RADIC .LE. 0.) GOTO 7
C                  -- Exception if A is not positive definite
               PIVOT = SQRT(RADIC)
               NDP = L
C          S1 <--- C1/pivot ;
               DO 3 I = L, N
3                  S(ADRLL + I - L) = S(ADRLL + I - L)/PIVOT
C          C <--- C - S1 * S1 ;
               LP1 = L + 1
               IF (LP1 .EQ. N) GOTO 6
               DO 5 J = LP1, N
                  ADRJJ = ADRESS(J, J)
                  ADRJL = ADRESS(J, L)
                  MUL = S(ADRJL)
                  DO 4 I = J, N
                     S(ADRJJ+I-J) = S(ADRJJ+I-J) - MUL*S(ADRJL+I-J)
C                    -- This loop is the only vectorizable one
4                    CONTINUE
5                  CONTINUE
6              CONTINUE
           GOTO 2
7      RETURN
       END
```

Fig. 3. Body of the Choleski program.

popularization of new 'number-crunching' machines, there is again a strong temptation to go back to low-level, machine-dependent, programming techniques, and to dismiss any attempts at better software engineering as incompatible with the efficient use of these very fast computers. We hope to have shown that such an attitude has no justification, and that systematic methods can be applied for the rational and efficient use of this new technology.

References

[1] J.R. Abrial, S.A. Schuman and B. Meyer, Specification language, in: Proceedings Summer School on Program Construction, Belfast (September 1979).

[2] A. Bossavit and B. Meyer, On the constructive approach to programming: the case for partial Choleski factorization (a tool for static condensation), in: Vichnevetsky and Stepleman (Eds.), Advances in Computer Methods for Partial Differential Equations III (IMACS, 1979).

[3] Cray-1 Computer System, FORTRAN (CFT) Reference Manual, Cray Document No. 2240009, Version E (1981).

[4] M. Dungworth, The Cray 1 computer system, in: Infotech State of the Art Report on Supercomputers, Volume 2: Invited papers (Maidenhead, 1979) pp. 51–76.

[5] P.M. Flanders, FORTRAN extensioms for a highly parallel processor, in: Infotech State of the Art Report on Supercomputers, Volume 2: Invited Papers (Maidenhead, 1979) pp. 117–134.

[6] L. Higbie, Vectorization and conversion of FORTRAN programs for the Cray-1 (CFT) compiler, Cray Document No. 2240207 (June 1979).

[7] Infotech State of the Art Report on Supercomputers, Volume 1: Total Systems Issues; Volume 2: Invited papers (Maidenhead, 1979).

[8] E.W. Kozdrowicki and D.J. Theis, Second-generation of vector supercomputers, Computer (IEEE), Special Section on Supersystems for the 80's 13 (11) (1980) 71–83.

[9] D.J. Kuck, Languages and compilers for parallel and pipeline machines, in: CREST Conference on Design of Numerical Algorithms for Parallel Processing, Bergamo, Italy (June 1981).

[10] D.J. Kuck, Automatic program restructuring for high-speed computation, in: W. Händler (Ed.), CONPAR 81, Nürnberg, June 1981, Lecture Notes in Computer Science 111 (Springer, Berlin, 1981) pp. 66–84.

[11] B. Meyer and C. Baudoin, Méthodes de programmation (Eyrolles, Paris, 1978).

[12] B. Meyer, Un calculateur vectoriel: Le Cray-1 et sa programmation, EDF Report HI/3452-01, Atelier logiciel No. 24 (May 1980).

[13] B. Meyer, A basis for the constructive approach to programming, in: S.H. Lavington (Ed.), Information Processing 80 (North-Holland, Amsterdam, 1980).

[14] R.H. Perrott, Parallel languages, in: Infotech State of the Art Report on Supercomputers, Volume 1: Total Systems Issues (Maidenhead, 1979) pp. 117–149.

[15] R.H. Perrott, A standard for supercomputer languages, in: Infotech State of the Art Report on Supercomputers, Volume 2: Invited Papers (Maidenhead, 1979) pp. 291–308.

[16] R.H. Perrott, A language for array and vector processors, TOPLAS (Transactions on Programming Languages and Systems, ACM) 1 (2) (1979) 177–195.

Algorithmic Languages, de Bakker/van Vliet (eds.)

Formal Language Definitions Can Be Made Practical

Paul Klint

Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

If some formal method is used to define a programming language, the problem arises that individuals with different backgrounds and intentions have to learn a notation and definition method they are unfamiliar with. The various uses of formal definitions are summarized in this paper and an improved method for operational language definitions is presented. This method aims at language descriptions that are understandable and useful for both designer, implementor and user of a defined language. The method has been used in the definition of the SUMMER programming language. Various examples of that definition are given and the method as a whole is assessed.

> "... *The metalanguage of a formal definition must not become a language known to only the priests of the cult. Tempering science with magic is a sure way to return to the Dark Ages.*" [6]

1. The Problem

Programming languages are being designed using pre-scientific methods. Of course, there is no substitute for experience, taste, style and intuition but a scientific design methodology to support them is lacking. Methods for describing programming languages are somewhat more developed, but most definitions are either ambiguous and inaccurate, or excessively formal and unreadable. In general, a language definition method should:

(1) help the language *designer* by giving insight in the language he or she is designing and by exposing interactions that might exist between language features. The definition should at the same time be a pilot implementation of the defined language or it should at least be convertible into one. It is assumed here, that design and definition can best be carried out simultaneously.

(2) help the language *implementor* by providing him with an unam-

biguous and complete definition that is capable of "executing" small programs in cases where the implementor is in doubt about the meaning of a particular language feature.

(3) help the *user* by providing him with a precise definition in a language he is not too *un*familiar with.

These three goals impose different and to a certain extent contradictory requirements on the definition method to be used. In particular, it seems difficult to combine precision and readability in one method, since a precise definition has to use some formalism to which the reader has to be initiated and such a definition will have a tendency to become long and unreadable. This paper reports on an experiment with a language definition method that may be considered as a first step in satisfying the above requirements.

The *defined* language is SUMMER [3, 4] an object-oriented string processing language. The definition method is similar in spirit to the SECD method [5], i.e. it is an operational language definition method which uses recursive functions and syntactic recognition functions to define a finite state machine that associates semantic actions with all constructs in the grammar of the language. In the method presented in this paper readability has been considerably enhanced by using a few imperative constructs and by introducing a very concise notation for parsing and decomposing the source-text of programs in the defined language. SUMMER, extended with such parsing and decomposing operations, is used as *defining language.* The definition is hence circular (see Sections 2.1 and 3).

A complete description of the definition method can be found in [4]. The next section gives only a birds-eye view of the description method and shows some illustrative examples from the SUMMER definition. In Section 3 the method as a whole and its application to SUMMER are assessed.

2. The Method

2.1. *Introduction*

An *evaluation process* or *interpreter* (with the name "eval") will be defined that takes an arbitrary source text ("the source program") as input and either computes the result of the execution of that program (if it is a legal program in the defined language) or detects a syntactic or semantic error. The evaluation process operates directly on the text of the source

program and the process as a whole may be viewed as performing a series of string transformations on that text. During this process a global *environment* may be inspected or updated. An environment is a mapping from identifiers in the source program to their actual values during the evaluation process. Environments are used to describe concepts such as variables, assignment and scope rules.

A fundamental question arises here: in which language do we write the definition? Several choices can be made, such as the formalism used in denotational semantics ([1], this boils down to mathematical notation for recursive functions and domains) or the Vienna Definition Language ([8], a programming language designed for the manipulation of trees). This is not the right place to discuss the merits of these formalisms, but none has the desired combination of properties as described in the previous paragraph. Instead of designing yet another definition language, the defined language itself (this is SUMMER in the examples given in this paper) will be used as definition language. This choice has the obvious disadvantage that the definition is circular, but it has the practical advantage that readers who have only a moderate familiarity with the defined language will be able to read the definition without great difficulty. An extensive discussion of circular language definitions can be found in [7]. It should be emphasized that there is no *fundamental* reason to make the definition circular. The definition method described here would also work if, for example, ALGOL 68 was used as defining language. In any case, it is essential that the defining language has powerful string operations and allows the creation of data structures (of dynamically determined sizes). This requirement makes, for example, PASCAL less suited as defining language. Choosing SUMMER as defining language gave us the opportunity to investigate the suitability of that language in the area of language definition (see Section 3).

In the following sections the definition method and an example of its application (in the SUMMER definition) are described simultaneously. In Section 2.2 some aspects of the use of SUMMER as a metalanguage are described. The definition method can be subdivided in the definition of semantic domains (Section 2.3) and of the evaluation process (Section 2.4). Some more detailed examples from the SUMMER definition are given in Section 2.5.

2.2. *SUMMER as metalanguage*

This paragraph focuses on some aspects of SUMMER that are used in the

formal definition. Most of these constructs have some similarity with constructs in, for instance, PASCAL and are assumed to be self-explanatory. Only less obvious constructs that are essential for the understanding of the definition are mentioned here.

SUMMER is an object-oriented language with pointer semantics. This means that an object can be modified by assignment and that such modifications are visible through all access paths to that object. For example,

$$s := stack(10)$$

assigns a *stack* object of size 10 to the variable *s,* and

$$s\,.\,push(v)$$

pushes the value of v on the stack s. As a side-effect the stack s is modified such that subsequent operations on s may perceive the effect of that modification. In the formal definition this is relevant for the concepts "state" and "environment", which are modified in this way.

The language is dynamically typed, i.e. the type of variables is not fixed statically (as in PASCAL) but is only determined during the execution of the program (as in LISP or SNOBOL4). Moreover, generic operations on data structures are allowed. If an operation is defined on several data types, then the procedure to be executed when that operation occurs is determined by the type of the (left) operand of that operation.

Control structures and data structures are self-explanatory except possibly *arrays* and *for-statements.*

Arrays are vectors of values, indexed by $0, \ldots, N-1$, where N is the number of elements in the array. If A is an array then the operation $A\,.\,size$ will yield the number of elements in the array. A new array is created by

$$[V_0, \ldots, V_{N-1}]$$

or

$$array(N, V).$$

In the former case, an array of size N is created and initialized to the values $V_0, \ldots, V_{N-1}$. In the latter case, an array of size N is created and all elements are initialized to the value V. Array denotations are also allowed as left hand side of assignments. This provides a convenient notation for multiple assignments. For example,

$$[x, y, z] := [10, 20, 30]$$

is completely equivalent with

$$x := 10;\ y := 20;\ z := 30$$

and, more generally,

$$[x_0, \ldots, x_k] := a$$

is equivalent with

$$x_0 := a[0]; \ldots ; x_k := a[k].$$

The general form of a for-statement is:

for *V* **in** *G* **do** *S* **od**

where *V* is a variable, *G* is an expression that has as value an object capable of generating a sequence of values VAL_i and where *S* is an arbitrary statement. For each VAL_i the assignment $V := VAL_i$ is performed and *S* is evaluated. In this paper, the expression *G* will be used in two forms: the value of *G* is either an array (in which case consecutive array elements are generated) or *G* is an array on which the operation *index* has been performed (in which case all indices of consecutive array elements are generated). For example, in

a := [144, 13, 7];
for *x* **in** *a* **do** *print*(*x*) **od**

an array object is assigned to the variable *a* and the values *144, 13* and *7* will be printed, while

for *i* **in** *a*. *index* **do** *print*(*i*) **od**

will print the values *0, 1* and *2*. Further examples of for-statements will be found in the following paragraphs.

2.3. *Semantic domains*

A semantic domain is a set, whose elements either describe a primitive notion in the defined language (like "variable" or "procedure declaration") or have some common properties as far as the language definition is concerned. The relationship between these domains is given by a series of domain equations.

In the remainder of this paragraph the domains in the SUMMER definition

are briefly described. The abstract properties of these domains are given in [4]. Here, they are only introduced informally. First, the domain equations are given. Next, the meaning of each domain is described.

The relationship between the domains BASIC-VALUES, DENOTABLE-VALUES, STORABLE-VALUES, ENVIRONMENT, LOCATIONS, STATE, PROC, CLASS and INSTANCE is as follows

BASIC-VALUES = STRING ∪ INTEGER ∪ UNDEFINED
DENOTABLE-VALUES = LOCATIONS ∪ INSTANCE ∪ PROC ∪ CLASS ∪ BASIC-VALUES
STORABLE-VALUES = INSTANCE ∪ BASIC-VALUES
ENVIRONMENT = ID → DENOTABLE-VALUES
STATE = LOCATIONS → (STORABLE-VALUES ∪ {unused})
PROC = PROC-DECL × ENVIRONMENT
CLASS = ID × CLASS-DECL
INSTANCE = ID × CLASS-DECL × ENVIRONMENT

Here, ID, PROC-DECL and CLASS-DECL are the sets of string values that can be derived from the syntactic notions ⟨*identifier*⟩, ⟨*procedure-declaration*⟩ and ⟨class-declaration⟩ in the SUMMER grammar. BASIC-VALUES is the domain of primitive values in the language. DENOTABLE-VALUES is the domain of values which can be manipulated by the evaluation process. STORABLE-VALUES is the domain of values which can be assigned to variables in the source program. The domain LOCATIONS is used to model the notion "address of a cell capable of containing a value". Inspection of the contents of a location does not affect the contents of that location itself or of any other location. Modification of the contents of a location does not affect the contents of any other location. STATE is the domain that consists of functions that map locations on actual values or unused.

PROC is the domain of procedures. Each element of this domain describes a procedure declaration and contains a literal copy of the text of the procedure declaration itself and an environment that reflects all names and values available at the point of declaration.

CLASS is the domain of classes. Each element of this domain describes one class declaration and contains the name of the class and a literal copy of the text of the class declaration. INSTANCE is the domain of class instances. All values that are created by a SUMMER program are instances

of some class. An instance consists of the name of the class to which it belongs, the literal text of the declaration of that class and an environment that has to be used to inspect or update components from the instance. Operations are defined on elements in PROC, CLASS and INSTANCE to manipulate the components of an element in these domains. For completeness, these domains are mentioned here, but they will not be used in the remainder of this paper.

STRING, INTEGER and UNDEFINED are the domains modeling the values and operations for the built-in types *string, integer* and *undefined* respectively. UNDEFINED is the domain consisting of undefined values. All variables are initialized to an undefined value. Operations are defined on elements in STRING, INTEGER and UNDEFINED that model the primitive operations on the data types *string, integer* and *undefined.*

ENVIRONMENT is the domain of environments. Environments administrate the binding between names and values and the introduction of new scopes (i.e. ranges in the program where names may be declared). The operations defined on environments modify, in general, the environment to which they are applied.

The definitions given in following sections are centered around operations on elements of these semantic domains, but we will see relatively few of them in the examples. Operations will only be explained when they occur in an example.

2.4. *Evaluation process*

An extended form of BNF notation is used to describe the syntax of the defined language. The extensions aim at providing a concise notation for the description of repeated or optional syntactic notions. A syntactic notion suffixed with "+" means one or more repetitions of that notion. A notion suffixed with "*" stands for zero or more repetitions of that notion. The notation

{notion separator} replicator

i.e. a *notion* followed by a *separator* enclosed in braces followed by a *replicator,* is used to describe a list of notions separated by the given separator. A replicator is either "+" or "*". The replicator "+" indicates that the list consists of one or more notions. The list begins and ends with a notion. The replicator "*" indicates that the list consists of zero or more notions.

An optional syntactic notion is indicated by enclosing that notion in square brackets, e.g. "[*notion*]". The terminal symbols of the grammar are either enclosed in single quotes (for example: ',' or ':=') or written in upper case letters if the terminal symbol consists solely of letters (for example: *IF* may be used to denote the terminal symbol **if**). Where necessary, parentheses are used for grouping.

Some parts of a syntax rule may be labeled with a ⟨*tag*⟩; their meaning will become clear below.

The evaluation process is described in SUMMER extended with *parse expressions*[1] of the form

'{{' ⟨*identifier*⟩ '==' ⟨*syntax-rule*⟩ '}}'

which are used as a very concise notation for parsing and extracting information from the text of the source program. A parse expression succeeds if the identifier at the left hand side of the '==' sign has a string as value and if this string is of the form described by the ⟨*syntax-rule*⟩ at the right hand side of the '==' sign. All ⟨*tag*⟩s occurring in the ⟨*syntax-rule*⟩ should have been declared as variables in the program containing the parse expression, in this case the evaluation process. Substrings of the parsed text are assigned to these variables. If the recognized part of the text is a list or repetition, then an array of string values is assigned to the variable corresponding with the *tag*. Consider, for example, the following program fragment:

```
if {{e== WHILE t:⟨test⟩ DO b: ⟨body⟩ OD}}
then
    put ('e is a while expression')
fi
```

The parse expression will succeed if *e* has the form of a while expression; the literal text of the ⟨*test*⟩ is then assigned to variable *t* and the text of the ⟨*body*⟩ is assigned to variable *b*. Repetition occurs in

```
if {{e== VAR list:⟨test⟩ DO b:⟨body⟩ OD}}
```

[1] There is no *fundamental* reason to introduce this language extension. However, the disadvantage of introducing such an ad-hoc extension is more than compensated by the fact that we use a notation which is sufficiently similar to BNF notation to be almost self-explanatory. The effect of introducing a language extension as proposed here is interesting in its own right but falls outside the scope of the current discussion.

```
then
    put('e is a variable declaration containing:');
    for l in list do put (l) od
fi
```

The parse expression succeeds if *e* has the form of a "variable declaration" (i.e. the keyword **var** followed by a list of ⟨*identifier*⟩s separated by commas) and in that case an array of string values corresponding to the ⟨*identifier*⟩s occurring in the declaration is assigned to the variable *list,* which is printed subsequently.

Parse expressions may be used as test in if statements or may stand on their own. In the latter case, the string to be parsed *has* to be of the form described by the parse expression. In this way, parse expressions can be used to decompose a string with a known form into substrings.

In the case of the SUMMER definition, the overall structure of the evaluation process is:

```
var E;
var S;
var varinit;
proc ERROR
    ...;
proc eval(e)
(var value, signal, ...;
 if {{e == ⟨program-declaration⟩}}
 then
     ...
     return([value, signal])
 fi;
 if {{e == ⟨variable-declaration⟩}}
 then
     ...
     return([value, signal])
 fi;
 ...
 if {{e == ⟨empty⟩}}
 then
     ...
     return([value, signal])
```

```
    fi;
    ERROR
);
```

The variable *E* has as value the current environment and *S* has as value the current state. The variable *varinit* has as value a string consisting of the text of all ⟨*variable-initialization*⟩s in the current ⟨*block*⟩.

The procedure *ERROR* is called when a syntactic or semantic error is detected during evaluation. In that case, the whole evaluation process is aborted immediately. The main defining procedure is *eval,* which selects an appropriate case depending on the syntactic form of its argument *e*. Some examples of these various cases will be given in Section 2.5. Note that each of these cases involves a complete syntactic analysis of the string *e*. The evaluation process is initiated by creating an initial, empty environment *E* and by calling *eval* with the text of the source program as argument. If the evaluation process is not terminated prematurely (by the detection of a semantic error) the result of the evaluation of the source program can be obtained by inspecting the resulting environment *E*. Note how syntactically incorrect programs are intercepted in *eval* by *ERROR,* which is called if none of the listed cases applies.

The procedure *eval* delivers as result an *array* of the form [*value, signal*], where *value* is the actual result of the procedure and *signal* is a success/fail flag that indicates how *value* should be interpreted. SUMMER uses a success-directed evaluation scheme: an expression can either *fail* or *succeed.* These success/fail signals are used by language constructs like ⟨*if-expression*⟩ and ⟨*while-expression*⟩ to determine the flow-of-control. The *signal* delivered by *eval* is used to model this evaluation mechanism. This *signal* may have the following values:

N: evaluation terminated normally.
F: evaluation failed.
NR: normal return; a ⟨*return-expression*⟩ was encountered during evaluation.
FR: failure return; a failure return was encountered during evaluation.

The signal is tested after each (recursive) invocation of *eval.* In most cases eval performs an immediate return if the signal is not equal to *N* after the evaluation of a subexpression. Exceptions are cases such as ⟨*if-expression*⟩ and ⟨*return-expression*⟩ in which the signal is used to determine how evaluation should proceed. This organization has the effect that aborting the evaluation of the "current" expression, which is necessary if failure

occurs in a deeply nested subexpression, can be achieved by passing a signal upwards until it reaches an incarnation of *eval* that can take appropriate measures. The difference between *F* and *FR* lies in the language constructs that handle these cases. For example, consider ⟨*if-expression*⟩s. An *F* signal generated in the ⟨*test*⟩ part of an ⟨*if-expression*⟩ can be treated by the semantic rule associated with ⟨*if-expression*⟩s. But an *FR* signal generated during the evaluation of the ⟨*test*⟩ can only be treated by the semantic rule associated with the invocation of the procedure in which the ⟨*if-expression*⟩ occurs. In general, the signals *NR* and *FR* are only *generated* by return-expressions and are only *handled* by the semantic rules associated with procedure calls. The latter rules turn *NR* into *N* and *FR* into *F* before the evaluation process is resumed at the point where it left off to perform the (now completed) procedure call. All other semantic rules return immediately when an *NR* or *FR* signal occurs.

Note that the [*value, signal*] artifact is induced by the specific form of expression evaluation in SUMMER and has nothing to do with the definition method itself. We have just chosen one particular way to describe a form of goto statement.

2.5. *Some examples*

2.5.1. *If expressions*

⟨*if-expression*⟩s correspond to the if-then-else statement found in most programming languages. If evaluation of the ⟨*test*⟩ immediately contained in the ⟨*if-expression*⟩ terminates successfully, the ⟨*block*⟩ following **then** is evaluated. Otherwise, the successive ⟨*test*⟩s following subsequent *elif*s are evaluated until one such evaluation terminates successfully (in which case the following ⟨*block*⟩ is evaluated) or the list is exhausted. In the latter case, the ⟨*if-expression*⟩ may contain an **else** and then the ⟨*block*⟩ following that **else** is evaluated. The formal definition is:

```
1 if {{e == IF t: ⟨test⟩ THEN b: ⟨block⟩
2             elifpart: (ELIF ⟨test⟩ THEN ⟨block⟩)*
3             elsepart: [ELSE ⟨block⟩] FI}}
4 then
5     [v, sig] := eval(t);
6     if sig = N then return(eval(b))
7     elif sig ≠ F then return([v, sig])
8     else
```

```
 9          for ei in elifpart
10          do {{ei == ELIF t: ⟨test⟩ THEN b: ⟨block⟩}};
11             [v, sig] := eval(t);
12             if sig = N then return(eval(b))
13             elif sig ≠ F then return([v, sig]) fi
14          od;
15          if {{elsepart == ELSE b: ⟨block⟩}}
16          then
17             return(eval(b))
18          else
19             return([a_undefined, N])
20          fi
21       fi
22 fi;
```

The parse expression in lines *1–3* decomposes the string value of *e* in several parts. In line *5* the ⟨*test*⟩ of the ⟨*if-expression*⟩ is evaluated. Note how the occurrence of non-standard (i.e. *sig* = *NR* or *sig* = *FR*) signals terminates the evaluation of the ⟨*if-expression*⟩ (lines *7, 13*). This is particularly relevant for the evaluation of the ⟨*test*⟩ part. SUMMER allows the occurrence of a return statement in a ⟨*test*⟩. This is reflected in the above definition.

For a better understanding of the above definition, it may be useful to note that parts of the source program are parsed *repeatedly* during *one* evaluation of a given ⟨*if-expression*⟩. For example, the ⟨*block*⟩ following an **elif** is parsed both in lines *2* and *10*. (This explains, by the way, why the parse expression in line *10* needs not be contained in an if statement, see Section 2.4.) In general, the source text of the ⟨*if-expression*⟩ is parsed *each* time that it is evaluated.

2.5.2. *Variable declarations*

A ⟨*variable-declaration*⟩ introduces in the current environment a series of new variables, i.e. names of locations whose contents may be inspected and/or modified. The declaration may contain ⟨*expression*⟩s whose value is to be used for the initialization of the declared variables. First, these initializing expressions are evaluated. Next, the ⟨*expression*⟩s following the ⟨*variable-declaration*⟩s are evaluated. In the formal definition this is described by appending all variable initializations in the current ⟨*block*⟩ to the variable *varinit* and by evaluating the string value of that variable

before the evaluation of the subsequent ⟨*expression*⟩s in the ⟨*block*⟩. The formal definition of ⟨*variable-declaration*⟩s is:

```
 1 if {{e == VAR vi : {⟨variable-initialization⟩ ',' } + ';' }}
 2 then
 3     for v in vi
 4     do if {{v == x : ⟨identifier⟩ ':=' ⟨expression⟩}} then
 5           varinit := varinit ‖v‖ ';' ;
 6           E . bind(x, S . extend(a_undefined));
 7        else
 8           {{v == x : ⟨identifier⟩}};
 9           E . bind(x, S . extend(a_undefined))
10        fi
11     od;
12     return([a_undefined, N])
13 fi;
```

In line *1, e* is decomposed into an array of strings which have the form of a ⟨*variable-initialization*⟩. These string values are considered in succession in the for loop in lines *3–11*. If the ⟨*variable-initialization*⟩ contains an initializing expression, that expression is appended to *varinit* (line *5*) using the string concatenation operator "‖". In both cases, the state *S* is *extend*ed with a location containing an undefined value, and that new location is *bound,* in the current environment *E,* to the identifier being declared. Note that, in line *8, v* is known to have the form of an ⟨*identifier*⟩.

2.5.3. *Blocks*

A ⟨*block*⟩ introduces a new scope to be used for the declaration of new variables and constants. It consists of a (perhaps empty) list of declarations followed by a sequence of expressions separated by semicolons. A ⟨*block*⟩ is evaluated as follows:

(1) Evaluate all declarations.

(2) Evaluate all variable-initializations resulting from the evaluation of the declarations.

(3) Evaluate the sequence of expressions in the ⟨*block*⟩. (Note that SUMMER forbids the failure of an expression inside a sequence of expressions. Only the last expression in a sequence is allowed to fail; this failure is passed upwards to enclosing language constructs.)

The formal definition is:

```
1 if {{e == dlist: ⟨variable-declaration⟩*
2             elist: {[⟨expression⟩] ';' }*}}
3 then
4     var E1, varinit1;
5     E1 := E;
6     E.new_inner_scope;
7     varinit1 := varinit;
8     varinit := '' ;
9     for d in dlist
10    do [v,sig] := eval(d);
11       if sig ≠ N then ERROR fi
12    od;
13    [v,sig] := eval(varinit);
14    varinit := varinit1;
15    if sig ≠ N then E := E1; return([v,sig]) fi;
16    for i in elist.index
17    do
18       [v,sig] := eval(elist[i]);
19       case sig of
20       N:,
21       F: if i ≠ elist.size - 1 then ERROR fi,
22       NR: FR: (E := E1; return([v,sig]))
23       esac
24    od;
25    E := E1;
26    return([v,sig])
27 fi;
```

In lines *5–8* local copies are made of *E* and *varinit* and new values are assigned to them. In lines *9–13* the list of ⟨*variable-declaration*⟩s in the ⟨*block*⟩ and the resulting ⟨*variable-initialization*⟩s are evaluated. In lines *16–24* the list of ⟨*expression*⟩s in the ⟨*block*⟩ are evaluated. Note how failure of an expression in the middle of the list is treated (line *21*, see above).

3. Assessment

The formal language definition presented in the previous section will now be assessed. It is tempting to try to get statements like:

> *"Users can answer* 87% *of their questions on language issues within five minutes if they have access to a formal language definition of the kind described in this article."*

or

> "35% *of all run-time errors in user programs are directly related to anomalies in the language definition*".

In the absence of such results and with the methods to obtain them lacking, we have to live with qualitative and more or less speculative observations.

A rough indication for the *conciseness* of the definition can be obtained by comparing various sizes as they apply to the SUMMER definition:

formal definition	20 pages
reference manual	100 pages
implementation	200 pages

These figures show that the implementation is ten times larger than the formal definition. This is not surprising, since the implementation has to be efficient while the formal definition does not have to be. In this light the "a-language-is-defined-by-its-implementation" approach can be rephrased as: "*if* a language is defined by its implementation, *then* that implementation had better be small".

The definition is *precise* and *complete,* in the sense that *all* semantic operations associated with a particular language construct *have* to be specified to allow the construction of an *executable* version of the definition. The number of *operational details,* i.e. details in the definition which stem from the chosen definition method and have no inherent meaning in the defined language, is surprisingly small. This is a consequence of the choice of the defining language (which should have powerful data types and string manipulation operations) and the choice of high-level environment manipulation primitives which correspond directly to operations in the defined language and which are not (yet) perverted by implementational details. SUMMER extended with parse expressions seems a quite reasonable vehicle for language definition. It is, however, not possible to make continuation-style (see [1]) definitions, since higher-order functions are lacking.

It is difficult to give an objective judgement on the *readability* of the definition, but we have observed that only a moderate effort (a few days) is required on the part of a programmer without any training in formal semantics and without any previous exposure to the language to learn SUMMER using only the (annotated) formal definition.

The advantages and disadvantages of the formal definition for designer, implementor and user will now be discussed in some detail.

The advantages for the *designer* are:

(1) Anomalies in the design are magnified. It is a general rule that ill-formed entities can only be described by ill-formed descriptions or by descriptions which list many exceptional cases. It is easier to locate such exceptions or anomalies in a concise formal definition than in an ambiguous natural language definition or in a bulky implementation. In the SUMMER definition, for example, a very specific operation on environments is needed ("partial-state-copy") to accommodate the definition of just one language feature ("try-expression"). It turned out that a slight modification of that feature would at the same time simplify the definition and improve the feature.

(2) Exhaustive enumeration of language features. A formal definition method forces the designer to enumerate all language features in the same framework and this may help him to find omissions in the design.

(3) Interactions between language features can be studied. In the SUMMER definition, for example, the designer is forced to decide what happens when a ⟨*return-expression*⟩ is evaluated during the evaluation of any other expression. There is, however, no guarantee that all interactions can be found, since the formal definition may still contain hidden interactions between language features. The use of auxiliary functions in the definition is an aid in making interactions explicit. One may even apply techniques such as calling graph analysis and data flow analysis to the definition to discover clusters of interacting features and to establish certain properties of the definition.

(4) An executable formal definition can be tested and used. This may help eliminate clerical and gross errors from the definition. An executable definition allows the designer to play with (toy) programs written in the language he is designing. Here is, however, a problem with circular definitions: some implementation of the defined language has to exist before the definition itself can be made executable.

Disadvantages for the *designer* are:

(1) A considerable effort is required to construct a formal definition.

(2) A general problem is that there are no canned, satisfactory definition methods available and that the designer has to begin with either creating a new method or adapting and extending an existing one.

Advantages for the *implementor* are:

(1) Unambiguous language definition.

(2) The implementor may be in doubt as to the meaning of a certain combination of features. Such cases can be executed both by the implementation and by the definition and the results can be compared.

Disadvantages for the *implementor* are:

(1) The implementor must be familiar with the definition method or become acquainted with it. This is only a minor effort if one compares it with the total effort required to implement the language.

(2) It is non-trivial to derive an implementation strategy from the language definition. This is a problem shared by all "abstract" language definitions, in which no attempt is made to use primitives in the definition with a direct counterpart in an implementation. This leads to the conclusion that such abstract definitions should be accompanied by an "annotation for implementors", which states where well-known implementation techniques can be used and where certain optimizations are possible.

Advantages for the *user* are:

(1) Unambiguous and concise language definition.

(2) The user is used to reading programs and the formal definition can be read as such. In the case of a circular definition, the formal definition may be considered as a very informative example program.

Disadvantages for the *user* are:

(1) The user must be exposed to the definition method.

(2) A formal definition is harder to read than a "natural language" definition.

(3) In the case of the SUMMER definition, the circularity may be confusing for the naive user.

In retrospect, it seems justified to conclude that the method presented in this paper is a first step in satisfying the requirements given in Section 1. However, many problems remain to be investigated. Does the given method lend itself to mathematical analysis? How can the "complexity" of a language be derived from its definition? Is it possible to "optimize" the executable version of definitions? (Attempts in this direction can be found in [2].) What is the relationship between this definition method and extensible languages? Answers to these questions will provide more insight in the structure of programming languages and the methods for defining them.

Acknowledgement

J. Heering, H.J. Sint and A.H. Veen made useful comments on various drafts of this paper. Parts of it were discussed with L.J.M. Geurts, F.E.J. Kruseman Aretz and L.G.L.T. Meertens. I am grateful for their support.

References

[1] M.J.C. Gordon, The Denotational Description of Programming Languages (Springer, Berlin, 1979).
[2] N.D. Jones, Semantics-directed Compiler Generation (Springer, Berlin, 1980).
[3] P. Klint, An overview of the SUMMER programming language, in: Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages (Jan. 1980) pp. 47–55.
[4] P. Klint, SUMMER Reference Manual, Mathematical Centre, to appear.
[5] P.J. Landin, The mechanical evaluation of expressions, Comput. J. 6 (1964) 308–320.
[6] M. Marcotty, H.F. Ledgard and G.V. Bochmann, A sampler of formal definitions, Comput. Surveys 8 (1976) 191–276.
[7] J.C. Reynolds, Definitional interpreters for higher-order languages, Proceedings ACM Annual Conference (Aug. 1972) pp. 717–740.
[8] P. Wegner, The Vienna Definition Language, Comput. Surveys 4 (1972) 5–63.

Algorithmic Languages, de Bakker/van Vliet (eds.)

Invited Address

Is Computer Science Based on the Wrong Fundamental Concept of 'Program'? An Extended Concept

John Backus

IBM Research Laboratory, 5600 Cottle Road, San Jose, CA 95193, U.S.A.

1. Introduction

1.1. *The Von Neumann concept of 'program'*

Ever since John von Neumann and others proposed the machine we call the Von Neumann computer, programmers have been writing 'Von Neumann style' programs. Originally these programs were written in machine language, then in assembly language, then in FORTRAN, and then in a great variety of so-called higher level languages. The units of action specified by a program element grew as these languages evolved, but all of these programs were primarily concerned with two things: (1) the transmission of input and output between the 'store' and the outside world, and (2) the transformation of the store from its state at input to some new state in which the desired output was available. (Of course, the concept of the 'store' also evolved from a device to an abstract entity comprising a set of 'cells' each with a 'name' and 'contents'.) Additionally these languages allowed one to make various assertions or declarations about the contents of the store, for example, that a certain cell contains an integer.

The variety and dynamic nature of input–output operations in the great assortment of Von Neumann languages make it difficult to include these operations in a uniform concept of 'program' that is common to all these languages over the past 30 years. So, following the example of ALGOL 60, we shall not attempt to do so.

Having dismissed the question of input–output, we are left with what may be considered the most fundamental concept of 'program': a 'program' is a mapping of some domain of 'stores' into itself. Each store in the domain of a machine language is a set of pairs, each pair being the

number of a cell and the contents of that cell. In assembly language the cell names of a store can be symbols; in higher level languages the cell names can be more complicated, and some languages permit stores whose 'cells' can hold 'contents' of more than one 'word'. But in all cases a 'store' is an association between 'names' and 'objects', each 'name' denoting a 'cell' in the store having a certain 'object' as its 'contents'.

Thus, neglecting input-output behavior, when any Von Neumann 'program' is given a store s in its domain, it will 'execute' and either the execution will go on forever or it will stop and yield some new store s' in its domain (it may happen that $s = s'$).

At this point we must confess that our simple notion of 'program' as a mapping of 'stores' into 'stores' differs from various precise notions to be found in works on denotational semantics ([18], e.g. [16,19, 20]), even though our simple notion reflects the spirit of these precise ones. To explain the details of *how* a program achieves a store-to-store mapping, or to explain dynamic storage reallocation, scope rules for variables, GOTOs, side effects, error stops, or other such issues, 'programs' are assigned mappings in denotational semantics that are more complex than store-to-store mappings. These mappings often involve 'environments' and 'continuations' as well as 'stores'.

In spite of the detailed explanations and language complications that force denotational semantics to represent 'programs' as more complex mappings, I submit that the single concept that is closest to our intuitive understanding and to all the various detailed concepts embodied in different Von Neumann languages is that 'programs' represent mappings of 'stores' into 'stores'. (For each particular notion of 'program' we must suitably choose the 'names' and 'contents' needed to construct the domain of its 'stores'.)

1.2. *A more general concept of 'program'; a basic question about the conventional concept*

To make it easier to consider a wider range of concepts of 'program', some of which are non-Von Neumann, let us define a more general property of 'programs' and then define what we mean by 'Von Neumann programs':

(A) A 'program' represents a mapping of some domain D into itself.

(B) The domain associated with 'Von Neumann programs' is some domain of 'stores'.

Thus for each notion of 'program' there is associated a particular domain D; and for each kind of 'Von Neumann program' there is a particular domain of 'stores' built from certain 'names' and certain 'contents'. Of course such a domain includes each possible association of 'names' with 'contents' in one of its 'stores'.

If you accept these notions as fundamental elements of the concept of 'program' and of 'Von Neumann program', then the question I should like to raise for your consideration is an important one, one which, oddly enough, seems to have received little attention. The question is this: *is the choice of 'stores' as the domain for 'programs' the correct choice?* That is, is there perhaps some other domain such that, if the notion of 'program' were associated with mappings of this new domain into itself, then the resulting concept of 'program' could be simpler, more powerful and elegant than the Von Neumann concept?

1.3. *Evolution of the Von Neumann concept; psychological barriers to adoption of non-Von Neumann concepts*

Before we consider notions of 'program' founded on other domains, let us review some of the reasons why 'stores' were chosen for the domain of conventional programs. Of course, in the first place the Von Neumann computer itself required programs based on the domain of stores. And when 'higher level' languages began to evolve it was important that their programs correspond closely with machine programs, therefore they naturally adopted an abstract notion of 'store' quite close to that of the machine. Thus the evolution of programming languages from machine languages is a natural and basic reason for the choice of 'stores' as the domain for 'programs'. But I believe there is another, deeper reason.

In natural language and in mathematics one of the most universal and deeply ingrained practices in thinking and writing is the use of names to stand in place of their referents. In every natural language sentence or mathematical expression most symbols denote something other than themselves. Thus in *'$a+b$'* it is universally understood that we are to add, not the letters *'a'* and *'b'*, but some numbers to which they refer. The store that is implicit in every conventional program is the repository of this name-referent association that is so much a part of our traditional way of thinking.

If we choose any domain other than 'stores' for our concept of

'program', then it may no longer be possible for the programmer to use a 'name' or 'variable' to refer to an object. Thus non-Von Neumann concepts of 'program', those employing domains other than 'stores', threaten to violate deep-rooted traditions of thought, therefore such concepts tend to confuse and disturb programmers.

Because the conventional concept of 'program' is so closely linked by its use of 'stores' to the traditional use of names in natural language and mathematics, the adoption of new concepts may cause many computer scientists a certain amount of anguish. But if we are to consider new concepts of 'program' that may yield a more profound order in the realm of programs, it is important that we be aware of the psychological difficulties we may face in dealing with such new ideas.

Alternative concepts of 'program' not based on the domain of 'stores' have been evolving over a long period. Their evolution has been confused because many developers of the new view have been unable to free themselves from the old one and have sought to find a viable mixture of the two. Thus there is as yet no consensus on the key elements of a new concept. Pure LISP was the first language whose program domain was not 'stores'. But most versions of LISP, other 'applicative' languages such as GEDANKEN [17], as well as others emphasizing functional elements, such as APL [11], all tend to incorporate the traditional notion that programs transform stores.

1.4. *The object level and the function level viewpoints*

One important consequence of the conventional concept of 'program' is an emphasis on the 'object level' view of programming. Since stores hold 'objects in their cells, programming becomes a process of describing how to combine objects to form other objects, a program being a description of how to combine the 'input objects', which are found in given cells, to form the desired 'output objects' and deposit them in the proper output cells.

Even the traditional applicative languages such as pure LISP and ISWIM [14] emphasize the object level view of programming. Though their programs do not map stores into stores, they do use names for objects and their basic semantics automatically builds a kind of store (during the execution of a program) in which the values of variables mentioned in the program are kept. And, like conventional programs, these programs are primarily concerned with combining objects. For example, consider the

following definition of the object-to-object function f in the style of the lambda calculus [6]:

$$f = \lambda x \cdot h(x, g(x)).$$

We are given the functions g and h and we wish to build the function f. We do this at the object level by introducing the object x, forming the object $g(x)$ and then the 'result object' $h(x, g(x))$; we then use the principal program-forming operation of the lambda calculus, lambda abstraction, to abstract the object variable x and convert the result object into the function f. Thus we see that the object level definition of f does not directly combine the functions g and h from which f is to be built, but instead this definition descends from the function level of g and h to the object level of x, $g(x)$, and $h(x, g(x))$, then ascends again to the function level of f. This down-then-up-again style is characteristic of the object level approach. (For further discussion of 'functional' object level programs see [4]).

As we shall see later, there is a 'function level' approach to programming that defines a new function in terms of given ones without descending to the object level. For example, the function level definition of f as above would use two program-forming operations to build f from the functions g and h and the identity function; no object or object variable would appear in the definition. In general the function level approach uses program-forming operations (functionals) to combine given functions directly to form the desired new one. ('Constants' in an object level definition are 'lifted' to constant-valued functions in the corresponding function level definition.) Object level definitions all correspond to some isomorphic function level one, but there are many function level definitions that have no isomorph at the object level, although of course there is some nonisomorphic object level definition for the defined function.

Of the various 'functional' or non-Von Neumann approaches to programming, we shall argue that it is this function level approach that offers the best possibility to have a universe of 'programs' with a deeper mathematical order than can be found in the universe of Von Neumann programs. It is also an approach that departs the farthest from the traditional use of names and variables for objects, hence it is also an approach that will be likely to cause the kind of deep unease I have tried to indicate above.

1.5. *Goals of the function level approach to programming*

In the area of data types, as in e.g. [5,9,21,24]. we have already gone far in moving from the object level viewpoint to a function level one. We have moved from focussing on the *objects* of a data type and on their 'structure' to an emphasis on (a) the *operations* used to build and manipulate those objects and their structure, and on (b) the algebraic properties of these operations as expressed in various 'axioms'.

The non-Von Neumann, function level approach to 'programs' seeks to shift our viewpoint similarly, to introduce the kind of order into the universe of 'programs' that the abstract, algebraic approach to data types has introduced into the universe of objects, an order represented by axioms or laws about the operations over the given universe (of objects or of 'programs'). With data types, we are concerned with objects (data) and with operations on them; with 'programs' we are concerned with objects (data), operations on objects (programs), and operations on programs (program-forming operations or PFOs).

For a long time we regarded programs as a kind of yard-goods pieced together by semicolons and begin/ends, just as we used to regard data structures as pieced together by commas and brackets. More recently, in the era of 'structured programs', we noticed that semicolons, if-then-elses, and while-do's were 'control operations' that served to 'structure' programs.

The goal of the function level approach to a concept of 'programs' is to move now to an emphasis on the operations (PFOs) used to construct programs and on the algebraic properties of those operations. For example, in this approach if-then-else becomes a program-forming operation, not a 'control' operation; it maps three given programs into one new one and it has important algebraic properties with respect to other PFOs, just as 'addition' in a ring has important properties with respect to the other ring operation, 'multiplication'. It is these algebraic properties that make it possible to transform programs from one form to another and to solve equations for programs, just as it is the properties of the ring operations that enable us to transform ordinary algebraic expressions and to solve equations.

1.6. *Incompatibility of the Von Neumann concept of 'program' and the function level approach to programming*

The choice of 'stores' as the domain for Von Neumann programs has

two immediate, harmful consequences for our ability to write general, composable programs (we discuss these in the next section). But it is an indirect consequence of that choice that blocks the use of a function level style in Von Neumann programming: the choice of 'stores' as the domain of programs limits our choice of program-forming operations, apparently to PFOs lacking the required algebraic properties. (Again, we shall see later why this is so.)

1.7. *Requirements and prospects for the function level view of 'programs'*

We shall show that by enlarging the domain for 'programs' beyond that of 'stores' that we can form domains for a new concept of 'programs' and that, using this new concept, we have a wider choice of program-forming operations from which we can then choose a set of PFOs with a strong algebraic structure.

If we adopt both the new concept of 'program' (with its new domain) and the new PFOs that become available with it, then we can move from an object-centered view of programming to a function-centered one (or perhaps a relation-centered one) – despite the temporary trauma this may cause. Already there are signs that a more profound order can be found in the universe of the new 'programs' than we have found in the universe of the old, but the concept and the form of its programs, being new, are very much open to change. Having incubated for 20 years, the non-Von Neumann concepts of 'program' are just beginning to develop a function level view and 'programs' built by PFOs with algebraic structure.

Much work remains to be done before a definite function level or other extended concept of 'program' (with its accompanying methodology and implementations) can be developed and achieve some form of consensus. Finding the 'best' concept of 'program' and achieving such a consensus would be aided by a larger abstract theory concerning the effect of various representations of programs over various domains on the properties of such systems. Much will depend on our ability to exploit the algebraic structure of PFOs to optimize programs. It will also be important to develop new computer architectures that directly implement both the PFOs and the composite data objects of the new concept and that exploit the inherent parallelism of its programs.

Of course, as with any new approach, we may find difficulties with the

function level view that make it unsuitable, but that too can only become clear after a lot of further work.

1.8. *Organization of this paper*

Section 2 discusses two difficulties associated directly with the choice of 'stores' as the domain for programs. Section 3 indicates how the Von Neumann concept of 'program' can be extended to a non-Von Neumann concept by enlarging the domain for programs. It discusses the advantages of this extension; it does so by describing a 'typical' Von Neumann language called L and its non-Von Neumann extension, called L^*, and comparing the two. The rest of the paper is best outlined by giving the section and subsection titles:

2. Two fundamental problems with the Von Neumann concept of 'program'

2.1. *Problem domains are not program domains*

Programmers are never approached with the following request: "I have a set of stores of this kind and I want you to write a program that will transform them into stores of this other kind." Instead they are asked, given a set of files and transactions, to write a program to transform a file-transaction pair into a new file and a response. Or they are asked to write a program that inverts matrices, and so on. But 'programs' can only map stores into stores.

Thus the primary difficulty with the Von Neumann idea of 'program' is that the solution of a problem is *never* a program. For example, there is no program for finding square roots. We tend to think this last statement is wrong because we regard as insignificant the 'explanation' needed to connect a so-called 'square-root program' with the actual mapping that carries numbers into their square roots. Thus a 'square root program', to be meaningful, actually consists of a program (that maps stores into stores) plus a storage plan, the 'explanation', without which the program proper is useless. If the program takes its input from, say, cell a and deposits its output in cell b of the store, then the storage plan is a statement of these facts. To compute the square root of some number n then requires two other mappings in addition to that of the program itself: an input mapping that creates a store with n in cell a and an output mapping that maps a store into the contents of cell b; then the composition output$\circ$program$\circ$input is a function that maps numbers into their square roots, where 'input' and 'output' depend on the storage plan of 'program'. (In practice the use of a square root program as a subroutine employs mechanisms to conform the storage plan of the program to that of its context or vice versa, according as it is called by name or by value.)

The choice of 'stores' as the domain of Von Neumann programs means that a typical program has a 'purpose', that is, to map some domain D into another domain E (possibly the same), which it can achieve only partially and indirectly; it can be interpreted as accomplishing its 'purpose' only by a mental transformation that requires full knowledge of its storage plan. This storage plan constitutes a kind of artificial representation of the domains of the program's 'purpose', D and E, in terms of stores.

The representation problem that is intrinsic in the Von Neumann approach, representing problem data by stores, greatly complicates programming by interposing storage plans between the straightforward 'purpose' of a program itself. This makes it impossible for a Von Neumann program to achieve its 'purpose' directly.

2.2. *The principal program-forming operation, composition, is ineffective for building Von Neumann programs*

The Von Neumann requirement for representing data by stores is also the main reason we have found it so difficult to effectively build new programs from existing ones. The principal program-forming operation for building programs is composition, thus from programs p and q we can form the new program $p;q$. But if p and q are Von Neumann programs written independently, then $p;q$ is almost certain to be meaningless. That is, the program $p;q$ will not achieve the 'purpose' of p followed by the 'purpose' of q (even in the indirect sense in which programs achieve their 'purpose') except in the unlikely event that q happens to take its inputs from just those cells in which p places its outputs.

Thus the Von Neumann concept of 'program' assures that composition is useful only for piecing together programs that are written together under a unified storage plan, whereas in mathematics and in a proper universe of 'programs', composition is the primary, most powerful operation for building functions or programs.

Some readers will complain that the above remarks about composition fail to take into account the existence of procedure declarations, since these enable one to make specific programs into general ones whose storage plans can be varied. Suppose one wishes to compose two procedure calls so as to obtain the composition of their 'purposes'. If they are $P(x, y, z, w)$ and $Q(x, y, z, w)$, where in each case x and y are inputs and z and w are outputs, then $P(a, b, c, d); Q(c, d, e, f)$ transforms a and b into e and f. Here

we have merely simplified the storage planning problem: that of designing two problems to be composed so that the storage plan for the first is appropriate for that of the second. For ordinary programs the 'store' they operate on is large and hence storage plans involve much detail; in the case of our procedure calls their 'stores' have effectively only four 'cells' that require planning (the other, local cells being isolated from all other cells in the basic store). Thus $P(a, b, c, d)$ becomes an actual program only after *a, b, c,* and *d* are chosen; once chosen the chance that $P(a, b, c, d)$ is meaningfully composable with another independent program is as slight as ever.

Thus by the use of procedure declarations we make it possible to plan storage at the time of use, rather than the time of writing a procedure. This means that storage plans can be more local and flexible when using procedure calls in place of 'programs'. But that still means that $p;q$ is almost always meaningless unless p and q have a common storage plan. Compare this situation with the composition of functions in mathematics. In that context a function and its 'purpose' are the same: a function maps its intended domain directly into its intended range without any intervening representations. Thus the composition of two functions is always meaningful if the composition of their 'purposes' is meaningful.

The contrast between composition of Von Neumann programs and that of functions is perhaps best seen by an example. If feet-inch is a program to convert feet to inches and inch-yard is one to convert inches to yards, then

feet-inch ; inch-yard

is a program, but it will almost certainly *not* convert feet to yards. On the other hand if the same names denote two conversion *functions,* then

inch-yard ∘ feet-inch

is a function that will convert feed to yards. The difference follows from the fact that the *program* feet-inch maps stores into stores whereas the *function* feet-inch maps numbers (in feet) into numbers (in inches).

Of course Von Neumann languages provide functions for use within *expressions,* and these can be composed meaningfully within individual expressions. Here we are addressing the problem of building *programs* from pre-existing ones by composition, a more central issue.

3. The extension of the Von Neumann concept of 'program' to a simpler, non-Von Neumann concept

3.1. *Introduction*

We have considered two important defects in the Von Neumann concept of 'program' that are direct consequences of choosing 'stores' as the domain for programs. These defects can be understood without examining the structure of 'programs' determined by the operations used to build them. But the most fundamental difficulties of Von Neumann programs and the languages used to express them lie in their unnecessarily complex structure, in our inability to use a powerful set of program-forming operations to build programs, and in the fact that the PFOs that we do use lack the algebraic properties that would allow us to prove useful general theorems about large classes of programs.

In order to illustrate these difficulties we propose to describe the elements of a typical, but oversimplified Von Neumann language, L. Since our purpose is to expose defects in the simplest conventional concepts of 'program', it will not be necessary to include in L many of the complications that appear in real languages. That complexity is very well illustrated in the descriptions of real languages in the literature of denotational semantics [19,20]. It will become apparent that the defects we discuss in L can only become worse with the addition of further features.

The complexity of real languages revealed in denotational semantics can be viewed in two different ways. The notion of 'continuations', for example, can be viewed as an ingenious invention that makes it possible to cope formally with various features of conventional languages. On the other hand, continuations can be regarded as yet one more level of mathematical obfuscation and complexity that is essential to shore up a failing concept of 'program', a concept tottering under its own weight.

In the following we shall briefly examine the structure of L-programs in terms of the operations used to build them and to build their components. We shall then describe a non-Von Neumann language L^* that is an extension of L in the sense that (a) the domain for L^*-programs contains that for L-programs and (b) every L-program has an exact image within a small subset of the programs of L^*.

Up to this point we have emphasized the association of the concept of 'program' with the domain that programs operate on. In the following we shall give equal emphasis to the second, independent element that is central

to the concept: the program-forming operations used to construct 'programs' and the properties of those PFOs.

We shall see that L^*-programs can be built with a set of PFOs different from that used to build L-programs, that these PFOs have a strong algebraic structure where those of L do not, and that L^*-programs have a simpler structure than L-programs. We shall see that L^*-programs represent a function level approach to programming and L-programs an object level one. But first we must discuss a few issues that have been dealt with only vaguely, and then describe L and L^*. We will then be able to compare the two approaches in more detail.

3.2. *Program domains*

By definition we have assumed that 'programs' map some domain D into itself. We do so because, if 'programs' mapped D into E, then, unless E is contained in D, it would not make sense to compose two such programs.

3.3. *Abstract programs and PFOs versus concrete programs and PFOs*

'Programs' can be regarded either as mappings (infinite entities) or as representations (finite expressions) for such mappings, that is, as 'abstract' or 'concrete' programs. In our discussions of 'expressions' in L (store-to-object mappings) and of programs in L and L^* we do not want to constantly distinguish between their abstract (mapping) and concrete (representation) aspects. Even more, we want to speak of expression-forming and program-forming operations without saying whether we mean operations that form abstract expressions or programs, or that form concrete ones. We sketch informally how this can be done without confusion.

Let us consider only programs, since the treatment of expressions is similar. Let us agree that all concrete programs that represent one abstract program are 'equivalent' and that when we speak of a 'program' we may be referring either to the abstract program (mapping) or to one of the concrete programs that represent it or to the entire equivalence class of concrete programs. This gives us, at least, a clear understanding of the ambiguity we shall allow ourselves in the use of the term 'program'.

Now consider an abstract program-forming operation π that builds, for example, the abstract program $\pi(p, q)$ from the abstract programs p and q. Then there is a corresponding concrete program-forming operation $\tilde{\pi}$ such

that, for any concrete program-representations $\tilde{p}$ and $\tilde{q}$ for p and q. then $\tilde{\pi}(\tilde{p},\tilde{q})$ builds a concrete program that represents $\pi(p,q)$. Of course, concrete program-forming operations must be blind to the particular choice of concrete program $\tilde{p}$ used to represent some abstract p; replacing an argument of a concrete PFO by an equivalent one must give equivalent results. Again, for each abstract PFO π there are many corresponding ('equivalent') concrete PFOs $\tilde{\pi}$ that differ in the representations they produce for an abstract program.

As with the term 'program', we shall use the term 'program-forming operation' or 'PFO' to mean either some abstract PFO π or some corresponding concrete PFO $\tilde{\pi}$ or the entire equivalence class of concrete PFOs that correspond to π.

3.4. *Algebraic structure of a set of operations*

We have referred to a set of operations as having a 'strong algebraic structure'; although we do not intend the phrase to have a precise meaning, it deserves some explanation. To the extent that pairs or tuples of operations in the set are related by algebraic laws, we think of the set as being structured by those laws. Thus the set of operations, addition and multiplication, has a strong algebraic structure since the only pair in the set is related by the distributive law $(a+b)c=ac+bc$, which expresses a multiplication involving addition as an addition involving multiplications. It is this kind of law, with this kind of 'symmetry' that has the greatest intuitive 'strength' in our notion of algebraic structure. Thus, for example, the recursive definition of while-do using condition and composition lacks this 'symmetry' and hence is not as 'strong' a law.

The intent of our notion of algebraic structure of a set of operations is that strong structure goes with strong theorems about the operations and their domains and weak structure with weak theorems. We do not pretend that the notion is precise or that it always achieves this intent. In the case of the set comprising addition and multiplication our informal notion does work: it has a strong algebraic structure and there are strong theorems about the operations and their domain of numbers, e.g., the general solution of quadratic equations.

A similar example, but with weak structure, is the set of operations of addition and square root. Without the introduction of multiplication there are no strong laws relating addition and square root, therefore there are no

strong, general theorems about systems with just these two operations, only particular ones like sqrt(4) + sqrt(9) = 5.

I believe it will turn out that the reason we find so few strong, general theorems about conventional programs is that their program-forming operations have a weak algebraic structure. We shall see that the PFOs of L have a weak structure; but those of L^* have a strong structure and, in accordance with the above notion, there are beginning to emerge some general theorems about programs of this new kind (see [2,3,13,22]) and I believe we shall see more theorems and stronger ones. We shall examine the algebraic structure of the PFOs of both L and L^* in more detail later on.

3.5. *The Von Neumann language L*

Our typical but oversimplified Von Neumann language L has the following elements:

(1) A set of L-programs that map L-stores into L-stores.

(2) A set of L-stores, each store being a set of cells, each cell having a 'name' and 'contents'.

(3) A set of L-objects that includes all the 'names' and 'contents' found in any L-store. We assume that the set of L-objects is a rich one containing objects built from elementary objects such as 'true', 'false', numbers, symbols, etc., by constructions such as sequences, arrays, stacks, files, etc.

(4) A set of L-expressions that map L-stores into L-objects.

(5) A set of object-forming operations such as +, square root, as well as more complex operations on more complex objects.

(6) A set of program-forming operations that build 'structured' L-programs from L-expressions and L-programs. These are: composition (semicolon), if-then-else, and while-do.

3.6. *The structure of L-programs*

L-programs are built on three planes. On the highest plane are L-programs; these are built from elementary L-programs by PFOs. On the middle plane are the elementary L-programs (assignments); these are built from 'names' and L-expressions by the 'assignment-forming operation'. On the lowest plane are L-expressions; these are built by object-forming operations (actually, by operations isomorphic to these) from elementary L-expressions, which are L-objects and L-variables. L-programs map L-stores into L-stores. L-expressions map L-stores into L-objects.

3.6.1. Elementary L-expressions map stores into objects. An L-object, serving as an L-expression, maps any store into the object itself. Thus 3 maps any store s into 3 (we write $3:s=3$). An L-variable $\hat{n}$ (associated with 'name' n) maps a store s into the contents, in s, of cell n ($\hat{n}:s=$ contents cell n in s). (We do not deal with the question of subscripted variables or other compound cell names, nor with the necessary distinction between 'names' (objects n) and 'variables' (expressions $\hat{n}$).)

3.6.2. Constructing L-expressions with object-forming operations. Actually, expressions are built with expression-forming operations that are derived from object-forming ones as follows. If $o(x_1,\ldots,x_n)$ is an object-forming operation of L, then $o_e(e_1,\ldots,e_n)$ is the corresponding expression-forming one, where the store-to-object mapping of the latter is

$$o_e(e_1,\ldots,e_n):s=o(e_1:s,\ldots,e_n:s)$$

for any store s. For example $+_e$ builds an expression from the expressions a and b, where $(a+_e b):s=a:s+b:s$ for any store s.

Having noted the isomorphic relationship between the object-forming and expression-forming operations of L, from here on we can pretend they are identical.

3.6.3. Properties of L-expressions. The points to notice about expressions are these:

(a) They are object level constructions describing the combination of objects to produce others.

(b) They represent the principal 'work' of a program.

(c) Laws concerning object-forming operations are suitable for proving facts about objects, whereas we often want to prove facts about programs rather than objects.

(d) Expressions cannot be built by composition since their domain (stores) and range (objects) differ.

3.6.4. Elementary L-programs are assignments; these are built by an assignment-forming operation, $:=$, from a name n and an expression e, yielding the assignment $n:=e$. The store-to-store mapping of this assignment satisfies the following two equations:

$$\hat{n}:s'=e:s, \tag{1}$$

$$\hat{n}' : s' = \hat{n}' : s \quad \text{for all } n' \neq n \tag{2}$$

where $s' = (n := e) : s$ for all stores s. Thus (1) asserts that the contents of cell n in s' (obtained from s by 'executing' the assignment) is the value of e with respect to s, and (2) asserts that the contents of other cells in s' are the same as in s.

3.6.5. Constructing L-programs with PFOs. The PFOs of L are composition (;), if-then-else and while-do; these are used to build programs from elementary ones. Thus if p and q are L-programs and b an L-expression, then

$p ; q$

if b **then** p **else** q

while b **do** p

are L-programs. Note that here we are emphasizing the operational character of PFOs as operations on program-mappings. For example, if-then-else is not to be seen as a kind of punctuation used to divide up the text of **if** b **then** p **else** q into the subtexts of b, p and q. Instead, we see it as an operation with three operands: b, a mapping of stores into objects, and p and q, mappings of stores into stores. The result of applying if-then-else to these operands is a store-to-store mapping we denote by **if** b **then** p **else** q.

Of course, as discussed earlier, there are many equivalent concrete-program-forming operations for if-then-else that map three concrete representations for the mappings b, p, q into some concrete representation for the mapping **if** b **then** p **else** q; as agreed earlier we have lumped the one abstract PFO and the many concrete PFOs for if-then-else into the one term 'if-then-else'. For this 'lumping' to be valid one must check that, if one equivalent concrete program is substituted for another in a program built by a PFO, then the resulting program will be equivalent to the original. This will be true for any PFO if every program built by it only *applies* the constituent programs in obtaining its result, since equivalent programs are indistinguishable in that case. Every PFO we shall use has this property.

The mappings associated with each of the PFOs of L are given by the following definitions, for all stores s:

composition $(p;q):s = q:(p:s)$;

if-then-else $\textbf{(if } b \textbf{ then } p \textbf{ else } q):s = \mathrm{p}:s$ if $b:s =$ 'true',

$= q:s$ if $b:s =$ 'false',

$=$ undefined otherwise

while-do $\textbf{(while } b \textbf{ do } p):s = s$ if $b:s =$ 'false',

$= \textbf{(while } b \textbf{ do } p):(p:s)$ if $b:s =$ 'true',

$=$ undefined otherwise.

3.6.6. Properties of L-programs. The main discussion of properties of L-programs is best left until we have described L*-programs and can compare the two. For the present the main thing to note about L-programs is their three-plane construction that divides them into expressions (that do most of the work, and whose object-forming operations may have good algebraic properties), assignments (that are built from expressions and change one cell of a store), and programs (that are built from assignments). Thus the power and possible algebraic elegance to be found in L-programs is confined to expressions whose individual effect in a program is restricted to changing one cell. (We shall examine the algebraic structure of the PFOs of L later on.)

3.7. *An extension of L: the non-Von Neumann language L**

The language L* has the following elements:

(1) A set of L*-programs that map L*-objects into L*-objects.

(2) A set of L*-objects (that contain all L-stores and L-objects as described below).

(3) A set of program-forming operations.

The set of L*-objects contains all L-stores and all L-objects and hence every 'name' and 'contents' found in any L-store. Furthermore, the set of L*-objects is closed under sequence-formation; thus if $x_1, \ldots, x_n$ are L*-objects then so is the sequence $\langle x_1, \ldots, x_n \rangle$. We shall see that enlarging the domain for 'programs' from 'stores' as in L to the domain of L*-objects results in a surprising simplification in the concept of 'program' (already it is evident that L* has fewer elements than L).

3.8. *The structure of L*-programs*

*L**-programs are built on one plane; they are built from elementary *L**-programs by PFOs.

3.8.1. Elementary *L**-programs. These are the given, primitive programs of *L**. They, and all *L**-programs, have a single argument; they are functions from *L**-objects into *L**-objects. They include the object-forming operations of *L*, such as + and square root, and various functions and predicates for accessing, rearranging and testing sequences, as well as functions for dealing with whatever special data types are included in the set of *L**-objects. For example, $+:\langle 3,4\rangle = 7$, null: $\langle\ \rangle$ = 'true', equal: $\langle A,B\rangle$ = 'false', length: $\langle A,B,C\rangle = 3$. Note that *A* and *B* are simply objects, they do not name other objects.

3.8.2. Constructing *L**-programs with PFOs. Having chosen a domain for programs does not determine the program-forming operations that can be used to build them. In *L* we have used the traditional PFOs for 'structured' programs. However, since *L**-programs have a larger domain and a simpler structure than *L*-programs, we shall use a somewhat different set of PFOs to build *L**-programs. We want this new set to have a strong algebraic structure; as it turns out the following three PFOs for *L** have such a structure (we describe for each one the program it constructs in terms of its argument-programs):

composition builds the program $p \circ q$ from programs *p* and *q*, where

$$p \circ q : x = p : (q : x) \quad \text{for all } L^*\text{-objects } x.$$

condition builds the program $p \rightarrow q ; r$ from the programs *p*, *q* and *r*, where

$$\begin{aligned} (p \rightarrow q : r) : x &= q : x && \text{if } p : x = \text{'true'}, \\ &= r : x && \text{if } p : x = \text{'false'}, \\ &= \text{undefined} && \text{otherwise.} \end{aligned}$$

for all *L**-objects *x*.

construction builds the program $[p_1, \ldots, p_n]$ from programs $p_1, \ldots, p_n$, where

$$[p_1, \ldots, p_n] : x = \langle p_1 : x, \ldots, p_n : x \rangle \quad \text{for all } L^*\text{-objects } x.$$

Composition is essentially the same as the PFO used in L except for its domain, notation, and the order of its arguments. Condition is slightly different from the if-then-else of L in that all three arguments are L^*-programs; this difference improves its algebraic relationship to composition. Construction is entirely new; in fact, it cannot be used in L since the mapping it builds from L-programs would map a store into a sequence of stores, and a sequence of stores is never a store, hence this mapping cannot be an L-program and construction cannot be used to build L-programs.

A fourth program-forming operation that is essential in L^* is 'constant', one that is different in that it builds a program from an object.

constant builds the program $\bar{x}$ from the object x, where

$$\bar{x}:y=x \quad \text{for all } L^*\text{-objects } y.$$

The reader may wonder at this point why there is no PFO in L^* analogous to while-do. Of course we could introduce one without difficulty. But the three principal PFOs above have a strong algebraic structure whereas there are no strong algebraic laws relating while-do to these (other than the function level defining equation for while-do, which relates it to composition and condition, a 'weak' law). Furthermore, the algebraic structure of the principal PFOs allows us to formally solve and reason about a much larger class of recursive equations than the class of tail recursive ones for which while-do represents solutions (see [3,22] for a discussion of such formal solutions).

Since recursive equations comprise a more powerful and expressive way of defining programs than while-do, we shall allow them as program definitions in L^*; thus we have no need for while-do and can retain the strong algebraic structure of the PFOs of L^*.

One of the major benefits of the L^* approach is that one can use a great many operations for building programs in L^*. Their analogues can be used in L only to build object-forming operations, operations that are then used to build L-expressions. Adopting this approach introduces a fourth plane into the structure of L-programs: (1) build object-forming operations from elementary ones using 'operation-forming operations', (2) build expressions from object-forming operations, (3) build assignments from variables and expressions, and (4) build programs from assignments with PFOs. This structure is found in APL [11].

A few other PFOs we might use in L^* are the following.

insert builds $/p$ from p, where

$$/p:\langle x_1\rangle = x_1,$$

$$/p:\langle x_1, x_2, \ldots, x_n\rangle = p:\langle x_1,\ /p:\langle x_2, \ldots, x_n\rangle\rangle;$$

apply-to-all builds αp from p, where

$$\alpha p:\langle x_1, \ldots, x_n\rangle = \langle p:x_1, \ldots, p:x_n\rangle;$$

fetch builds $\uparrow x$ from the object x where

$$\uparrow x:s = \text{contents of cell } x \text{ in store } s \quad \text{for any store } s:$$

store builds $\downarrow x$ from object x where

$$\downarrow x:\langle y, s\rangle = s'$$

where s' is s with contents of cell x now equal y.

3.9. *L*-images of L-expressions and L-programs*

By the 'image' $\tilde{p}$ in L^* of an L-program or an L-expression p we mean the L^*-program $\tilde{p}$ such that $\tilde{p}:s = p:s$ for all L-stores s; we shall not be concerned whether $\tilde{p}$ is minimal, i.e., undefined for all non-stores. Because of the defects of L-programs, their images in L^* are perhaps the least interesting and least useful programs in L^*. But it may be of interest to some readers, as an exercise in programming in L^*, to see how expressions and programs of L can be built (as programs) in L^* using its different set of PFOs. (This section is not essential to understanding later ones and may be skipped.)

3.9.1. Elementary L-expressions. We illustrate images by examples and consider the L^*-images of the two kinds of elementary L-expressions, that of the L-object 3 and that of the L-variable $\hat{v}$. The L^*-image of 3 is $\bar{3}$; if we take v to be an L^*-object, then the L^*-image of $\hat{v}$ is $\uparrow v$. To back up this claim we must show that each entity and its image is, one in L and the other in L^*, the same mapping of stores into objects:

$$3:s = 3 \quad \text{in } L,$$

$$\bar{3}:s = 3 \quad \text{in } L^*,$$

$$\hat{v}:s = \text{contents of cell } v \text{ of } s \quad \text{in } L,$$

$$\uparrow v:s = \text{contents of cell } v \text{ of } s \quad \text{in } L^* \text{ (see the PFO fetch).}$$

3.9.2. Composite L-expressions. If a and b are L-expressions and $\tilde{a}$ and $\tilde{b}$ are their L^*-images, then $+\circ[\tilde{a},\tilde{b}]$ is the L^*-image of the L-expression $a+_e b$; $\text{sqrt}\circ\tilde{a}$ is the image of $\text{sqrt}_e(a)$, since, for all stores s:

$$(a+_e b):s = a:s + b:s \quad \text{in } L,$$

$$+\circ[\tilde{a},\tilde{b}]:s = +:\langle \tilde{a}:s, \tilde{b}:s\rangle = a:s + b:s \quad \text{in } L^*,$$

$$\text{sqrt}_e(a):s = \text{sqrt}\ (a:s) \quad \text{in } L,$$

$$\text{sqrt}\circ\tilde{a}:s = \text{sqrt}:(\tilde{a}:s) = \text{sqrt}(a:s) \quad \text{in } L^*.$$

3.9.3. Elementary L-programs (assignments). The L^*-image of $v:=e$ is $\downarrow v\circ[\tilde{e}, \text{id}]$, where $\tilde{e}$ is the image of e and id is the identity function:

$(v:=e):s = s'$ in L, where contents of cell v in s' is $e:s$.

$\downarrow v\circ[\tilde{e}, \text{id}]:s = \downarrow v:\langle \tilde{e}:s, s\rangle = s'$ in L^*, where s' is the same as above, since $\tilde{e}:s = e:s$ (see PFO 'store').

3.9.4. Composite L-programs. The L^*-image of $p;q$ in L is $\tilde{q}\circ\tilde{p}$. The L^*-image of **if** b **then** p **else** q is $\tilde{b}\rightarrow\tilde{p};\tilde{q}$. The L^*-image of **while** b **do** p is the solution f of the equation

$$f = \tilde{b}\rightarrow f\circ\tilde{p};\ \text{id}.$$

(To apply f as defined above to any object, apply the right side.) I leave it to the interested reader to verify that these L^*-images represent the same store-to-store mappings as the original L-programs.

3.10. *The algebraic structure of the PFOs of L* and L*

The principal three PFOs of L^* have a strong algebraic structure as shown by the following 'strong' laws that relate each pair (with two laws for one of the pairs). For all programs p, f, g, h, $f_1,\ldots,f_n$ the following function level identities hold:

Composition and condition

$$f\circ(p\rightarrow g;h) = p\rightarrow f\circ g; f\circ h, \tag{1}$$

$$(p\rightarrow g;h)\circ f = p\circ f\rightarrow g\circ f; h\circ f. \tag{2}$$

Composition and construction

$$[f_1, \ldots, f_n] \circ h = [f_1 \circ h, \ldots, f_n \circ h]. \tag{3}$$

Construction and condition

$$[\cdots (p \rightarrow g; h) \cdots] = p \rightarrow [\cdots g \cdots]; [\cdots h \cdots]. \tag{4}$$

(4) holds either in the domain for which p is boolean -valued or always if the sequence constructor is strict.

Each of the above laws relates two PFOs, call them A and B, where each law expresses a program built by A (involving a program built by B) as an equivalent program built by B (involving programs built by A).

Many other algebraic laws hold in L^*, some relating other PFOs to the principal ones and each other, and others that involve primitive L^*-programs. But most of these laws are less symmetric, 'weaker' than those above. For example, the law relating the PFOs insert, composition and construction.

$$/f \circ [g_1, g_2, \ldots, g_n] = f \circ [g_1, /f \circ [g_2, \ldots, g_n]]$$

is a 'function level' version of the second part of the object level description of $/f$ given earlier under the PFO insert.

Since any abstract L^*-program can be represented by programs built from suitable primitive programs by the three principal PFOs and recursive equations, the strength of their algebraic structure indicates that there should be a lot of strong general theorems involving these PFOs whose universally quantified variables range over L^*-programs.

Now consider the PFOs of L. They satisfy only one symmetric law, which relates composition and if-then-else:

$$(\textbf{if } b \textbf{ then } p \textbf{ else } g); r = \textbf{if } b \textbf{ then } (p; r) \textbf{ else } (q; r).$$

This is the analogue of (1); the analogue of (2) fails because one cannot compose a program and an expression to form an expression (even if this is allowed there are other complications). There are no symmetric laws relating while-do with either composition or if-then-else (the function level definition of while-do is a 'weak' law that relates all three PFOs of L).

The weak algebraic structure of the PFOs of L is consistent with the existence of few general theorems about the programs of L; instead we tend to find theorems about particular programs ("my program is

correct") or small classes of programs ("this program, where o is any associative operation, is equivalent to that one"). One can (and should) ask whether there are better sets of PFOs for building L-programs. The answer is unclear; at this point all we can say is that construction, which relates well to composition and condition (which is close to if-then-else) to form a strongly algebraic triad, cannot be used as a PFO for any programs that have 'stores' as their domain. Since it is hard to do without the PFOs for composition and condition, or something similar, this does not bode well for finding PFOs for L with a good algebraic structure.

Not only does L^* have a strong algebraic structure in its major PFOs but also it has, as noted above, weaker laws relating these to lesser PFOs such as insert and apply-to-all, and to primitive L^*-programs, such as those that select the nth element of a sequence or rearrange various data structures. These lesser PFOs and primitive L^*-programs could only be used within expressions of L, therefore theorems relating all of these elements would be blocked in L by the barrier between programs and expressions. In L^*, on the other hand, we can expect to obtain theorems that interrelate both major elements of a program-scheme (corresponding to the program plane in L) and minor elements (corresponding to the expression plane in L), since all these are programs in L^* and are all put together by PFOs that are related by algebraic laws, either weak or strong.

4. Comparison of L and L^*

4.1. *Problem domains and program domains*

We noted earlier that L-programs never solve problems directly since they are store-to-store mappings and real problems require other kinds of mappings. Provided the data types of a problem are in the set of L^*-objects, there is an L^*-program that is a rather direct solution. Thus there are L^*-programs for square root, matrix inversion and file updating that do not require storage plans to be useable, programs that map numbers into their square roots, matrices into their inverses, and so on.

4.2. *Composition of programs*

Again we observed earlier that in L the composition $p;q$ of two independent programs has little chance of achieving a meaningful program

that represents the composite purpose of p and q. Thus if the purpose of p is to transform A's into B's, and that of q is to transform B's into C's, then p; q will almost certainly not transform A's into C's unless p and q have a common storage plan.

On the other hand, if p and q are L*-programs for the same purposes, then the results of p will be B's, as will the arguments of q, and $q \circ p$ is an L*-program to map A's into C's.

4.3. *Relationship between a composite program and its subprograms*

We have observed that composition cannot assemble independent programs in L into a meaningful composite program. That observation applies equally well to the other two PFOs of L, if-then-else and while-do. If we wish to use if-then-else to build a program from expression b and programs p and q, then p and q must, in most cases, use corresponding input and output cells, otherwise the composite program is likely to be meaningless. Thus again p and q must have a common storage plan.

In contrast, in L* the PFO condition easily assembles appropriate independent programs into a meaningful composite. For example, if b tests whether its argument is an A or a B, and p maps A's into C's and q maps B's into C's, then $b \rightarrow p;q$ is a program that maps $A \cup B$ into C in a way that is evident from its structure.

Again in the case of the program **while** b **do** p the expression b and the program p must have a common storage plan for the composite to be meaningful. The recursive definition in L* corresponding to while-do can, like the other PFOs of L*, easily combine its constituent programs into a meaningful composite.

The PFOs of L* that are not in L also have this important ability to create meaningful programs from existing, independent programs. Thus, for example, from programs p, q and r, all defined on the domain A, the PFO construction builds the program $[p,q,r]$ that maps A into $B \times C \times D$, where B, C, D are the ranges of p, q, r.

It is important to note that the inability of the PFOs of L to assemble independent programs into meaningful new ones comes from the choice of 'stores' as the domain for its programs and the barrier this poses between the purpose of a program and what it actually does: map stores into stores. In L*, on the other hand, there is no such barrier; if the purpose of a program is to map A's into B's, then that is what it does. (Within L* there are 'Von Neumann programs' that map stores into stores, but we do not have to use them.)

The most important property of any system of programming is its ability to build new programs from existing ones at any level. The lack of this ability is the primary weakness of L and the Von Neumann concept of 'program'; having this ability is one of the primary strengths of the L^* concept of 'program'.

4.4. *Complexity of program structure and of language structure*

It is obvious that L^*-programs have a much simpler structure than L-programs. The latter have a three-plane structure (expressions, assignments – the interface between expressions and programs, and programs), each plane having its own entity-forming operation(s). (The structure of real Von Neumann programs is generally more complex than those of L and their domain is usually more complex than 'stores'.)

L^*-programs have a one-plane structure; they are built from primitive ones by PFOs.

The term 'structured programming language' is often applied to languages with PFOs like those of L in which the use of GOTOs is restricted (if it is not, it is hard to define the effect of PFOs). This use of the term is trivial and misleading: if programs are built by any PFOs at all, then they are 'structured' by those PFOs, the only 'non-structured' programs being those that are not built exclusively by well-defined PFOs. But the fact that a language uses PFOs to build programs does not mean that the *language* is structured, only its *programs* are.

If programs are 'structured' by the way they are built by PFOs, then what is a reasonable notion of 'structure' for a language? Since the PFOs of a language are one of its principal elements and since PFOs may be 'structured' by their interrelating algebraic laws, I propose that a programming language should be considered 'structured' to the extent that its PFOs have an algebraic structure. In this sense L and other Von Neumann languages are very weakly structured, whereas the language L^* is strongly structured.

4.5. *The relationship between languages and machines, serial versus parallel*

As outlined earlier, languages like L evolved from the Von Neumann computer and its machine language; this is the basic reason behind the choice of 'stores' as the domain for its programs. Therefore there is a

relatively small 'distance' between *L*-programs and machine programs. Hence it is relatively easy to (a) convert one into the other. (b) project a notion of efficiency from one to the other, and (c) retain the efficiency of one in converting it into the other. But, like their machine counterparts, *L*-programs are hard to transform, to optimize, and to reason about.

The 'distance' between *L**-programs and Von Neumann machine programs is relatively large and therefore it is more difficult to (a) convert one to the other, and (b) project notions of efficiency onto *L**-programs.

The symbiosis between conventional programs and Von Neumann machine architecture, in which each needs the other, has kept the concepts of 'program' and 'computer' all too static over the last 30 years. Originally the Von Neumann concept of machine design was an elegant and beautiful one that matched, in a design of great economy, the economics of circuitry and the object level ideas about programming that where then current. Those basic circuit economics persisted over several generations of new circuitry, so there was little pressure for radical changes in machine design. Now, however, VLSI circuitry has changed those economics and the Von Neumann designs does not seem able to exploit the new economics nearly as well as it did the old.

At the same time, VLSI is making computers so cheap that programming costs are becoming far greater than equipment costs. Ever larger and more complicated Von Neumann languages have been produced in response to the resulting pressure to reduce programming costs, but they have not succeeded in making a satisfactory reduction.

Thus there are twin pressures to find radical new designs that exploit VLSI better and to find radical new languages that reduce the cost of programming so that cheap machines can be cheaply and conveniently programmed. It may be that non-Von Neumann languages such as *L** and others can offer some help in both areas.

There is now a race underway to find new architectures to exploit VLSI to the full. One of the main concerns is to achieve a high degree of parallel operation and to do so without paying the penalty (paid by some earlier parallel designs) of greatly increasing the already soaring costs of programming. On the contrary, it is vital to *reduce* programming costs drastically while at the same time exploiting VLSI. This will require finding parallel designs that are relatively 'close' to new non-Von Neumann concepts of 'program' that appear to offer greater programming power.

Therefore many machine designers are studying non-Von Neumann

languages and attempting to find designs that correspond, that minimize the 'distance' between their design and their chosen language model. Some are looking at 'object level' functional languages, such as LISP, others at 'function level' ones, such as FP [2] or L^* (plus other language models that are hard to classify). (Here is a small sample of papers on representative machine designs: [1,7,8,10,12,15].)

The resulting machine designs vary widely and, until their economics are better understood, we can only wait to see what their cost-performance turns out to be and what the 'distance' is between each design and the various concepts of 'program'. (Of course that 'distance' is short for some designs built around specific languages, e.g., the 'Scheme-79' machine and the Scheme variant of LISP [10], Mago's machine and FP.)

One of the principal challenges in designing machines based on any of the non-Von Neumann languages is to find economical machine techniques to store, manage and operate on data having a hierarchical structure (such as lists or sequences). Another challenge in designing machines based on function level languages like L^* is to implement a variety of PFOs in hardware, thereby making the machine language 'higher level' in some sense than today's 'higher level languages', and thereby helping to make programs for such machines easier and cheaper to produce.

In the effort to find machine designs for parallel operation, languages like L are poor guides. Basically this is because all of its PFOs combine programs in a serial fashion, and having 'stores' as the domain for programs greatly complicates the problems of introducing parallel PFOs. In contrast, the one PFO of L^* that cannot be used in L, construction, is the one that combines L^*-programs in parallel. Thus $[p,q,r]$ is a program all of whose subprograms, p, q, and r, can be applied in parallel to its argument. Other PFOs can be used in L^* that introduce parallel operations, such as 'tree', which serves the same purpose as 'insert' [24]. Thus L^* allows a programmer to naturally express parallel operation where that is called for.

4.6. *Object level and function level programs*

The essence of an object level definition of a function or program is the description, for every possible set of input objects, of how to build a succession of objects (by applying given object-forming operations or given programs) until the desired result-objects have been constructed. But

what are the 'ingredients' from which we build some desired program? They are not the objects we construct in an object level definition. They are in fact the object-forming operations and programs we use in laboriously building the succession of objects that culminates in the 'results'.

In contrast, the function level approach to defining a program is to build it directly from the given 'ingredients', the given operations and programs that must be used in either an object level or a function level definition. Instead of applying the given operations to objects, a function level definition applies functionals (PFOs) to the given operations; instead of building a succession of objects to obtain the 'result-objects', it builds a succession of programs to produce the desired program.

For example, consider the problem of defining a program that transforms x into $\text{sqrt}(x) + \text{square}(x)$, given the object-forming operations (in L) or programs (in L^*) for $+$, sqrt, and square. A program in L is

$$y := \text{sqrt}(x) + \text{square}(x). \tag{1}$$

If this object level program is applied to a store s in which the input number is in cell x, then the result is a store s' in which the result is in cell y. The expression on the right causes sqrt and square to be applied to the number $x:s$, giving two intermediate objects that are added to form the result-object.

The corresponding L^*-program is

$$+ \circ [\text{sqrt}, \text{square}]. \tag{2}$$

This program maps numbers directly into the desired result. When it is *used* it will construct the same intermediate and final results that (1) does:

$$+ \circ [\text{sqrt}, \text{square}] : x = + : \langle \text{sqrt} : x, \text{square} : x \rangle.$$

But (1) and (2) differ in that (1) applies sqrt and square to the ('abstract') object x, giving two *objects,* whereas (2) applies the functional 'construction' to the programs sqrt and square, giving a *program,* [sqrt, square], and so on.

While L-programs focus on combining objects, L^*-programs focus on combining programs. Thus L-programs draw attention to object-forming operations whereas L^*-programs draw attention to PFOs. The algebraic properties of object-forming operations are the basis for general theorems about objects, whereas the algebraic properties of PFOs are the basis for general theorems about programs.

Thus if we want to have a concept of 'program' in which the set of programs themselves, together with some set of PFOs, form an elegant mathematical space for which there are strong, interesting theorems, then it behooves us to pursue a function level approach to the concept of 'program', the approach that is founded on just this viewpoint.

5. Conclusions

Powerful forces now threaten the Von Neumann concept of 'program': The ever greater need to reduce programming costs and the continuing failure of ever more gigantic Von Neumann languages to bring about significant reductions. The drive to find non-Von Neumann architectures that better exploit VLSI. These forces will continue to grow.

It is unclear what new concept of 'program' will emerge in response to these forces, but I think it will become increasingly clear that the answer to the question in the title of this paper is 'yes', that we, computer scientists, need to work hard to develop a new concept of 'program'. And if we are to succeed we need to be aware of and free ourselves from the psychological barriers, the ancient traditions of language that keep us trying to modify the Von Neumann concept based on 'stores' rather than develop something new.

The new concept of 'program' that finally is developed may not turn out to be of the function level kind that I have tried to sketch by describing L^*, but I believe the L^* approach does at least bring out several properties that will be important in any new concept of 'program'. First, and of the most practical and immediate importance, is the ability to combine independent programs to form new, meaningful programs at all levels, and to use a rich set of operations in doing so. This essential element of programming power is just the property that the Von Neumann concept of 'program' lacks, and it is the one that the L^* approach is designed to provide.

But perhaps the most important property for the success of the new concept is that it should make possible an elegant and powerful mathematics of 'programs'. Just as numbers, under the operations of addition and multiplication, form a mathematical system called a ring, so should 'programs', under their program-forming operations, form a mathematical system of a similar kind. Just as there are hundreds of important theorems about numbers and about the solutions of equations in

the ring of numbers, so there should be hundreds of important theorems about 'programs' and about the solutions of equations in the mathematical system of 'programs'. Just as theorems about numbers and their equations represent a great body of deep understanding and knowledge that saves an immense amount of work for mathematicians, so should theorems about the mathematical system of 'programs' contain a deep understanding going far beyond what has already been achieved, and save programmers an immense amount of work.

The Von Neumann concept of 'program' has not given us a powerful mathematics of its programs with a large body of useful general theorems. The best mathematicians and logicians in computer science still struggle hard to prove (often from first principles) that one single program does what is claimed for it. Often they find it necessary to use elaborate programs to help them. One has only to look at the voluminous formal description of any Von Neumann language to realize that its programs do not form a system one can regard as 'mathematical' any more than one can regard a system with many pages of axioms as a useful mathematical one.

I believe that the fundamental reason behind the scarcity of general theorems about conventional programs is the lack of algebraic structure of their program-forming operations. The mutual properties of their PFOs probably guarantee the non-existence of such theorems, just as the mutual properties of addition and square root probably guarantee the lack of interesting general theorems about a system of numbers having only these two operations. But whatever the underlying reasons, it seems time to abandon hope that the Von Neumann concept of 'program' can be the basis of a mathematical system of programs, since 20 years of effort by the best computer scientists have failed to produce the body of general theorems that one expects of such a system.

The work to discover general theorems about function level 'programs' of the sort belonging to L^* has just begun, thus it is too early to predict whether it can produce a large number of theorems, some of which are of general practical importance and others, perhaps, which provide new fundamental insights. But at least the outlook is brighter, since we begin with PFOs having a strong algebraic structure and have the possibility of discovering others that may strengthen it further. The laws relating the principal PFOs of L^* are themselves already useful and very general identities and from them a small number of general theorems have been obtained [2,3,13,22].

The kind of mathematical system represented by L^* is somewhat new in having a relatively large number of operations (PFOs) on a single domain (of L^*-programs) and an even larger number of laws relating these operations. I believe this may turn out to be an exciting new area for study, a relatively unexplored one that invites adjustment, exploration and classification by mathematicians. I believe it is such studies that give the best hope for a mathematics of 'programs', one that can guide us toward the rapid development of a program by using its accumulated knowledge and one that will provide the general tools for concisely proving a program correct and for optimizing it.

Acknowledgements

I am grateful to Steven S. Muchnick for his careful and thoughtful review of an earlier draft of this paper. I owe thanks also to John H. Williams for many helpful discussions on various topic covered in the paper.

References

[1] Arvind, K.P. Gostelow and W. Plouffe. An asynchronous programming language and computing machine, Univ. of calif. (Irvine) Report, Dept. of Information and Computer Science (1978).

[2] J. Backus, Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs, Comm. ACM 21 (8) (1978).

[3] J. Backus, The algebra of functional programs: function level reasoning, linear equations, and extended definitions, Proc. Int. Colloq. on Formalization of Programming Concepts, Peniscola, Spain (April), Lecture Notes in Computer Science, Vol. 107 (Springer-Verlag, Heidelberg, 1981).

[4] J. Backus, Function level programs as mathematical objects, Proc. Conf. on Functional Programming Languages and Computer Architecture, Portsmouth, N.H. (1981).

[5] R. Burstall and J.A. Goguen, The Semantics of CLEAR, A Specification Language, Lecture Notes in Computer Science, Vol. 86 (Springer-Verlag, Heidelberg, 1980).

[6] A. Church, The Calculi of Lambda-Conversion (Princeton Univ. Press, Princeton, NJ, 1941).

[7] J. Darlington and M. Reeve, ALICE: A multi-processor reduction machine for parallel evaluation of applicative languages, Proc. Conf. on Functional Languages and Computer Architecture, Portsmouth, N.H. (1981).

[8] J.B. Dennis, C.K.C. Leung and D.P. Misunas, A highly parallel processor using a data

flow machine language, CSG Memo 134-1, Laboratory for Computer Science, MIT, Cambridge, MA (1974).

[9] J.V. Guttag and J.J. Horning, The algebraic specification of abstract data types, Acta Informat. 10 (1978).

[10] J. Holloway, G.L. Steele, G.J. Sussman and A. Bell, The SCHEME-79 chip, A1 Memo No. 559, Artificial Intelligence Laboratory, MIT, Cambridge, MA (1980).

[11] K.E. Iverson, A Programming Language (Wiley, New York, 1962).

[12] R.M. Keller, G. Lindstrom and S. Patil, An architecture for a loosely-coupled parallel processor, TR UUCS-78-105, Dept. of Computer Science, Univ. of Utah, Salt Lake City, UT (1978).

[13] R. Kieburtz and J. Shultis, Transformation of FP program schemes, Proc. Conf. on Functional Programming Languages and Computer Architecture, Portsmouth, N.H. (1981).

[14] P.J. Landin, The next 700 programming languages, Comm. ACM 9 (3) (1966).

[15] G.A. Mago, A network of microprocessors to execute reduction languages, Int. J. Comput. Informat. Sci. 8 (5 and 6) (1979).

[16] R.E. Milne and C. Strachey, A Theory of Programming Language Semantics (Chapman and Hall, London; Wiley, New York, 1976).

[17] J.C. Reynolds, GEDANKEN – A simple typeless language based on the principle of completeness and the reference concept, Comm. ACM 13 (5) (1970).

[18] D.S. Scott and C. Strachey, Toward a mathematical semantics for computer languages, Techn. Monograph PRG-6, Univ. of Oxford (1971).

[19] J.E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory (MIT Press, Cambridge, MA, 1977).

[20] R.D. Tennent, The denotational semantics of programming languages, Comm. ACM 19 (8) (1976).

[21] J.W. Thatcher, E.G. Wagner and J.B. Wright, Data type specification: parameterization and the power of specification techniques, Proc. Tenth Annu. ACM Symp. on Theory of Computing (1978). New York (1978).

[22] J.H. Williams, On the development of the algebra of functional programs, Techn. Report RJ2983, IBM Research Laboratory, San Jose (1980).

[23] J.H. Williams, Notes on the FP style of functional programming, Lecture notes for the course "Functional Programming and its Aplications", Univ. of Newcastle upon Tyne (1981).

[24] S.N. Zilles, An Introduction to Data Algebras, Lecture Notes in Computer Science, Vol. 86 (Springer-Verlag, Heidelberg, 1979).

Algorithmic Languages, de Bakker/van Vliet (eds.)

Issues in the Design of a Beginners' Programming Language

Lambert Meertens

Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Some problems are related that have been encountered in the design of a programming language for beginners. The solutions were sometimes unexpected, and required doing away with preconceptions. The use of systematic methods has been of some help.

1. Introduction

Of the commonly available algorithmic languages, some are definitely better suited to convey the algorithmic thoughts of the programmer than others. Whatever the preferred point of view, be it structured programming, provability of correctness or the expressibility of abstraction, some languages stand out for their excellence, some for their abomination.

The latter should not worry us for languages in disuse. It should, for languages used widely. The relatively abominable FORTRAN, though far from dead, seems on its way out. Reasonable alternatives for FORTRAN exist. That absolute champion, BASIC, however, is steadily marching on. Moreover, BASIC has it attractive points, from the viewpoint of the casual, non-professional user.

An attempt is under way to redress that situation, by issuing a rival language, provisionally referred to as 'B' (no relation to the precursor of C; the 'B' is only a language-name name referring to the yet unknown language name). For a language to beat a rival, more is involved than language issues. The example of FORTRAN more than goes to show this point. This paper will be restricted, however, to linguistic points. It is not intended as an introduction to B, but as an exposition of some of the choices and problems encountered in the process of designing an algorithmic language. The attempt has been to base the solutions, in a rational way, on the design objectives.

B is designed as the limit of a sequence: B_0, B_1, ... The most recent approximation, B_2, is the joint effort of Robert Dewar of the Courant Institute of Mathematical Sciences, New York University, Leo Geurts of the Mathematical Centre, and the author. Contributions have been made by Peter King of the University of Manitoba, Jack Schwartz of the Courant Institute, and Dick Grune and Paul Klint of the Mathematical Centre. The responsibility for the opinions expressed is solely that of the author.

2. The Design Objectives for B

The idea underlying the design objectives for B are: beat the enemy at its strong points. The same idea has governed the design of ELAN [5]. There is one important difference: ELAN aims primarily at the 'market' of (introductory) education in computer science, whereas B aims first of all at personal computing. The latter has not always been the case. The first approximation of B (see [3]) was designed when personal computing was in its infancy. Although the design objectives themselves have remained the same, their impact on the design has changed quite drastically.

The design objectives for B are:
- simplicity;
- suitability for conversational use;
- inclusion of structured-programming tools.

These objectives are elaborated upon in [3]. The change referred to above is mostly concerned with the objective of simplicity. In [3], this is interpreted as simplicity not only for the user, but also for the implementer. It is stated that "B should be implementable on small minicomputers".

The latter reflects our awareness, at the time, of the onset and future importance of personal computing. At the same time, it reveals a lack of perception of the torrent of hardware evolution. Tomorrow's hand-held computers are yesterday's main-frames. Designing a language to run smoothly on eight bit 8K machines is designing for the past. In designing B_2, it was decided to ignore implementation issues completely. Not that we do not care about implementation complexity; for the time being we have merely disregarded the feelings of prospective implementers and concentrated on the happiness of the user. Once sufficient implementation experience is available, it may be decided to revise features that pose undue

implementation problems in exchange for little or no gain in language appeal. The impact of ALGOL 68R on the revision of ALGOL 68 reveals that this may even help to improve the language from the user's point of view.

3. The Types of B_2

In B_0 and B_1, the types were **INT, REAL, STRING** and **'RANGE'** types (similar to the scalar types of PASCAL), and **ARRAY**s of scalar elements indexed by a compound of **RANGE** values (but without the PASCAL restriction of compile-time fixed bounds). The type system had not really been given much thought, and was the first thing tackled again in the design of B_2.

The type system of B_2 has been designed in a new way that is, in itself, of interest. If a sufficiently powerful collection of types is available (where 'type' includes type constructors as 'array'), any desired type (e.g., deque, or ternary tree) can be 'simulated' or implemented by the user. The type could also be added as a 'standard' type to the language. This may increase the ease of use of the language. Not all types, however, are equally helpful in this respect. Moreover, the language is made more complex, and possibly much so. A type system is competitive only if it is better than each other type system in at least one respect (ease of use, simplicity).

So we compiled a list of candidate types (including, e.g., bag, deque, enumerated types, map, multi-valued map, queue, sequence, set, stack and tree), constructed various schemes for implementing these types in terms of other types, and assigned numerical values for (relative) algorithmic importance and learning complexity of each type and for implementation complexity of each scheme. The values took into account, of course, that the user we have in mind is not a computer scientist. This made it possible, with the assistance of a program, to weed out the non-competitive type systems from the rather large powerset of the candidate types. The resulting list of competitive systems was quite small, and it was easy, using old-fashioned human taste, to settle on one for use in B_2.

If B_1 might be called ALGOL 60 in BASIC-like disguise (the abstract of [3] reads: "FORTRAN : ALGOL 60 = PL/I : ALGOL 68 = BASIC : ?"), B_2 came out like SETL [1] in sheep's clothing. The result is that the types of B_2 are 'number', 'text', 'compound', 'list' and 'table'.

Numbers come in two kinds, 'exact' (i.e., rational) and 'approximate'

(i.e., floating point). The distinction is made at run time. This choice attempts to combine the following desiderata:

(a) The user must be allowed control over quantities that should not be subject to rounding errors. (The choice for rational numbers, rather than integers, is mainly a nicety. But there is some obvious advantage in having, e.g., **1.25**, represent an exact number.)

(b) The user should have no need to worry about the distinction if it is not important. (E.g., adding exact and approximate numbers is allowed.)

(c) The language has strong typing.

(d) Coercions, i.e., automatic implicit type conversions, are deemed undesirable.

(e) Approximateness propagates upwards in evaluating arithmetic expressions.

(This list is not really exhaustive. It implies, among others, the presupposition that there should be some built-in treatment of approximate numbers.)

The approach taken satisfies these five desiderata almost perfectly. Almost ...; in conformance with Murphy's Eighth Law, there is one ugly snag. If **x** is approximate, **x/x** does not equal **1**. For approximateness propagates, and the approximate number **x/x** cannot be equal to the exact number **1**. It is, presumably, equal to the approximate number **~1**.

In fact, no proper solution satisfying the desiderata (a) through (e) exists. As soon as one of these is lifted, a full solution becomes possible. The fact that **1** does not equal **~1** is a violation of (b): sometimes the user does have to worry. We choose this solution because we felt that the user should be careful anyway when comparing approximate numbers and has no business to expect exact answers. Moreover, it is still possible to define the comparison **1 = ~1** to succeed, even though the values are not 'identical'. The solution of allowing one coercion, from exact to approximate numbers (and coercions in its wake on composite values), is still under consideration.

Texts are quite ordinary strings. (The term 'text', instead of the esoteric 'string', was taken from [5].) No character values are provided; a text of length one will do. Two subtext operators are available. If the value of **t** is the sequence of characters $c_1, \ldots, c_n$, then the expression **t@p**, with $1 \leq \mathbf{p} \leq n+1$, stands for $c_p, \ldots, c_n$ and the value of **t|q**, with $0 \leq \mathbf{q} \leq n$, is $c_1, \ldots, c_q$. A common combination will be **t@p|q**. If **t|q^t@(q+1)** is defined ('^' is concatenation), its value is **t**.

These subtext operators may also be applied to text variables in target ('l.h.s.') positions. The replacing text need not have the same length as the text replaced.

Compounds (tuples) are like structured values ('records'), but without tags for selecting the fields. If, e.g., **u** and **v** are variables, then **u,v** may be used in a target position. This allows decomposition of compounds.

Lists exist for values of any type (e.g., list of list of text). A list is simply a multi-set, or bag. In an algorithmic context, given the choice between sets and multi-sets, the latter are more useful. Having both is unnecessarily complex, and even a potential source of confusion. Since we do not expect the user to be familiar with the concept of a multi-set, the semantics are explained in terms of ordered lists. A consequence is that a total ordering has to be defined on the values of any given type. This can be done in a reasonably natural way.

Tables are like SETL maps: generalized arrays whose domain is variable and not necessarily a range of consecutive values. In contrast to SETL, tables are a genuine type, not a syntactic sugaring for interpreting a set of pairs as a map. In particular, a table cannot be a 'multi-valued' map.

Originally, there were many restrictions in this type system. For example, the elements of a compound, list or table could only be numbers or texts. Table keys (indices) were numbers, texts or compounds. Especially the compounds had a special status. Although we thought we had good reasons for these restrictions (at the time the decision to ignore the ease of implementation had not been fully mentally digested), one by one better reasons appeared to relax these constraints. At first, the relaxations tended to make the complexity worse, until we took the step that, in hindsight, seems so obvious: the type system was made completely orthogonal: tables may be indexed with tables, and so on. (This decision nevertheless required reworking most of the provisional language definition.)

As the type system stands now, we are quite pleased by it. The types appear in some way to span together the space of needs, as was the purpose of the exercise. A carefully tamed 'free' type was at some time included, but abandoned later on.

4. Command Syntax

Commands (statements) in B are rather wordy. Each command begins

with a keyword, and keywords are also used to separate the parameters of a command. For example, the following is an assignment command:

```
PUT a+1 IN a.
```

The philosophy behind this approach is given in [3]. An obvious drawback of verbose syntax is that the user has to key in so many symbols. However, as is already stated in [3], the language is embedded in a system that is dedicated to B. In particular, the editor knows the syntax of B. If this is combined with screen-editing facilities, it is possible to reduce the number of key strokes drastically. As soon as the editor knows (or maybe guesses) that a **PUT** command is intended, it may already display the **IN** and position the cursor at the first parameter.

In [4] it is remarked that the keyword approach makes it possible to have user-defined commands. This option has indeed been chosen for B_2. Such command definitions take the role of procedures. For example, the user may define

```
HOW'TO INCR x: PUT x+1 IN x
```

and next use this **INCR** command as though it had been part of the language all of the time.

Since programs are entered through a B-dedicated editor, it is realistic to consider program lay-out as an integral part of the syntax. In particular, indentation is used to indicate grouping of commands. Although this was already so in [3], it took us quite some time to disengage ourselves completely from the idea that programs are prepared on one system and parsed by a second one that need not trust its input. The fact that there is no distinction between editor and parser means that no special delimiters like **BEGIN** and **END** are needed. That **BEGIN** was superfluous, we had already realized; but this was true anyway. But for quite some time, we required **END** lines, as in

```
FOR p IN feasible:
    IF p in cand:
        REMOVE p FROM cand
        INSERT p IN chosen
    END IF
END FOR
RETURN chosen, cand.
```

But the lines with **END** are pure noise. Once one gets used to it, the following is much more legible:

```
FOR p IN feasible:
    IF p in cand:
        REMOVE p FROM cand
        INSERT p IN chosen
RETURN chosen, cand.
```

5. Strong Typing without Declarations

It has been clear from the beginning that B should have strong typing. Not for efficiency reasons, but to aid the user in spotting silly errors as soon as possible. It seemed to us that this calls for declarations revealing the type of identifiers. (The FORTRAN 57 solution of restricting the choice of identifiers for a given type is unacceptable, as is the addition of special symbols as in BASIC.)

One of the attractive features of BASIC is the lack of declarations. Therefore, without really believing in it, we have searched for a system that allows strong typing without declarations. (The advantage of declarations that they provide a redundancy protecting against typos can be taken over by checks against the use of uninitialized variables and warnings for assignment to dead variables.) In some languages with strong typing, it is essential that the type of identifiers is revealed through a declaration. For ALGOL 68, e.g., the value yielded by

```
(amode block = ("abc", "def");
 2 upb block
)
```

is 3 if **amode** is [,] **char**, but 1 if **amode** is [, ,] **char**. But this is clearly a peculiarity. In almost all cases one can reconstruct the types from the context in which identifiers are used.

This has led us to finding a system for B_2 in which it is always possible to reconstruct the type of identifiers from the context. This statement should be slightly weakened in two respects.

The first is that it may be possible to assign types to the identifiers consistently in more than one way. This happens, for example, in

```
PUT {} IN x
IF x={}: WRITE 'yes'.
```

Here **x** could be an empty list of numbers, but it could equally well be an empty list of texts or anything else, or, in fact, an empty table (assuming **x** is not used otherwise). In such cases the net effect is always the same for each type assignment, so we do not care. It also happens in

```
PUT a IN a,
```

if no other assignments to **a** are made. But then **a** is not initialized, which is illegal by itself (and is checked statically).

The second is that commands defined with **HOW'TO** may be truly generic. The definition

```
HOW'TO SWAP a AND b: PUT b, a IN a, b
```

will work for any type, as long as the two parameters have the same type. So no type can be assigned to **a** and **b**. Instead, the requirement is that if **HOW'TO**s are expanded as macros to an arbitrary depth, consistent type assignment remains possible. This raises some hard questions, and undecidability is lurking around the corner [2, 6]. Nevertheless, for B_2 this appears to be decidable without undue restrictions. Only after the last sentence was written down, did the author become aware of the work on type polymorphism by Milner [10]. Although this is described for an applicative language, it appears equally applicable for a language as B. In fact, the situation is simpler there, since the items carrying a polymorphic type are not treated as values in B.

There is one point where an unconventional step had to be taken to uphold the system. If a value comes into being through an operation on other values, it is sufficient if the result type is only dependent on the operand types, which is the case in B_2. We may thus concentrate on the spots where values appear directly. This can happen in two ways.

One is through a constant denotation (literal). This is no problem, since constants in B_2 immediately reveal their types, with one exception: for empty lists or tables. This case has been treated above.

The other case is when a value is obtained through interactive input. There is no a priori way to determine the type. Therefore, it is required that the **READ** command reveal the type of the (expected) input. A first attempt required the presence of a 'type specifier', where the size of the syntax for

specifiers turned out not unsubstantial. This was not very satisfying; it meant the user had to learn a lot of (relatively weird) syntax for this one purpose. Luckily, we found another solution, made possible by the fact that for each value an explicit notation can be given. The type is now specified by providing a 'sample': an expression of the same type. So one has to write, e.g.,

```
READ n, v EG 0, {''}
```

if **n** is a number variable and **v** is a list of texts. (The constant {} will not do in this case.)

6. Formulas

Just like 'procedure calls' and 'commands' are unified in B_2, so are 'function calls' and 'formulas'. A new operator or function is introduced by a **YIELD** unit:

```
YIELD fac n:
    PUT 1 IN f
    FOR i IN {1..n}: PUT f*i IN f
    RETURN f.
```

The compound mechanism gives a natural way to introduce more parameters:

```
YIELD abs (x, y): RETURN sqrt(x*x+y*y).
```

The parentheses are only required since the formal parameter is an explicit compound; the definition might also have run:

```
YIELD abs z:
    PUT z IN x, y
    RETURN sqrt(x*x+y*y).
```

These two definitions are functionally completely equivalent.

For some reason or other, the priorities of operators are a trouble spot in algorithmic languages. An extreme solution as in APL is not attractive; the more so since B_2 is not really expression-oriented. Anyway, it is unacceptable if **2*n+1** really means **2*(n+1)** (although it certainly helps in making the users feel they belong to an esoteric cult). The MABEL solution of re-

quiring parentheses as soon as several operators are involved [7], combines the virtues of simplicity and error resistance. Still, it seems a bit harsh to require parenthesizing of **2*m*n**.

The solution that has been adopted for B_2 is to require parenthesizing whenever the priorities are not established by standing convention and might matter. This is achieved by not assigning simple priorities to operators, but a priority *interval* instead. This interval represents a 'fuzzy' priority. If the precedence decision is independent of the choice of priorities from the intervals, the expression is acceptable. Otherwise, parentheses must be inserted. User-defined operators are always assigned the maximal interval.

Acceptable expressions are, e.g., **m*n/d+c+1**, **a−b+1** and **2*sqrt x**. Unacceptable are **a/2*b**, **a/2/b** and **sqrt 2*x**, to give just a few examples. Of course, the editor warns the user on the spot that parentheses must resolve the ambiguity.

It was a bit surprising that such a simple device as priority intervals could be tuned to give such reasonable results.

7. Generators

Lists are only useful if there is some easy way to step through them. Originally, there were two ways for stepping through a list, one (**OVER alist**) in the normal, and one (**REVO alist**) in reversed order (word play intended). The second form followed an idea from [9], and was connected to the scalar type requirement for table domains in B_0. Once this requirement is relaxed, the convenience of the additional form no longer justifies the extra complexity.

The keyword **OVER** was changed to **IN** for B_2. For example, the command

```
FOR i IN a: INSERT i IN b
```

merges list **a** into **b**. This was done after it had already been decided to allow quantified tests: the test

```
SOME i IN a HAS i<0
```

succeeds if **a** contains a negative element (and sets **i** to stand for the value of the first such element, if any). Instead of **SOME**, also **EACH** and **NO** are allowed.

In B_0, the domain of a table had to be defined as a **RANGE** type in order to create the table. With a dynamic domain, this no longer applies. But there should be some way for the user to go through a table domain. As a first attempt, a domain operator was introduced: **keys t** (during some time written **[]t**) gives the list of keys **i** such that **t[i]** is defined. So we could write:

FOR i IN keys t: . . . .

Switching to a seemingly unrelated topic, we wanted some simple but powerful mechanism for text parsing. A first attempt was a '**FITS** test' of the form

e FITS v_1: t_1, . . . , v_n: t_n,

with **e** a text expression, v_i variables and t_i tests. (The keyword **FITS** keeps appearing and disappearing in the design of B, each time with a different meaning.) The whole test succeeds if an assignment of texts to $v_1, \ldots, v_n$ is possible, such that $e = v_1 \hat{} \ldots \hat{} v_n$ and all of the tests t_i succeed. If several successful assignments were possible, the lexicographically first one would be returned.

Now this would have filled an appreciable part of the syntax for one specialized capability. Moreover, it was unlike anything else in the language. Then we realized that we almost had the capability already there, right under our hands. For the semantics were exactly those of

SOME $v_1, \ldots, v_n$ IN ??? HAS (t_1 AND $\cdots$ AND t_n),

provided some suitable expression for the **???** could be substituted. This expression should be a list of all compounds $s_1, \ldots, s_n$ such that $e = s_1 \hat{} \cdots \hat{} s_n$. A provisional notation for this list was **e/n** (**e** divided in **n** parts). This raises the problem that the type of **e/n** is dynamically dependent on **n**, which is incompatible with strong typing. If the form were only allowed in this context, the problem would disappear; in fact, the **n** is then redundant, since there are exactly **n** bound variables.

This triggered the solution adopted now. It is illustrated by the following example:

```
WHILE SOME h, s, t PARSING sent HAS s = ',':
    INSERT h IN words
    PUT t IN sent.
```

If **sent** contains a comma, the parsing will be found that positions **s** at the first comma (so **h** will not contain a comma). If **sent** does not contain a comma, the test fails. If **sent** originally held the text **'hickory,dickory,dock'**, the effect is that of

```
INSERT 'hickory' IN words
INSERT 'dickory' IN words
PUT 'dock' IN sent.
```

This is the most complicated feature in B_2; it is, however, quite powerful. Its semantics can be explained in already familiar terms. At the same time, it takes away the nagging problem that a simple command as

```
PUT 'memory is becoming cheap'/24 IN m
```

threatens to blow up even gigabyte systems.

When **OVER** and **REVO** were originally introduced, and when they were replaced by **IN**, we did not think of the construction as a generator. With **PARSING**, we clearly have a generator. It is quite natural then to have a generator **INDEXING** to go through all keys of a table. For example,

```
PUT 0 IN s
FOR i INDEXING t: PUT s+t[i] IN s
```

sums the elements of **t**.

Such a decision may seem simple. But it has many ramifications. One is that the function **keys** should be abolished. Inspection of programs shows that in practice it is never used in a command like

```
PUT keys t IN kt.
```

But the function is used in other ways, such as

```
PUT min keys t IN mt,
```

which finds the smallest key in the domain of **t**. The meaningful test

```
i in keys t
```

would also have to be replaced by some new notation. Instead, it was decide to leave **keys** alone, not to introduce **INDEXING**, but to generalize **FOR ... IN ...** to iterate also over the characters of a text and the elements of a table. Summing the elements of a table may thus be written:

```
PUT 0 IN s
FOR e IN t: PUT s+e IN t.
```

The same generalization applies, of course, to **SOME ... IN ...**, but also to all functions and tests previously only defined on lists (such as **min** and **in**).

8. The final composition

As has been clear from the exposition, composing a language is not merely a matter of putting ingredients together and stirring till the result is a smooth paste. It would be helpful to language designers, if some top-down design method existed for algorithmic languages. If such a method exists, it has escaped our attention. The requirement for applying a method as 'separation of concerns' is that the relevant concerns be separable. The whole experience of language design points in a different direction: apparently innocent minor decisions may quite unexpectedly work major havoc in seemingly unrelated corners. A well-composed language is one in which the 'features', although orthogonal, lend themselves to easy combination in many natural modes of expressing algorithmic thought. This means that the whole language is a tightly knit fabric, threatened by loose ends.

The best aid to systematic language design, until now, is the paradigm of orthogonality, that derives its name from the title of Van Wijngaarden's [14], but whose essence can already be found in his [13]. Experience shows that its application requires skill, if not expertise. It is interesting to see that the evolution of B has been in the direction of more orthogonality, mainly by virtue of the quest for simplicity.

For part of the work in designing B_2, a new systematic approach has been used: the method described in Section 3 to select the type system. This method is more widely applicable; it can be used, e.g., to find a proper system of string operations from a large set of candidates. Work is in progress to apply another systematic method for the final polishing of the whole language.

The idea has been used before by the author in a composition exercise of a different nature: composing a string quartet with traditional harmony [8]. The same idea is applicable here. In its bare essence, it boils down to

considering all combinations of all alternatives for the microscopic design decisions. For each combination, a check list is inspected of potential unacceptable or undesirable consequences. For each transgression, a fine is imposed. The combination that collects the minimal total fine, comes out as the winner.

The method is, of course, NP-complete. In practice, however, it is expected to be feasible with the aid of some heuristics, since many design decisions form relatively independent small clusters. Still, this computational complexity is indicative of how hard it is to design a language. The example of the five reasonable desiderata for the numbers, only four of which could be satisfied simultaneously, is just one example of the problems a language designer may run across.

It would be misleading to call such methods 'language design by computer'. The real skill goes into identifying the decisions, weighing the importance and merits of various approaches, and identifying harmful combinations. Only a dumb, but hard, part of the work is left to brute force. It is expected that the first-time 'winner' will mainly serve to show deficiencies in the input to the program, and that several iterations will be needed to come up with a nice product. Indeed, the exercise may point out directions we have overlooked. If anything, the method requires that human prejudice is made explicit. The algorithm itself is, like Justice, blind-folded.

References

[1] R.B.K. Dewar, The SETL programming language, Courant Institute of Mathematical Sciences, New York University (1980).

[2] N. Gehani, Generic procedures: an implementation and an undecidability result, Comput. Languages 5 (1980) 155–161.

[3] L.J.M. Geurts and L.G.L.T. Meertens, Designing a beginners' programming language, in: S.A. Schuman (Ed.), New Directions in Programming Languages 1975 (IRIA, Roquencourt, 1976) pp. 1–18.

[4] L.J.M. Geurts and L.G.L.T. Meertens, Keyword grammars, in: J. André and J.-P. Banâtre (Eds.), Implementation and Design of Algorithmic Languages (IRIA, Rocquencourt, 1978) pp. 1–12.

[5] G. Hommel, J. Jäckel, S. Jähnichen, K. Kleine, W. Koch and K. Koster, ELAN – Sprachbeschreibung (Akademische Verlagsgesellschaft, Wiesbaden, 1979).

[6] H. Langmaack, On correct procedure parameter transmission in higher programming languages, Acta Inform. 2 (1973) 110–142.

[7] P.R. King, MABEL manual, University of Manitoba (1978).
[8] L.G.L.T. Meertens, The imitation of musical styles by a computer, in: Information Processing 68, Proc. of IFIP Congress 1968, Vol. 1 (North-Holland Publ. Co., Amsterdam, 1968) pp. xxv–xxvi.
[9] L.G.L.T. Mode and meaning, in: S.A. Schuman (Ed.), New Directions in Programming Languages 1975 (IRIA, Roquencourt, 1976) pp. 125–138.
[10] L.G.L.T. Meertens, Preliminary draft proposal for the B programming language, Mathematical Centre, Amsterdam (May 1981).
[11] R. Milner, A theory of type polymorphism in programming, J. Comput. System Sci. 17 (1978) 348–375.
[12] K. Tracton, 57 Practical Programs and Games in Basic (Tab Books, Blue Ridge Summit, 1978).
[13] A. van Wijngaarden, Generalized ALGOL, in: Symbolic Languages in Data Processing, Proc. of an ICC Symp. (Gordon and Breach, New York, 1962) pp. 409–419; also in: R. Goodman (Ed.), Annual Review in Automatic Programming, Vol. 3 (Pergamon Press, Oxford, 1963) pp. 17–26.
[14] A. van Wijngaarden, Orthogonal design and description of a formal language, Report MR 76, Mathematical Centre, Amsterdam (1965).

Appendix A: a B_0 and a B_2 Program for the Sieve of Eratosthenes

The following B_0 program is copied from [3].

```
BEGIN
CONST n IS 1999
RANGE sievesize FROM 2 TO n
RANGE primality HAS prime, nonprime
ARRAY (sievesize) a TYPE primality
FOR i OVER sievesize PUT prime IN a(i)
VAR k TYPE int, kmult TYPE sievesize
PUT 2 IN k
WHILE k*k FITS kmult
  BEGIN
  VAR k1 TYPE sievesize
  IF k FITS k1, a(k1) = prime DO sieve
  PUT k+1 IN k
  END
sieve:
  BEGIN
  PUT nonprime IN a(kmult)
```

```
    WHILE kmult + k FITS kmult PUT nonprime IN a(kmult)
    END
  FOR i OVER sievesize
  IF a(i) = prime
    BEGIN
    NEWLINE
    PRINT i
    END
  END
```

This problem was certainly not selected in [3] to show the cluminess of B_0. The algorithmic thought is captured more easily, though, in B_2:

```
HOW'TO SIEVE n:
  PUT {2..n}, 2 IN primes, k
  WHILE k*k⇐n:
    PUT k*k IN kmult
    WHILE kmult⇐n:
      IF kmult in primes: REMOVE kmult FROM primes
      PUT kmult + k IN kmult
    PUT k min primes IN k
  WRITE primes
SIEVE 1999
```

Note that this program is algorithmically slightly different from the B_0 program given above. The formula **k min primes** yields the smallest element of the list **primes** exceeding **k**.

Appendix B: a BASIC and a B_2 Program for Tabulating a Recurrent Sequence

The following program is copied from [12]. It has been selected because for this problem none of the 'strong' points of B_2, such as manipulation of lists, apply. For purposes of fair comparison, non-keywords have been rendered in lower case.

```
10 REM This program computes a table of Fibonacci
   numbers
20 PRINT 'Enter first term'
```

```
 30 INPUT a
 40 PRINT 'Enter second term'
 50 INPUT b
 60 PRINT 'Maximum number of terms ='
 70 INPUT n
 80 PRINT
 90 PRINT 'Table of Fibonacci numbers'
100 PRINT 'Term no.','Fibonacci number'
110 LET k=1
120 PRINT k,a
130 LET k=2
140 PRINT k,b
150 LET k=k+1
160 LET q=a+b
170 PRINT k,q
180 LET a=b
190 LET b=q
200 IF k>= n THEN 220
210 GOTO 150
220 PRINT 'Maximum numbers of terms reached'
230 PRINT
240 PRINT 'Type 1 to continue, 0 to stop'
250 INPUT /
260 IF /=1 THEN 280
270 STOP
280 PRINT
290 GOTO 20
300 END
```

The following B2 program is not an exact transliteration; it contains an obvious improvement that might also be applied to the BASIC version. As to the question if this is fair in making a comparison, it should be considered that part of the thesis motivating the development of B is that BASIC invites clumsy programming.

```
HOW'TO TABULATE'FIBONACCI'NUMBERS:
  PUT 'yes' IN cont
  WHILE cont|1='y':
    WRITE / 'Enter first term: '
```

```
READ a EG 0
WRITE / 'Enter second term: '
READ b EG 0
WRITE / 'Maximum number of terms='
READ n EG 0
WRITE // 'Table of Fibonacci numbers'
WRITE / 'Term no.  Fibonacci number'
FOR k IN {1..n}:
  WRITE / k>>5, a>>15
  PUT k+1, b, a+b IN k, a, b
WRITE / 'Maximum number of terms reached'
WRITE / 'Do you want another table? '
READ cont EG ''
```

This program shows some 'formatting': the formula **x>>n** yields a text of length **n** representing the value of **x**, right adjusted (left-padded with blanks).

Algorithmic Languages, de Bakker/van Vliet (eds.)

From VW-grammar to ALEPH

D. Grune

Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

This paper gives an exposition of the designing of ALEPH. ALEPH (acronym for A Language Encouraging Program Hierarchy) is a programming language developed at the Mathematical Centre; it is unusual in that it originates from the world of grammars rather than from the world of programming languages. It has the interesting property that it is large enough not to be dismissed as a toy language and small enough to keep the task of designing it intellectually manageable.

An account of the design of ALEPH is interesting not only because of its results, a language with a very simple but powerful flow-of-control in which the uninitialized-variable problem is solved and in which side effects are under full control, but also because the way in which these results are obtained lies open to examination.

1. Introduction

ALEPH (acronym for *A L*anguage *E*ncouraging *P*rogram *H*ierarchy) [6] is a programming language developed at the Mathematical Centre; it is unusual in that it originates from the world of grammars rather than from the world of programming languages. It has the interesting property that it is large enough not to be dismissed as a toy language and small enough to keep the task of designing it intellectually manageable (although barely so).

Therefore an account of the design of ALEPH is interesting not only because of its results, a language with a very simple but powerful flow-of-control in which the uninitialized-variable problem is solved and in which side effects are under full control, but also because of the fact that the way in which these results are obtained lies open to examination.

In this paper we shall give an exposition of the designing of ALEPH. Little is known about design rules for programming languages. In essence design rules serve to reduce the intellectual complexity of a task. Traditional

means are: imposing a structure, divide-and-conquer, defining interfaces, etc. Hardly any of these applies to the design of programming languages. The most successful principle is still orthogonality, which also has its problems. It does not allow the designer to distinguish between the cheap and the expensive, and its consistent application is difficult.

1.1. *Vocabulary*

Our discussion leads us from VW-grammars through affix grammars to ALEPH and conventional programming languages. A VW-grammar (2.1) can be seen as a recipe for generating an (infinite) grammar capable of generating the context-sensitive language we want. An affix-grammar (2.4) can be seen as a parametrized context-free grammar where the context is stored in the parameters (affixes).

Different terminology is (traditionally) used in these different fields, and it may be helpful for the reader to refer to Table 1.

Table 1

VW-grammars	Affix-grammars	ALEPH	Conventional programming languages
grammar	grammar	program	program
–	initial symbol	root	–
hyper-rule	rule	rule	procedure
–	primitive predicate	external rule	built-in function
	left-hand-side, LHS	rule head	procedure heading
	right-hand-side, RHS	rule body	procedure body
may produce empty	may produce ε	always succeeds	always yields **true**
is a blind alley	produces ω	fails	yields **false**
hypernotion	affix expression	affix form, rule call	call
metarule	affix rule	–	data type
metanotion	affix	affix	parameter
	bound affix	formal affix	formal parameter
	free affix	local affix	local parameter

2. Turning a VW-grammar into a Programming Language

2.1. *VW-grammars*

A VW-grammar (named after its originator, A. van Wijngaarden [9, 10]) is a special type of context-sensitive (CS) grammar which has many properties of a context-free (CF) grammar. It is based on the observation that we can use a CF grammar to describe a CS language, provided that this grammar has infinitely many production rules; every actual production of a desired sentence in the CS language, however, needs only a finite number of them. In essence a VW-grammar is a recipe for generating such an infinity of CF production rules. For an extensive explanation see [2].

A VW-grammar has the following main constituents:

- the metarules, a collection of (interrelated) CF grammars, each producing a language for a specific metanotion,
- the hyper-rules, a collection of templates from which to form (an infinity of) CF production rules.

A CF production rule is derived from a hyper-rule by replacing consistently each of the metanotions it contains by a terminal production of that metanotion. For an example see TCGl below.

2.2. *Two-colour grammars*

Let me now introduce the notion of a 'two-colour' VW-grammar. We start from a VW-grammar R, which produces sequences of symbols in red. We then take a second VW-grammar P, which shares part or all of its metarules with R and which produces its symbols in blue (or in a different alphabet if you will). We now combine the two grammars and insert hypernotions of P in hyperalternatives of rules of R: the resulting grammar produces sentences in mixed red and blue text.

If it now so happens that a hypernotion of P shares one or more metanotions with its neighbours that belonged to R, then the production of blue text is controlled by the same choice of metanotion substitutions as that of the red text, and the red and blue pieces of text will become correlated.

As an example we shall now rewrite grammar Q from [2, p. 64] as a two-colour grammar.

TCG1:

```
N   :: N n;  .
ABC :: a; b; c.

text: red N a, blue N b, blue N c.

red N ABC:
    red symbol ABC, red N1 ABC, where rd N1 plus one is N;
    where rd N is zero.
red symbol ABC: red letter ABC symbol.
where rd N plus one is N n: where true.
where rd    is zero: where true.

blue N ABC:
    where bl N is zero;
    blue symbol ABC, where bl N1 is N minus one, blue N1 ABC.
blue symbol ABC: blue letter ABC symbol.
where bl N is N n minus one: where true.
where bl    is zero: where true.

where true:  .
```

A possible production is (with N = nnn in 'text'):

red-a red-a red-a blue-b blue-b blue-b blue-c blue-c blue-c.

2.3. *A top-down parser*

It is well known that a CF grammar can be turned into a recognizer for the language it produces. In the case of an unrestricted CF grammar such a recognizer has to do extensive backtracking, which is painful in terms of space and time, but if enough restrictions are put on the CF grammar, neat recognizers result. Specifically, the LL(1) restriction leads to an efficient top-down parser, which, as a program, has virtually the same form as the original grammar.

This suggests that it may be possible to consider the red part of the two-colour grammar TCG1 (which, in a sense, *is* LL(1)) as a top-down parser for the red text, while at the same time retaining the producing nature of the blue part. If we do this, we are led to consider the occurrences of metanotions in hypernotions as parameters. We shall not worry at the

moment about the exact parameter-passing mechanism; for the time being it can be thought of as 'call-by-name'. This brings us to the following grammar/program:

P1:

text: read N a, print N b, print N c.

read N ABC:
    read symbol ABC, read N1 ABC, where rd N1 plus one is N;
    where rd N is zero.
read symbol ABC: absorb letter ABC.
where rd N1 plus one is N: set N to N1 plus one.
where rd N is zero: set N to zero.

print N ABC:
    where pt N is zero;
    print symbol ABC, where pt N1 is N minus one, print N1 ABC.
print symbol ABC: produce letter ABC.
where pt N1 is N minus one: set N1 to N minus one.
where pt N is zero: is N zero.

When we read this with the firm conviction that it is a program, semantics begins to attach itself to various constructs. To perform 'text', read N a's, then print N b's, then print N c's. To read N ABC's, we have the choice between two alternatives which we shall try in order. We attempt to read a symbol ABC, and if we succeed we read N1 ABC's and set N to N1 plus one; otherwise (if we cannot read a symbol ABC) we set N to zero. In this same vein we can understand the rest of the program, which prints N b's and N c's.

At this point the reader will have gathered that we have cheated. The above example was rigged so that its interpretation as a program suggested itself. A general VW-grammar does not exhibit such a nice structure, and the parsing problem cannot in general be solved. There is, however, a type of CS grammar related to VW-grammars for which the parsing problem *can* be solved: the affix grammars.

2.4. *Affix grammars*

Affix grammars are defined by Koster [7]; this definition is slightly

corrected and explained well in [1]. Koster shows that if an affix grammar is 'well-formed' (see below) it is possible to construct a parser for the language it generates. Most constituents of a VW-grammar also exist in an affix-grammar. For a list of correspondences see Table 1. The principal differences between affix grammars and VW-grammars are:

- a hypernotion consists of a characteristic name, its 'handle', followed by one or more metanotions, called 'affixes', and
- context conditions are enforced by special rules called 'primitive predicates'; they can be thought of as affix checkers.

A 'primitive predicate' is similar to a (normal) rule in that it has affixes; but rather than producing its output by specifying affix forms and terminal symbols, it contains a total recursive function T which, depending on the affixes, will produce either 'empty' (ε) or the forbidden symbol (ω). We shall call T the 'test' of the primitive predicate.

The well-formedness criterion requires (among other things) that all occurrences of affixes be divided into two groups, the 'derived' (δ) and the 'inherited' (ι) affixes, in such a way that they can properly be interpreted as output and input parameters, respectively. Moreover, for each primitive predicate with derived affixes D, inherited affixes I and test T, a total recursive function must be given which will calculate D from I such that $T(I, D)$ succeeds (i.e., produces ε); this requirement marks the transition from a specification language to an algorithmic language.

We shall now show an affix-grammar equivalent to TCG1 (some comment is given between {{ and }}):

AG1:

```
⟨{{V[n]:}} (text, red, red symbol, blue, blue symbol),
 {{V[t]:}} (red-a, red-b, red-c, blue-a, blue-b, blue-c),
 {{A[n]:}} (N, N1, ABC, ABC1),
 {{A[t]:}} (n, a, b, c),
 {{Q:}} (where rd plus one is, where rd is zero, where is,
         where bl is minus one, where bl is zero
        ),
 {{E:}} text,
 {{R:}} (N: N n; .      ,
         N1: N.         ,
         ABC: a; b; c. ,
         ABC1: ABC.
        ),
```

{{S:}} (⟨text, 0, ϕ, ϕ, ϕ⟩,
⟨red, 2, (δ, ι), (N, ABC), ϕ⟩,
⟨red symbol, 1, (ι), (ABC), ϕ⟩,
⟨where rd plus one is, 2, (ι, δ), (N, N1),
$\lambda x\ \lambda y$: ($x+1=y\rightarrow\varepsilon$, $x+1\neq y\rightarrow\omega$)⟩,
⟨where rd is zero, 1, (δ), (N),
λx: ($x=0\rightarrow\varepsilon$, $x\neq 0\rightarrow\omega$)⟩,
⟨where is, 2, (ABC, ABC1), (ι, ι),
$\lambda x\ \lambda y$: ($x=y\rightarrow\varepsilon$, $x\neq y\rightarrow\omega$)⟩,
⟨blue, 2, (ι, ι), (N, ABC), ϕ⟩,
⟨blue symbol, 1, (ι), (ABC), ϕ⟩,
⟨where bl is minus one, 2, (ι, δ), (N, N1),
$\lambda x\ \lambda y$: ($x=y-1\rightarrow\varepsilon$, $x\neq y-1\rightarrow\omega$)⟩,
⟨where bl is zero, 1, (ι), (N),
λx: ($x=0\rightarrow\varepsilon$, $x\neq 0\rightarrow\omega$)⟩
),
{{P:}} (text: red + N + a, blue + N + b, blue + N + c.
red + N + ABC:
red symbol + ABC, red + N1 + ABC,
where rd plus one is + N1 + N;
where rd is zero + *N*.
red symbol + ABC:
where is + ABC + a, red-a;
where is + ABC + b, red-b;
where is + ABC + c, red-c.
blue + N + ABC:
where bl is zero + N;
blue symbol + ABC, where bl is minus one + N1 + N,
blue + N1 + ABC.
blue symbol + ABC:
where is + ABC + a, blue-a;
where is + ABC + b, blue-b;
where is + ABC + c, blue-c.
)
⟩

To satisfy the well-formedness requirement this text must be augmented by a list of functions, one for each primitive predicate, that calculate the

derived affixes from the inherited ones. Since lambda-notation does not allow output-parameters, these functions cannot be written down here. They correspond to the "set N to ···" in P1.

3. From Affix Grammar to ALEPH

Although the affix grammar AG1 can be converted easily into a program, it will be clear that affix grammars are still a far cry from a usable programming language. We have 'primitive predicates' which form a kind of language inside the language. The global flow-of-control may be obvious but details about the local flow-of-control (i.e., inside a rule) have to be decided. The exact nature of affixes is open to negotiation. The affix rules describe data structures, but their form will depend on decisions about the affixes.

There are of course many ways to approach these problems. One such approach has led to the Compiler Description Language CDL, designed by Koster [8], and its successor CDL2 [4]. We shall follow here a different way which leads to ALEPH.

Like in CDL we shall restrict ourselves to top-down (recursive descent) parsers, since they lead more easily to programming languages than bottom-up parsers. Bottom-up parsers for affix grammars have been constructed by Crowe [3] and Böhm [1].

3.1. *Global flow-of-control*

The global flow-of-control relies completely on rules calling rules (recursively); since there is only one level of rules and rules cannot occur as parameters (nor be assigned to 'rule variables'), the program is a directed graph; the starting point is the **root**. This has the great advantage that many properties of the program can be decided mechanically (recursion check, automatic cross-referencing). On the other hand it means that the rule-calling and affix-passing mechanism will be used heavily and that efficiency will be an important factor in the design of both.

3.2. *Finding a place for the primitive predicates*

We shall incorporate the ι/δ affix information in the rule heads; an ι-affix (input affix) is marked by a *pre*fixed $\rangle$, a δ-affix (output affix) by a

*post*fixed ⟩. We shall postpone the decision about the affix-passing mechanism to Section 4.1.

The number of primitive predicates can often be greatly reduced by describing their effect (producing ε or ω) in hyper-rules. Many full-size examples of this technique can be found in [10, Ch. 7] and in [5]. This suggests the possibility of using a fixed set of metarules for every grammar, i.e., to supply a fixed set of data-types in the programming language. These data-types are then supported by a predefined set of predicates on them, the 'externals'.

The RHS of a rule may contain both affix forms and terminal symbols; we shall simplify this situation by introducing two rules, 'absorb + ABC' and 'produce + ABC'. 'Absorb + ABC' looks at the next character in the input stream; if it is equal to ABC, 'absorb' absorbs it and succeeds; otherwise it fails. 'Produce + ABC' produces the character ABC. They replace the absorption and production mechanism implied in the functioning of a two-colour grammar.

Our program now has the form (character constants are quoted with /'s):

```
P2:

root text.

external set to plus one + N⟩ + ⟩N1 = 'INCR',
    set + ⟩N + N1⟩ = 'SET',
    set to minus one + N⟩ + ⟩N1 = 'DECR',
    equal + ⟩N + ⟩N1 = 'EQUAL'.

text: read + N + /a/, print + N + /b/, print + N + /c/.

read + N⟩ + ⟩ABC:
    read symbol + ABC, read + N1 + ABC, where rd plus one is + N1 + N;
    where rd is zero + N.
read symbol + ⟩ABC: absorb + ABC.
where rd plus one is + ⟩N1 + ⟩N: set to plus one + N + N1.
where rd is zero + N⟩: set + 0 + N.

print + ⟩N + ⟩ABC:
    where pt is zero + N;
    print symbol + ABC, where pt is minus one + N1 + N,
      print + N1 + ABC.
```

print symbol +)ABC: produce + ABC.
where pt is minus one + N1) +)N: set to minus one + N + N1.
where pt is zero +)N: equal + N + 0.

end

Note that characteristic strings have been supplied in the **external** declarations, which enable the compiler to find the proper routines outside the program.

3.3. *Local flow-of-control*

Local flow-of-control is the flow-of-control inside a rule once it is called due to global flow-of-control rules. Since global flow-of-control is trivial, we shall use simply 'flow-of-control' for 'local flow-of-control'.

The parsing problem for affix grammars can be solved by a general top-down parser [7, par. 8], at the expense of extensive back-tracking. Now ALEPH is intended for the writing of production soft-ware; here any back-track problems should be solved once at the writing desk, rather than over and over again when the program is run. A traditional way to avoid back-tracking is to require the grammar to be of type LL(1).

What does it mean for an affix grammar to be LL(1)? It should be borne in mind that the LL(1)-property is important only because it allows simple flow-of-control rules for a backtrack-free deterministic parser. We shall therefore take these rules as a starting point:

LL(1) rules:
- call the initial rule; iff it succeeds, the input belongs to the language;
- a rule is 'called' by trying the alternatives in its RHS for applicability and calling an applicable alternative (there can only be one such alternative);
- an alternative is 'applicable' iff its first rule call succeeds;
- an alternative is 'called' by calling its rules in textual order as long as these rule calls succeed;
- an alternative 'succeeds' iff all of its rule calls succeed;
- a rule call 'succeeds' iff the rule called has an applicable alternative that succeeds.

Moreover we have an error condition:
- if any applicable alternative fails, the input does not belong to the generated language (i.e., if an alternative is applicable it is the correct one).

We want to take over these rules as much as possible. After some experimentation we have come to the following flow-of-control rules:

ALEPH rules:
- execute the affix form in the **root**; it must succeed;
- an affix form is 'executed' by trying the alternatives in the RHS of its rule for applicability and executing the first applicable alternative;
- an alternative is 'applicable' iff its first affix form succeeds;
- an alternative is 'executed' by executing its affix forms in textual order as long as these affix forms succeed;
- an alternative 'succeeds' iff all of its affix forms succeed;
- an affix form 'succeeds' iff the rule called has an applicable alternative that succeeds.

These flow-of-control rules allow us to view the first affix form as an 'entrance key': you enter the first alternative to which you have the right key. Once you enter this alternative no others can be reached any more. An important consequence is that there is only one way to reach a given affix form. This leads immediately to the Central Theorem of ALEPH:

Central Theorem. *When the* Nth *affix form in the* Mth *alternative is reached, the entrance keys of alternatives* 1 *through* $M-1$ *have failed, and affix forms* 1 *through* $N-1$ *in this alternative have succeeded.*

This Central Theorem is a great help in deriving assertions (see below).

We still have to investigate the error condition inherited from the LL(1) flow-of-control rules; we shall postpone this until Section 3.5.

The above rules are (almost) all the flow-of-control ALEPH has: there are no **case**-, **while**-, **do**-, **repeat**-, **until**-, or **exit**-clauses. Rather than emphasizing repetition, ALEPH emphasizes decomposition: each problem is decomposed into several alternatives with entrance keys and each alternative is decomposed into a sequence of sub-problems (which may, of course, be congruent to the original problem). In short, every problem is attacked by recursive descent.

Often a problem that requires a complicated application of the traditional **if**'s and **while**'s can be formulated simply in ALEPH. A good example is searching a list for a given name; the search process stops in one of two ways; the list is empty, or we found the name. We want to do different things in both cases. Here we would need a multi-exit loop or a global toggle; or we would have to perform the same test twice. In ALEPH we simply state the alternatives and tell what to do:

```
find name + >name + >list + entry>:
    is empty + list, insert + name + list + entry;
    is name on top + name + list, top of + list + entry;
    next of + list + list1, find name + name + list1 + entry.
```

3.4. *Success/failure*

We have assumed in the above that any rule can fail (but we have not based any conclusions on that). It soon becomes clear, however, that some rules cannot fail, e.g., because a rule produces ε regardless of the values of its affixes.

The Central Theorem shows us immediately that if any alternative but the last one in a rule has an entrance key that cannot fail, part of the RHS is inaccessible.

3.5. *Side effects*

It is the error condition for LL(1)-parsing in Section 3.3 that allows us to avoid back-tracking, in the following way. When a rule call fails, it has only called other rules that failed. Now since the only terminal rule is 'absorb', and since 'absorb' has no side effect when it fails (Section 3.2), no rule call that fails will have had side effects (by induction). So nothing is modified on failure, and no back-track is necessary. This is the 'No cure – no pay' principle: you may order something, but if you don't get it, you don't pay.

We would certainly like to carry this nice feature of LL(1) parsing over into our programming language. This is done trivially by forbidding any applicable alternative to fail (either statically or dynamically). But we can do better than this.

Where a CF grammar only has rules (which have side effects on success), we have rules (which also have side effects on success) *and* primitive predicates (which never have side effects). Moreover, some of our rules derive entirely from primitive predicates (see Section 3.2). So in ALEPH a successful affix form does not necessarily imply side effects.

Consequently it is perfectly safe to allow failure of an applicable alternative, provided no affix form with side effects has yet succeeded in the alternative.

Under this regime the 'No cure – no pay' principle holds:

If an affix form fails, it has had no side effects.

In Section 3.4 we have divided the rules into two groups, those that can fail and those that can't. Now we have a second division, in those that can have side effects (on success) and those that can't. These divisions are independent, so four classes (rule types) result:

	can fail	cannot fail
can have side effects	**predicate**	**action**
cannot have side effects	**question**	**function**

This classification allows us to give a proper place to 'absorb' and 'produce': their rule types are **external predicate** and **external action**, respectively.

In principle the compiler could assess these properties, but it is much more useful to have the programmer specify his intentions (opinions) and have the compiler check them. The non-trivial redundancy obtained is exploited for error detection.

Our program is now (affixes are written in small letters):

P3:

root text.

external function set to plus one + n⟩ + ⟩n1 = 'INCR',
function set + ⟩n + n1⟩ = 'SET',
function set to minus one + n⟩ + ⟩n1 = 'DECR',
question equal + ⟩n + ⟩n1 = 'EQUAL',
predicate absorb + ⟩abc = 'ABS',
action produce + ⟩abc = 'PROD'.

action text: read + n + /a/, print + n + /b/, print + n + /c/.

action read + n⟩ + ⟩abc:
read symbol + abc, read + n1 + abc, where rd plus one is + n1 + n;
where rd is zero + n.
predicate read symbol + ⟩abc: absorb + abc.
function where rd plus one is + ⟩n1 + ⟩n: set to plus one + n + n1.
function where rd is zero + n⟩: set + 0 + n.

```
action print + )n + )abc:
        where pt is zero + n;
        print symbol + abc, where pt is minus one + n1 + n,
          print + n1 + abc.
action print symbol + )abc: produce + abc.
function where pt is minus one + n1) + )n: set to minus one + n + n1.
question where pt is zero + )n: equal + n + 0.

end
```

We see the impact the rule type classification has on the program: for each rule it is locally clear what to expect of it in terms of flow-of-control. The consistency of the indications is checked by the compiler; we have here strong type checking, not for data types but for rule types.

As with strong type checking on data the errors detected originate from inconsistencies on behalf of the programmer. Suppose there is a rule *'xyz'* which has ε as one of its alternatives and which is used for testing the presence of an *'xyz'*. Now, if *'xyz'* is declared as a **predicate**, the empty alternative will cause an error message, and if it is declared as an **action**, its use as a test will be noticed.

4. Affixes

Rules in an affix grammar can have bound affixes (those that occur in the LHS and in the RHS) and free affixes (that occur in the RHS only). In ALEPH these correspond to formal and local affixes, or 'formals' and 'locals'. There are 'input' and 'output' formals; an input formal has a value upon entry to the rule an output formal must have received a value when the rule ends.

Of course it is necessary that all input affixes of an affix form have obtained a value when the affix form is executed. Now, since

- the Central Theorem states that there is only one path from rule entrance to a given affix form, and the Central Theorem gives that path;
- the initial states of all formals and locals at rule entrance are known from the LHS; and
- for each affix form A on the path the effect on the affixes passed to it is known from the LHS of A,

the compiler can ascertain in an efficient way that never the value of an affix will be used before that affix has received a value. No run-time checking is necessary. A similar test can ensure that an output formal will always receive a value.

The details of this test depend on the affix-passing mechanism.

4.1. *The affix-passing mechanism*

The affix-passing mechanism has to obey two conditions: the value of an inherited affix must be available inside the rule, and the value obtained by a derived affix inside the rule must be made available to the caller.

If we do not allow the value of an affix to be changed (once it has obtained a value), then the story ends here: all affix-passing mechanisms that conform to the above conditions are indistinguishable (except, perhaps, as to efficiency).

Little is known, however, about the possibility of programming with initializable constants only, and we felt that variables are indispensable. This decision has led to an interesting extension of the 'No cure – no pay' principle to local variables.

Since rules need the possibility to change values of affixes of calling rules, it seems that we need at least call-by-reference (or a more general mechanism). Call-by-reference, however, can surprise the programmer painfully with invisible aliases, as in:

```
action produce a or b+p⟩+q⟩:
        set+p+/a/, set+q+/b/, produce+p.
```

where a call 'produce a or b+x+x' produces /b/. Moreover, back-track rears its ugly head again when a rule fails after having changed the value of an (output) affix.

On the other hand it is clear that call-by-value is insufficient.

A good in-between is found in 'copy-restore': upon rule entry all input affixes are copied to a local work space, and upon rule exit all output affixes are restored from that local work space. If we now suppress the restoring if the rule fails ('copy-*maybe*-restore'), no effects on affixes will propagate upwards upon failure, and a failing rule will never spoil information: the 'No cure – no pay' principle also holds for affixes.

Under these circumstances we can easily introduce 'inout-affixes', which

must have a value upon entrance and which return the (possibly changed) value; notation: +⟩tag⟩.

The copy-maybe-restore mechanism allows us to view the (formal and local) affixes as local variables, some of which are already initialized upon rule entrance and some of will be returned to the caller if and when the rule succeeds. This mechanism is easy to explain and efficient to implement. It aids programming in that it supplies automatic back-tracking on local variables.

The introduction of variables allows the following shorter form of our program:

P4:

```
root text.

external function increment by one+⟩n⟩ = 'INCR',
    function set+⟩n+n1⟩ = 'SET',
    function decrement by one+⟩n⟩ = 'DECR',
    question equal+⟩n+⟩n1 = 'EQUAL',
    predicate absorb+⟩abc = 'ABS',
    action produce+⟩abc = 'PROD'.

action text−n:      $ a 'local'
    read+n+/a/, print+n+/b/, print+n+/c/.

action read+n⟩+⟩abc:
    read symbol+abc, read+n+abc, where rd plus one+n;
    where rd is zero+n.
predicate read symbol+⟩abc: absorb+abc.
function where rd plus one+⟩n⟩: increment by one+n.
function where rd is zero+n⟩: set+0+n.

action print+⟩n+⟩abc:
    where pt is zero+n;
    print symbol+abc, where pt minus one+n, print+n+abc.
action print symbol+⟩abc: produce+abc.
function where pt minus one+⟩n⟩: decrement by one+n.
question where pt is zero+⟩n: equal+n+0.

end
```

5. Other Features

Program P4 is correct ALEPH and, given suitable external routines INCR ··· PROD, it will run. However, a number of externals have been predefined in ALEPH; there are other data types besides the integers used here; there are abbreviations for right-recursive rule calls; and there are other features. All these allow the program to be simplified. For lack of space we shall not treat them here. Details can be found in the ALEPH Manual [6].

6. Conclusion

We have shown that by drawing heavily on the analogy between grammars and programs, and between parsing and problem solving, a practical language can be designed that has some properties not generally found in programming languages.

Among these properties are:

- a simple and effective flow-of-control based solely on selection, decomposition and procedure calling;
- a Central Theorem which states in simple terms the conditions that apply when a given construct is reached;
- an efficient compile-time check on the initialization of variables;
- a firm and compiler-checkable concept of side effects.

References

[1] A.P.W. Böhm, Affixgrammatica's, afstudeerverslag (Affix Grammars, MSc. Thesis), TH Delft (1974) in Dutch.

[2] J.C. Cleaveland and R.C. Uzgalis, Grammars for Programming Languages (Elsevier, Amsterdam, 1977).

[3] D. Crowe, Generating parsers for affix grammars, Comm. ACM 15 (1972) 728–734.

[4] J.P. Dehottay, H. Feuerhahn, C.H.A. Koster and H.M. Stahl, Syntaktische Beschreibung von CDL2, Forschungsbericht Technische Universität Berlin (1976).

[5] R. Glandorf, D. Grune and J. Verhagen, A W-grammar of ALEPH, IW 100/78, Mathematical Centre, Amsterdam (1978).

[6] D. Grune, R. Bosch and L.G.L.T. Meertens, ALEPH Manual, IW 17/75, Mathematical Centre, Amsterdam (1975) (third printing).

[7] C.H.A. Koster, Affix grammars, in: J.E.L. Peck (Ed.), ALGOL 68 Implementation (North-Holland, Amsterdam, 1971) p. 95.
[8] C.H.A. Koster, A compiler compiler, MR 127/71, Mathematical Centre, Amsterdam (1971).
[9] A. van Wijngaarden, Orthogonal design and description of a formal language, MR 76, Mathematical Centre, Amsterdam (1965).
[10] A. van Wijngaarden et al. (Eds.), Revised report on the algorithmic language ALGOL 68, Acta Inform. 5 (1975) 1–236.

Algorithmic Languages, de Bakker/van Vliet (eds.)

On Design Principles for Programming Languages: An Algebraic Approach*

M. Broy, P. Pepper** and M. Wirsing***

Institut für Informatik der Technischen Universität München, Postfach 202420, D-8000 München 2, Federal Republic of Germany

Based on the technique of the algebraic specification of programming languages a number of design principles for programming languages are formally characterized and discussed. The notions covered in this article are abstractness, independence and duality of concepts, expressive power, coherence and formal soundness.

Although these notions cannot be used as a complete methodology for the design of programming languages, they allow for important insights into the semantic structures of programming languages including their comparisons, such that design alternatives can be compared and evaluated.

1. Introduction

The design of a programming language is an intricate task requiring careful reflection and sophisticated decisions. Since questions of taste, personal styles and individual perceptional habits are intermingled with formal considerations and technical requirements, discussions on programming languages are not only challenging and pleasurable but also subjective and quarrelsome topics. In addition, such discussions are all too often based by the use of impressive, yet undefined, slogans and catchwords like 'coherence', 'abstractness', 'high-level', 'very-high-level' and so on.

* This work was partially sponsored by the Sonderforschungsbereich 49, Programmiertechnik, München, Federal Republic of Germany.

** Present address: Department of Computer Science, Stanford University, Stanford, CA 94305, U.S.A.

*** Present address: Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, Great Britain.

With the gradual development of more and more rigorous methods for the definition of programming languages there should also emerge a way of formally characterizing and justifying such catchwords. Accordingly, we will investigate in this paper some consequences that can be drawn from a particular formal view of programming languages. In doing so, our aim is twofold: On the one hand we try to isolate and discuss some basic design principles, and on the other hand we try to give a formal background for obtaining a more precise definition of these principles. Consequently, we will only look at those principles here, for which we can offer some kind of formal characterization.

The approach that we are taking for the specification of programming languages emerged from investigations on a 'wide spectrum language'. This language (cf. [2, 14]) was designed in the course of the project CIP under the joint guidance of Professor F.L. Bauer and the late Professor K. Samelson. Although being a general purpose language, it is devoted to a particular view and methodology of programming: The language is used for the formal specification of software and its stepwise development by applying verified transformation rules. The formalization of this approach led us to consider programming languages as *algebraic theories* (or more technically as *abstract types*). This point of view now allows us to conceive a number of general principles for (the design of) programming languages.

Note: As to the theoretical foundation of our algebraic approach, we will try to burden this article as little as possible with heavy mathematics. Detailed elaborations can be found in the papers listed in the references.

2. Abstractness

In recent years' computer science 'abstractness' has become one of the most popular notions. Indicatively, it is used in various senses and too often without a proper definition. Here we adapt the following idea (leading to the definition given below): *Abstractness* means *to describe phenomena or concepts independent of particular representations.* More mathematically speaking, this means that we are dealing with the whole class of structures in which the respective phenomena occur. This clearly leads to *algebraic theories* (which are nowadays often presented in the form of abstract data types).

Definition. The *abstract syntax* of a programming language is the signature of a type. The basic nonterminals are represented by *sorts,* the individual productions by *functions* over these sorts (cf. [20]).

Example 1. Let the sorts **id** and **expr** representing identifiers and expressions, respectively, be given. Then we may have the following correspondence between a concrete BNF-syntax and a signature:

⟨stat⟩ ::= ⟨id⟩ := ⟨expr⟩	assign: **id** × **expr** → **stat**
⟨stat⟩ ::= **skip**	skip: → **stat**
⟨stat⟩ ::= ⟨stat⟩; ⟨stat⟩	semi: **stat** × **stat** → **stat**
⟨stat⟩ ::= **if** ⟨expr⟩ **then** ⟨stat⟩ **fi**	cond: **expr** × **stat** → **stat**
⟨stat⟩ ::= **while** ⟨expr⟩ **do** ⟨stat⟩ **od**	while: **expr** × **stat** → **stat**
⟨stat⟩ ::= **do** ⟨stat⟩ **until** ⟨expr⟩ **od**	until: **expr** × **stat** → **stat**

The *term algebra* $W(\Sigma)$ (also called the word algebra) of such a signature Σ provides the set of *abstract programs.* Each concrete syntax can be considered as an initial model of the type $T' = \langle \Sigma, \emptyset \rangle$ having the signature Σ and an empty set of axioms. Hence, each concrete syntax (in particular the most common one of parsing trees) is isomorphic to the abstract syntax. (For a deeper analysis see [24].) The next step is now straightforward:

Definition. The *abstract semantics* of a language is given by a set $\mathcal{A}$ of axioms, which are added to the abstract syntax Σ.

In this way, the complete syntactic and semantic specification of a language is given by the pair $T = \langle \Sigma, A \rangle$, the meanwhile classical presentation of an abstract data type.

In this setting, the *context conditions* (sometimes also called static semantics) may be given by a set of *definedness predicates;* the requirement that context conditions must be checkable at compile time therefore means that the definedness predicates must be specified 'sufficiently complete' and must be decidable.

The (dynamic) semantics causes more intrinsic problems. For termination (i.e. least fixed point properties) cannot be expressed 'sufficiently completely' by first order conditional equations. There are two solutions to

this issue: Either one designates a sufficiently powerful kernel of the language with its, say mathematical or operational semantics, and reduces all other language constructs to this kernel ('transformational semantics' [28]), or one extends the theory of abstract types by considering special homomorphisms (cf. [6, 7, 8]).

Example 2. Given the signaure of Example 1, the essential axioms for algebraically specifying the language are:

$$\text{semi}(\text{skip}, s) = s = \text{semi}(s, \text{skip}),$$

$$\text{cond}(\text{true}, s) = s, \qquad \text{cond}(\text{false}, s) = \text{skip},$$

$$\text{while}(e, s) = \text{cond}(e, \text{semi}(s, \text{while}(e, s))),$$

$$\text{until}(e, s) = \text{semi}(s, \text{cond}(\neg e, \text{until}(e, s))).$$

The rest of the specification centers around the assignment. For example, in the 'transformational semantics' version one uses axioms like

$$\text{semi}(\text{assign}(x, e), \text{assign}(x, f)) = \text{assign}(x, f_x^e)$$

in order to transform every statement into the 'normal form' of a single collective assignment (cf. also Example 5). Then a rule such as

$$\lceil x := E; x \rfloor = E$$

associates an input/output relation (based on the semantics of expressions) to each statement.

In the other approach, one introduces a semantic function like

$$\text{value}: \mathbf{stat} \times \mathbf{expr} \to \mathbf{data}$$

that gives the value of an expression after executing a statement. This requires axioms such as

$$\text{value}(\text{semi}(s, \text{assign}(x, e)), f) = \text{value}(s, f_x^e).$$

The extended theory of abstract types is needed in order to cope with the possibly arising infinite reduction sequences.

In contrast to e.g. denotational semantics, where one particular model is fixed, our algebraic specifications characterize in general whole classes of semantic models. In these classes there are models corresponding to mathe-

matical semantics as well as models corresponding to various possibilities of operational semantics. With the help of order relations induced by the different homorphisms one can then compare these different semantic structures. To elucidate this further, we will cite here one result (that stems from [6]):

The minimal model in the aforementioned ordering is called 'weakly terminal model'. In [26] the notion of 'fully abstract semantics' is introduced, which means informally that two programs are equivalent (i.e. equally interpreted in the semantic model) if and only if they can be substituted for each other in any 'primitive' context without effecting any changes (i.e. considered as black boxes they are indistinguishable). The weakly terminal model – if it exists – provides such a fully abstract semantics, and we call two programs *extensionally equivalent*[1] if they are equal in this model. (For instance, in Example 2 the programs until($\neg e, s$) and semi(s, while(e, s)) are extensionally equivalent.)

To conclude this brief exposition of the algebraic specification of programming languages we would like to mention a general property (shown in [12]): Every formal definition of a language induces some extensional equivalence, and vice versa every equivalence relation (when considered as being extensional) induces a formal semantics.

The abstraction achieved by the algebraic approach encourages one to design languages in terms of concepts and their properties rather than by giving meanings to notations.

The following sections are now devoted to conclusions that can be drawn from this algebraic view.

3. Expressive power

Expressive power is not an absolute measure but rather a means for comparing languages. Hence, we will only define the expressive power of one language relative to another language. Though concerning mainly the number and strength of the concepts of a language this notion is sometimes also used in connection with a mere *syntactical richness* ('notational variants').

As to the latter point, our notion of abstractness allows us to get rid of

[1] Sometimes also called 'observably equivalent' or 'visibly equivalent'.

such notational variants. For instance, the two iterative constructs of the previous section are related to each other by means of the equation

$$\text{until}(e, s) = \text{semi}(s, \textit{while}(\neg e, s))$$

leaving only the concept of 'iteration'. Analogously, the concept of a 'recursive function' is independent from its presentation as an ALGOL-like declaration

$$\textbf{funct}\, f \equiv (\textbf{m}\, x)\textbf{r} : E\langle x, f\rangle$$

or as a fixed point expression in the style of the types λ-calculus

$$\mathbf{Y}\, f : [\textbf{m} \rightarrow \textbf{r}] \,.\, \lambda x : \textbf{m} \,.\, E\langle x, f\rangle .$$

Example 3. This syntactic richness becomes most apparent in languages that provide a huge collection of special operators. Consider the operator '/' of APL (cf. [22]); its effect is given by the functional

funct slash ≡ (**vector** x, **funct(real, real) real** op) **real**:

if length(x) = 1 **then** first(**x**)

▯ length(x) > 1 **then** op(first(x), slash(rest(x, op)) **fi**.

Similar operators, which are oriented towards special data structures, can be found in SETL (cf. [16]).

These considerations may be formalized as follows (cf. also the notation of 'extensions by definitions' in [30]).

Definition. Let L be a language (i.e. an abstract type). A new construct g (i.e. a new function) can be added to L as a *notational variant* by specifying an axiom of the form

$$g(x_1, \ldots, x_n) = E$$

where E is a term of L in the free variables $x_1, \ldots, x_n$ (and of course the operation g must not occur in E).

The classical way of comparing the expressive power of two languages is to map both onto the same semantic model. This is rather straightforward for two applicative languages, say LISP and the Backus-language, or for

two procedural languages, say ALGOL 60 and PASCAL; but it becomes quite artificial, when an applicative language is to be compared with a procedural one.

In order to get a more direct mode of comparison, we employ again our algebraic techniques. As a prerequisite we need a common basis, viz. a common set of primitive data types. (We will consider this basis as an unspecified parameter, both for gaining flexibility and for avoiding 'simple' translations via Gödelization.)

Definition. A language L_1 *is interpretable* in a language L_2, if there is a mapping (morphism) $\phi: W(L_1) \to W(L_2)$, which associates to each term of L_1 a term of L_2 (and which is the identity on the 'primitive terms' of the common basis) such that congruent terms of L_1 are mapped to congruent terms of L_2, i.e.

$$t \sim_1 t' \Rightarrow \phi(t) \sim_1 \phi(t')$$

In other words, the equivalence classes constituting the semantics of L_1 are mapped to equivalence classes of L_2. Two languages are called *equipollent,* if each of them is interpretable in the other one. This renders the notion of a 'coherent family of languages' [19] more precise.

The above definition still contains a certain degree of freedom, viz. the congruence relations to be chosen. For instance, if in both cases the extensional equivalence (cf. Section 2) is taken, then the expressive power refers to the mathematical semantics, i.e. to the input/output behaviour. Other equivalence relations allow us to compare versions of operational semantics.

Example 4. Consider a language L_m with collective assignments and a language L_s with only single assignments. The correspondence

$$\phi: (x, y) := (e, f) \mapsto \lceil \textbf{var } h := e;\ y := f;\ x := h \rfloor$$

establishes the interpretation of L_m in L_s (the converse is trivial). Hence, both languages are equipollent, if we consider the extensional equivalence. However, if we consider an 'operational' equivalence, where the number of used variables plays a role, then the equivalence classes of L_m are not necessarily mapped to equivalence classes of L_s.

Similarly, without blockstructure the above translation does not work. For there exists always an environment in which the auxiliary identifier h occurs. In this case L_s is definitely 'weaker' than L_m.

4. On the duality of styles

In programming, we distinguish two major 'styles', viz. applicative languages and procedural languages. The former comprise expressions and possibly also declarations of constants and functions, the latter are made up of assignments, loops, procedures and even goto's.

As an example, we will now give an interpretation of the procedural language of Example 1 in the following applicative language (for reasons of easier readability we use concrete syntax and let x stand for a whole tuple of identifiers):

function abstraction:	$\lambda x . E$,
function application:	$(\lambda x . E_1)(E_2)$,
conditional expression:	$(B \to E)$,
fixed point:	$Yf . \lambda x . E$

where $(\lambda x . \text{false} \to E) = \text{id}$.

To each statement S of a procedural language we can now define its *associated expression* E_s by induction on the structure of the language.

Example 5. The language of Example 1 can be associated to the above expression language as follows:

skip	$\to \text{id}(x)$
$x := E$	$\to E$
$S; T$	$\to (\lambda x . E_T)(E_S)$
if B **then** S **fi**	$\to (B \to E_S)$
while B **do** S **od**	$\to (Yf . \lambda x . B \to f(E_S))(x)$.

In this way, every statement containing the variable x corresponds to an expression where x (in general) occurs freely.

Now it remains to show that the translation is compatible with the equivalences characterizing the semantics of the languages. We will consider here only two examples: First, the associativity of the semicolon requires the equivalence of the expressions

$$(\lambda x\,.\,E_3)((\lambda x\,.\,E_2)(E_1) \quad \text{and} \quad (\lambda x\,.\,((\lambda x\,.\,E_3)(E_2)))(E_1).$$

This equivalence is indeed valid for usual expression languages (both under call-by-value and call-by-name semantics). As the second example, we consider the recursive characterization of the while-loop, viz. the equivalence.

$$\textbf{while } B \textbf{ do } S \textbf{ od} = \textbf{if } B \textbf{ then } S\textbf{; while } B \textbf{ do } S \textbf{ od fi;}$$

this is compatible with the applicative language because of the characteristic fixed point property

$$Yf\,.\,\lambda x\,.\,E = \lambda x\,.\,E_f^{Yf\,.\,\lambda x\,.\,E}.$$

In the above translation the image of every construct of the statement language is a notational variant of an expression.

Definition. Let the language L_1 be interpretable in L_2 wrt. the mapping ϕ. A construct g of L_1 is *representable in the style of* L_2 if $\phi(g)$ can be specified by

$$\phi(g(x_1, \ldots, x_n)) = E$$

where E is a term of L_2 in the free 'variables' $\phi(x_1), \ldots, \phi(x_n)$. If all constructs of L_1 are representable in L_2 and vice versa, then L_1 and L_2 are called *'dual in style'*.

E.g. the while-statement is represented in the expression language by 'tail recursion'. The procedural language is representable in the style of the applicative language, but the converse is not true (cf. [31]).

A number of interesting results on the equipollence of certain recursion and iteration mechanisms are well known from the literature:

(i) Counted iteration (**for** *i* **to** *n* **do** ... **od**) and primitive recursion are equipollent and even mutually representable in the other style.

(ii) General goto-systems as well as nested loops with multiple-level exits are equipollent to systems of (mutually) tail-recursive functions (cf. [14, 15]), but are not dual in style.

(iii) Parallel programs with processes described by simple tail-recursion

(for instance by while-loops) are equipollent to mutually recursive, tail-recursive sequential nondeterministic programs (see [4]), but are not dual in style (note, that this holds only for closed parallel programs but not for single processes).

The translation of Example 5 may be used to clarify relationships between a number of other semantic properties. We will exemplify this by two observations on call-by-value and call-by-name semantics, showing the close relationships of different topics.

Example 6. Consider the trivial procedural program

(P) **while** true **do** $x := x$ **od**; $x := 1$

and its associated expression

(E) $(\lambda x\,.\,1)((Yf\,.\,\lambda x\,.\,\text{true} \rightarrow f(x))(x))$.

In *call-by-value* semantics the applicative program (E) does not terminate and so does the procedural program (P) in classical languages. In a *call-by-name* semantics, however, the program (E) does terminate and yields 1; the same is intended for the program (P) in procedural data flow semantics.

Example 7. Consider the program part

var y; **var** $x := a$, **while** $b(x)$ **do** $y := g(x)$; $x := h(x, y)$ **od**

although the variable y is not initialized, this program may have a perfect meaning. We can capture this meaning and also the possible failure by the expression.

$(\mathbf{Y}f\,.\,\lambda x, y\,.$ **if** $b(x)$ **then** $\langle h(x, g(x)), g(x)\rangle$ **else** $\langle x, y\rangle$ **fi**$)(a,$ **error**$)$,

provided that we assume a call-by-name semantics for functions such that the argument **error** does not harm. There is no direct way of representing the above procedural program under a call-by-value semantics, which is, however, necessary for explaining e.g. the semicolon by the composition of function applications (the only way out is to introduce some artificial value 'still not initialized').

Besides the duality of applicative and procedural styles, there is a second major duality between two different conceptual issues, viz. the correspondence of concurrent program and nondeterministic sequential pro-

grams. (This correspondence was put into formal transformation rules in [3, 4].)

Example 8. The semantics of conditional critical regions may be explained by axioms like

$$/\!/ \textbf{ await } B_1 \textbf{ then } S_1 \textbf{ end; } T_1 \; /\!/ \textbf{await } B_2 \textbf{ then } S_2 \textbf{ end; } T_2 \; /\!/$$
$$= \textbf{if } B_1 \textbf{ then } S_1; \; /\!/ \; T_1 \; /\!/ \textbf{await } B_2 \textbf{ then } S_2 \textbf{ end; } T_2 \; /\!/$$
$$[] \; B_2 \textbf{ then } S_2; \; /\!/ \textbf{ await } B_1 \textbf{ then } S_1 \textbf{ end; } T_1 \; /\!/ T_2 \; /\!/ \textbf{ fi.}$$

In this way, every program text containing parallel constructs is associated to a program text without such constructs (based on Dijkstra's guarded commands). Hence, the equivalence classes of these sequential programs include equivalence classes on the parallel programs. The semantic interpretation of nondeterministic sequential programs (in the style of Example 5) induces then a (functional) semantics for concurrent programs.

Again, there are interesting conclusions that can be drawn from this duality (cf. [5]):

Example 9. Consider the concurrent program (cf. [17, 27])

(C) $x, y, z := 0, 0, 0;$ /// **await** true **then** $x := 1$ **end**

$$/\!/ \textbf{while } z = 0 \textbf{ do } y := y + 1;$$
$$\textbf{await } \text{true } \textbf{then } z := x \textbf{ end od } /\!/.$$

According to the rules of Example 8 this program corresponds to the sequential nondeterministic one

$x, y, z := 0, 0, 0;$ **call** q **where**

proc $q \equiv$ **if** $z = 0$ **then** $y := y + 1;$ $[x := 1; \; z := x;$

(S) **call** $p [] z := x;$ **call** $q]$

else $x := 1$ **fi**

proc $p \equiv$ **while** $z = 0$ **do** $y := y + 1; \; z := x$ **od**

where [... [] ...] is an abbreviation for **if** true **then** ... [] true **then** ... **fi**. If we

are only interested in the final value of y, then we get the associated expression of (S) by the rules of Example 5 (leaving away the superfluous call of p):

(E) $\quad Yf \,.\, \lambda y \,.\, [y+1 \,\mathbb{I}\, f(y+1)](0).$

Under the assumption of *fairness* (C) always terminates and returns some natural number $y>0$. In contrast to this, (E) either terminates and also yields some $y>0$ or does not terminate at all. However, an expression with an infinite number of possible results but without the possibility of nontermination is not continuous in the Egli-Milner ordering (which is needed to define least fixed points of nondeterminate functions). Hence, general fairness assumptions are not compatible with our explanation of parallel programs. (We would get 'computable' functions which are not continuous). Or, in other words, without any fairness assumptions concurrent programs with shared variables and sequential nondeterministic programs are equipollent. Note, however, that fairness assumption even may be introduced for sequential nondeterministic programs (cf. [21]).

5. Formal soundness: order structures and monotonicity

Numerous phenomena of programming and programming languages are based on order relations. The most widely known of these orderings is probably the 'less defined' relation used in the fixed point theory underlying denotational semantics ($x \subseteq y$ basically means that x is undefined or equal to y). To cope with such issues we have to supplement our algebraic structures with order structures.

Of course, the order structure has to be compatible with the algebraic one. As a prominent example for justifying this requirement consider the relation "A is more efficient than B". This relation can only be useful, if there is no context P such that $P[A]$ is less efficient than $P[B]$.

Definition. Let L be a language (i.e. an algebraic type) and let $\leq$ be an ordering. Then L is said to be *formally sound* w.r.t. $\leq$, if in each of its semantic models the following monotonicity is valid for every function f:

$$\forall x, y\colon x \leq y \Rightarrow f(x) \leq f(y).$$

We may now apply these criteria to some of the more intrinsic problems

of nondeterminism and parallelism. Since here the various design decisions have rather subtle effects, a formal means for analyzing their mutual influences is utterly necessary to provide for the desired rigidity. For this reason, we will stick to the semantic definition of Example 8 throughout the rest of this chapter.

The natural ordering for nondeterminate programs has already been introduced in [25]:
P' is a *descendant* of P, denoted by $P' \subseteq P$, if the set of possible outcomes of P' is contained in that of P. (For convenience let us denote by $\perp \subseteq P$ that P also leads to nonterminating computations.)

Example 10. Let us resume the applicative program of example 9:

(E) $\quad Yf \,.\, \lambda y \,.\, [y+1 \,[\!]\, f(y+1)](0).$

The descendants of (E) are given by the nonterminating program $\perp$ and by the programs

(E_n) $\quad (\lambda y \,.\, n)(0)$ for $n \in N \setminus \{0\}$.

Of course, our semantic definition immediately transfers the notion of *descendant* also to parallel programs. Thus, the sequential programs

(S_n) $\quad x, y, z := 1, n, 1 \quad$ for $n \in N \setminus \{0\}$

and $\perp$ are all descendants of the concurrent program (C) of Example 9.

Unfortunately, there are at least three different views of nondeterminism that can be found in the literature (cf. [9]). The one we have adapted so far may be called *totally erratic,* since it may arbitrarily choose any of the possible execution paths be it terminating or nonterminating. With the help of our notion $P' \subseteq P$ we can now explain the two other ones:

The *angelic nondeterminism* (as termed by Hoare) is used in [25] and also in automata theory. Here possible termination is equivalent to guaranteed termination. In this case, only the programs (S_n) are descendants of (C), which is exactly what fairness conditions shall achieve. In other words, fairness conditions correspond to angelic nondeterminism.

The *demonic nondeterminism* (again a term of Hoare) is underlying Dijkstra's wp-calculus. Here possible nontermination is equivalent to guaranteed nontermination. In this view, only the program $\perp$ is a descendant of (C).

Example 11. Consider a language construct for unbounded nondeterminism (cf. [11]):

some x: $p(x)$.

Assume we intend a demonic semantics for **some,** but the erratic semantics for '$[\!]$', i.e.

(**some** x: $p(x)$) $=_e \perp$ if $\perp \subseteq p(a)$ for some a.

This means that the program

some x: (x **equal** (Yf . $\lambda\tilde{x}$. [1 $[\!]$ $f(\tilde{x})$])(0))

is equivalent to $\perp$. However, if we pass over to a descendant of f, the resulting program

some x: (x **equal** (Yf . $\lambda\tilde{x}$. 1)(0))

is equivalent to 1. This clearly violates the requirement of formal soundness.

6. Structured language design

In the field of abstract data types much emphasis has been given to questions of a proper modularization. (In fact, this has been one of their major motivations.) This led to very precise notions of e.g. *algebraic enrichment* and *hierarchical types* (cf. [33]). Our approach to the algebraic specification of programming languages allows us to apply all these results to language design.

The enrichment technique, for instance, leads to a structuring of the language into a small sublanguage representing the conceptual skeleton and a number of enrichments introducing notational variants.

The hierarchical structure introduces several layers of the language for each of which the lower ones act as primitive basis (in the same way as the type INTEGER forms the basis for the type STACK of INTEGER).

Example 12. A classical procedural language can be represented by the folowing hierarchy of abstract types

type STATEMENT ≡ **sort statement,**	procedural language, defines statements based on the sorts **identifier, expression, boolean, integer, stack,** ...
type EXPRESSION ≡ **sort expression,**	expression language, defines expressions based on the sort **identifier, boolean, integer, stack,** ...
type STACK OF INTEGER **type** INTEGER **type** BOOLEAN	hierarchy of data types

Obviously this way a clear structure is induced on the language. For instance, we may understand the basic data types (BOOLEAN, INTEGER, STACK, ...) without knowing anything about expressions or statements. Similarly we may understand the type EXPRESSION without considering the type STATEMENT. If the language, however, incorporates expressions with side-effects, then the sort **expression** can no longer be explained without considering statements. The language 'looses structure'. (Which again provides a formal justification for an often cited argument.) Note, that in the case of including parallel programs into the statements, we have to include nondeterminism both for statements and for expressions to maintain the hierarchy.

7. Conclusion

A rigorous proceeding along the line described in this paper is applied in the design of the wide spectrum language CIP-L (cf. [14]), demonstrating that the algebraic treatment makes also large-sized languages manageable. But our experience with the language shows that the major effect of the algebraic approach does not lie in the resulting formal description but rather in guidelines provided for the design. Although there is still enough room for decisions that give a language its characteristic appearance, there exist at least criteria to classify these decisions ('notational variant', 'new concept' etc.). Above all, the compatibility of the various parts of the language can be checked, which is particularly important in a wide spectrum language where different styles are combined within a single syntactic frame.

Acknowledgement

The authors gratefully acknowledge a number of fruitful discussions with colleagues of the Project CIP, especially with Prof. F.L. Bauer and the late Prof. K. Samelson.

References

[1] F.L. Bauer and H. Wössner, Algorithmische Sprache und Programmentwicklung (Springer, Berlin, 1981) to appear.

[2] F.L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner, H. Partsch, P. Pepper and H. Wössner, Towards a wide spectrum language to support program specification and program development, SIGPLAN Notices 13 (12) (December 1978) 15–24.

[3] M. Broy, Transformation parallel ablaufender Programme, Dissertation, Technische Universität München, Fakultät für Mathematik (1980).

[4] M. Broy, Transformational semantics for concurrent programs. Inform. Process. Lett. 11 (1980) 87–91.

[5] M. Broy, Are fairness assumptions fair? 2nd Int. Conf. on Distributed Programming Systems, Paris (April 1981).

[6] M. Broy and M. Wirsing, Algebraic definition of a functional programming language, TU München, Techn. Rep. TUM-I8008 (1980).

[7] M. Broy and M. Wirsing, Programming languages as abstract data types, 5ème Coll. les Arbres en Algebre et Programmation, Lille 1980, Univ. de Lille (1980) pp. 160–177.

[8] M. Broy and M. Wirsing, Partial recursive functions and abstract data types, EATCS Bull. 11 (1980) 34–41.

[9] M. Broy and M. Wirsing, On the algebraic specification of nondeterministic programming languages, 6ème Coll. les Arbes en Algebre et Programmation, Genova (1981).

[10] M. Broy, P. Pepper and M. Wirsing, On relations between programs, in: B. Robinet (Ed.), 4th Int. Symp. on Programming, Lecture Notes in Computer Science, Vol. 83 (Springer, Berlin, 1980) pp. 59–78.

[11] M. Broy, R. Gnatz and M. Wirsing, Semantics of nondeterministic and noncontinuous constructs, in: F.L. Bauer and M. Broy (Eds.), Program Construction, Lecture Notes in Computer Science, Vol. 69 (Springer, Berlin, 1979) pp. 553–592.

[12] M. Broy, H. Partsch, P. Pepper and M. Wirsing, Semantic relations in programing languages, IFIP-Congress (1980).

[13] M. Broy, B. Möller, P. Pepper and M. Wirsing, A model-independent approach to implementations of abstract data types, in: A. Salwicki (Ed.), Proceedings of the Symposium on Algorithmic Logic and the Programming Language LOGAN, Lecture Notes in Computer Science (Springer, Berlin, 1981) to appear.

[14] CIP, Report on a wide spectrum language for program specification and development, Techn. Universität München, Institut für Informatik, TUM-I8104 (May 1981).

[15] G. Cousineau, An algebraic definition for control structures, Theoret. Comput. Sci. 12 (1980) 175–192.

[16] R.B.K. Dewar, A. Grand, S. Lin, J.T. Schwartz and E. Schonberg, Programming by refinement, as exemplefied by the SETL representation sublanguage, ACM TOPLAS 1 (1) (1979) 27–49.

[17] E.W. Dijkstra, A discipline of programming (Prentice Hall, Englewood Cliffs, NJ, 1976).

[18] E.W. Dijkstra, On weak and strong termination, EWD 673, The equivalence of bounded nondeterminacy and continuity, EWD 675 (1978).

[19] A.P. Ershov, Problems in many-language systems, in: F.L. Bauer and K. Samelson (Eds.), Language Hierarchies and Interfaces, Lecture Notes in Computer Science, Vol. 46 (Springer, Berlin, 1976) pp. 358–427.

[20] J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, Initial algebra semantics and continuous algebras, J. Assoc. Comput. Mach. 24 (1) (1977) 68–95.

[21] O. Grümberg, N. Francez, J.A. Makowsky and W.P. de Roever, A proof rule for fair termination of guarded commands, Rijksuniversiteit Utrecht, Vakgroep Informatica, RUU-CS-81-2 (January 1981).

[22] K.E. Iverson, Notation as a tool of thought, Comm. ACM 23 (8) (1980) 444–465.

[23] P. Landin, A correspondence between ALGOL 60 and Church's lambda notation: Part I, Comm. ACM 8 (2) (1965) 89–101.

[24] A. Laut, Darstellung kontexfreier Grammatiken als Rechenstrukturen und ihre Verwendung für die Transformation von Programmen, Technische Universität München, Institut für Informatik, to appear.

[25] J. McCarthy, A basis for a mathematical theory of computation, in: B. Braffort and D. Hirschberg (Eds.), Computer Programming and Formal Systems (North-Holland, Amsterdam, 1963).

[26] R. Milner, Fully abstract models of typed lambda-calculi, Theoret. Comput. Sci. 4 (1977) 1–22.

[27] D. Park, On the semantics of fair parallelism, in: D. Björner (Ed.), Abstract Software Specifications, Lecture Notes in Computer Science, Vol. 86 (Springer, Berlin, 1980) 504–526.

[28] P. Pepper, A study on transformational semantics, in: F.L. Bauer and M. Broy (Eds.), Program Construction (Proc. Int. Summer School, Marktoberdorf 1978), Lecture Notes in Computer Science, Vol. 69 (Springer, Berlin, 1979) pp. 322–405. (Also: Dissertation, Technische Universität Munchen (1979)).

[29] D. Scott and C. Strachey, Towards a mathematical semantics for computer languages, in: J. Fox (Ed.), Computers and Automata (John Wiley, New York, 1972).

[30] J. Schoenfield, Mathematical Logic (Addison-Wesley, Reading, MA, 1967).

[31] H.R. Strong, Translating recursion equations into flow charts, in: Proc. 2nd Annual ACM Symposium on Theory of Computing, New York (1970). Also in J. Comput. System Sci. 5 (1971) 254–285.

[32] E.G. Wagner, J.W. Thatcher and J.B. Wright, Programming languages as mathematical objects, in: 7th MFCS, Lecture Notes in Computer Science, Vol. 64 (Springer, Berlin, 1978).

[33] M. Wirsing, P. Pepper, H. Partsch and W. Dosch, On hierarchies of abstract data types, TU München, Techn. Rep. TUM-I8007 (1980).

Algorithmic Languages, de Bakker/van Vliet (eds.)

Invited Address

The Structured Description of Algorithm Derivations

John Darlington

Department of Computing, Imperial College, London, Great Britain

0. Introduction

The Computer Scientist can gain many benefits from a close study of algorithms and their development. Not least amongst these are the intellectual satisfaction gained and the insights such studies give into the process of programming. However, this activity can be surprisingly difficult and frustrating. Often one's attention is drawn to some clever algorithm published in the literature and only after an investment of time and energy that seems out of all proportion to the actual amount of text being studied does one begin to understand how the algorithm works, or even be convinced that it does indeed achieve its intended purpose. This process can often be circumvented by seeking someone who knows the algorithm in question. Such a person is usually able to convey the basic ideas underlying the algorithm very quickly and armed with these insights one is able to return to the code and quickly complete one's understanding. Often it turns out that the code is in the opaque form it is because of some relatively unimportant final optimisation.

As it is with the academic discipline of algorithm study so it is with the humble activity of programming. We have long lamented our inability to produce correct understandable and efficient programs or to maintain or modify systematically programs produced elsewhere. It is often commented that programming is closer to an art than a science and many of the methods or tools felt necessary in established engineering disciplines are completely lacking in programming.

The work presented here is aimed at removing some of these difficulties and deficiencies. We hope to provide a means of describing algorithms and their development that does not obscure the fundamentals of the algorithm in question but maintains the level of preciseness and formality that we

think is necessary. The crucial point we feel is that if one is just presented with the code form of an algorithm what one is seeing is the end point of some quite involved intellectual activity any record of which has been thrown away. We wish to encourage programmers to 'show their working' for the benefit of those who follow. We have all made or been the subject of exhortations to document programs and their 'design'. Quite what is the status of this object a 'design' has never to our minds been satisfactorily explained, we hope to provide a notation in which it becomes a formal object capable of being communicated and studied precisely.

In this endeavour we share many aims with those who have studied the programming process and conducted many elegant studies of algorithm developments such as that reported in [11]. However, there are some crucial differences between the aims of these studies and our own. Firstly there is the question of discovery versus communication. The intellectual processes that a programmer goes through when initially developing a new program or discovering a clever algorithm form a fascinating area for study but we are primarily interested in the question of how such successful developments can be communicated or checked. Of course any methodology that aids communication will probably aid discovery but we think it is important to separate these two concerns.

Secondly we are interested in using computers themselves to assist programming. It would be ironic if programming itself was one of the few 'white collar' activities to totally ignore the advances in productivity achievable by a sensible utilisation of the computing power becoming increasingly available. At present many of the phases of program development prior to coding are difficult to mechanise. We think computers could assist in making this much more rigorous and systematic with consequent improvements in reliability, understandability and modifiability. Thus we are seeking notations formal enough to communicate to a machine. Formality though need not imply unintelligability. At the present state of technology a complete automation of the programming process is unachievable. A sensible division of labour can be achieved by relying on the user to supply the main intellectual insights and leaving to the machine the detailed checking and book keeping that is necessary for accuracy.

The solution that we present to these interrelated problems is based on the idea of program transformation. Using the transformational approach to programming a programmer does not attempt to produce directly a program that is correct, understandable and efficient, rather he initially

concentrates on producing a program that is as clear and understandable as possible ignoring any question of efficiency. Having satisfied himself that he has a correct program he successively transforms it to more and more efficient versions using methods guaranteed not to change the meaning of the program. Our work on transformation started in collaboration with R.M. Burstall at Edinburgh University. The need to perform significant manipulations on programs led us to concentrate on programs written in a functional language first, NPL [3] and then HOPE [6]. Support is growing for the thesis that these languages are more suitable tools for program development than the so-called imperative languages, such as PASCAL or ALGOL, see [1] and we will confine our discussions to program developments expressed within these languages. With Burstall we developed a simple yet powerful methodology for transforming functional programs that has become known as the unfold/fold system [4]. Many different types of transformation can be expressed within this formalism which guarantees that (partial) correctness is maintained. However, significant transformations became too complex when expressed solely within this formalism. To overcome this we are developing a meta-language that can be used to explicate transformations in a structured way. This we feel offers a way to achieve the goals outlined above of providing a calculus of algorithm development that is both intelligable and formal enough to be checked by machine.

In Section 1 we introduce our functional language HOPE and the unfold/fold transformation methodology. In Section 2 we, very briefly, describe the types of transformations that can be achieved using the unfold/fold system. Section 3 describes our meta-language and discusses some of the higher level transformations or transformation 'tactics' that can be written. Section 4 includes the full development of a well-known algorithm described using the meta-language, and Section 5 concludes with a discussion of the style of programming environments that could be provided in the future.

1. Languages and Transformation Methodology

1.1. *HOPE*

In this section we will outline enough of HOPE to enable the reader to

understand the examples used. A fuller exposition is contained in [6]. HOPE is a higher order strongly typed functional language using recursion equations, first implemented at Edinburgh. We are at present developing a HOPE compiler at Imperial. The syntax we will use here differs slightly from that of the current implementation.

A HOPE program is a set of equations defining functions. Separate equations can be written for separate cases of the input variable. For example

$$\text{fib}(0) = 1$$
$$\text{fib}(1) = 1$$
$$\text{fib}(n+2) = \text{fib}(n+1) + \text{fib}(n)$$

defines the Fibonacci numbers. The cases on the left hand side of a function definition must be non-overlapping and exhaust all the possibilities for the types of the arguments involved.

HOPE is strongly typed, thus before being defined a function must have its type declared, this is done using the **dec** statement

dec fib : num → num

HOPE employs polymorphic type checking [24] so that type declarations can involve type variables e.g.

typvar alpha
dec f : alpha → num
$f(a) = 0$

is the stubborn function that returns 0 whatever you give it. Data structures in HOPE are represented as terms built up from constructor functions i.e. functions having no equations. These are introduced using the **data** statement. Thus

data listnum == nil ++ cons(num,listnum)

defines the data type list of numbers built up using the constructors nil (the empty list) and cons. Data statements can also be paramaterised thus

data list(alpha) == nil ++ cons(alpha,list(alpha))

now defines a *type constructor* list such that list(num) is equivalent to the type listnum defined earlier.

Thus

dec length : list(alpha)→num
length(nil) = 0
length(cons(a, 1)) = 1 + length(1)

calculates the length of any list whatever its constituents.

Running a HOPE program involves reducing an expression until it is totally composed of constructor functions i.e. no more equations apply. Thus cons(1, cons(2, cons(3, nil))) is the list of length 3 and length(cons(1, cons(2, cons(3, nil)))) reduces to 3.

Infix operators are widely used in HOPE, thus we can define :: as an infix operator for cons and the above equations become

length(nil) = 0
length(a :: 1) = 1 + length(1)

HOPE also allows user defined distfix (distributed-fix) operators, similar to the traditional **if then else**. These are introduced using the **distfix** statement, e.g.

distfix while __ do __

Underscores mark the places where operands should go. The name of such an operator, for use in **dec** statements is the leftmost word in its declaration.

There are two equivalent forms for the conditional expression. Thus either

fact(n) = 1 **if** $n = 0$
else n * fact($n - 1$)

or

fact(n) = $n = 0$ **then** 1
else n * fact($n - 1$)

Local variables may be introduced using either the **let** or **where** construct. Thus

$f(x)$ = **let** $u == x^2$ **in** $u + u$

and

$f(x)$ = $u + u$ **where** $u == x^2$

are both equivalent to $f(x) = x^2 + x^2$.

Being higher order HOPE allows functions to be passed as parameters and

returned as values. Thus

```
typevar alpha, beta
dec *:(alpha→beta) # list(alpha)→list(beta)
infix *:6
f*nil = nil
f*(a::1) = f(a)::(f*1)
```

defines an operator * that applies a function to every element of a list, thus

fact *(1::2::3::nil) evaluates to 1::2::6::nil

Higher order functions are especially useful as they provide iterators which 'package' recursion as in * above and avoid having to write it explicitly many times. Of particular use are iterators over sets and HOPE has borrowed the traditional set comprehension schema, thus

```
primesquares: set(num)→set(num).
primesquares(S) = {n² | n ∈ S & isprime(n)}
```

is the set of squares of all primes contained in a given set.

1.2. *A transformation methodology*

Assume we have the following functions defined

```
dec length : list(alpha)→num
length(nil) = 0                                          (1)
length(cons(n, 1)) = 1 + length(1)                       (2)

dec append: list(alpha) # list(alpha)→list(alpha)
append(nil, 12) = 12                                     (3)
append(cons(n, 11), 12) = cons(n, append(11, 12))        (4)
```

(append joins two lists together)

and say we wanted to write a program to join two lists together and calculate the length of the resulting list. Naively we could write this as

```
dec lengthof2: list(alpha) # list(alpha)→num
lengthof2(11,12) = length(append(11, 12))                (5)
```

This is a perfectly adequate program but it contains some avoidable inefficiency. Let us see if we can improve it. All our manipulations will

take equations and produce further equations that do not change the meaning of the program.

Firstly we can **instantiate** (5) by letting 11 be nil getting

$$\text{lengthof2(nil, 12)} = \text{length(append(nil, 12))}$$

(3) allows us to rewrite this as

$$\text{lengthof2(nil, 12)} = \text{length(12)} \tag{6}$$

Returning to (5) we now instantiate 11 to cons(n, 11) getting

$$\text{lengthof2(cons}(n, 11), 12) = \text{length(append(cons}(n, 11, 12))$$

(4) allows us to rewrite this as

$$\text{lengthof2(cons}(n, 11), 12) = \text{length(cons}(n, \text{append}(11, 12)))$$

(2) allows us to rewrite this as

$$\text{lengthof2(cons}(n, 11), 12) = 1 + \text{length(append}(11, 12))$$

Finally (5) allows us to replace the subexpression length(append(11, 12)) on the right hand side by lengthof2(11, 12) getting

$$\text{lengthof2(cons}(n, 11), 12) = 1 + \text{lengthof2}(11, 12) \tag{7}$$

Thus we have produced two new equations, (6) and (7), which are true statements about lengthof2

$$\text{lengthof2(nil, 12)} = \text{length(12)}$$
$$\text{lengthof2(cons}(n, 11), 12) = 1 + \text{lengthof2}(11, 12)$$

what is more these two equations constitute a complete program for lengthof2 which is more efficient than the one given by (5), so we can replace (5) by (6) and (7).

More formally we have the following transformation operators which act on equations and produce further equations

(i) *Definition.* Introduce a new recursion equation whose left hand expression is not an instance of the left hand expression of any previous equation.

(ii) *Instantiation.* Introduce a substitution instance of an existing equation.

(iii) *Unfolding.* If $E = E'$ and $F = F'$ are equations and there is some

occurrence in F' of an instance of E, replace it by the corresponding instance of E' obtaining F'', then add the equation $F=F''$.

(iv) *Folding.* If $E=E'$ and $F=F'$ are equations and there is some occurrence in F' of an instance of E', replace it by the corresponding instance of E obtaining F'', then add the equation $F=F''$.

(v) *Abstraction.* We may introduce a **where** clause, by deriving from a previous equation $E=E'$ a new equation

$$E=E' \ [u1/F1, \ldots, un/Fn]$$

$$\textbf{where } \langle u1, \ldots, un \rangle = \langle F1, \ldots, Fn \rangle$$

($E[E1/E2]$ means E with all occurrences of subexpressions $E2$ replaced by $E1$.)

(vi) *Laws.* We may transform an equation by using on its right hand expression any laws we have about the primitives (associativity, commutativity, etc.) obtaining a new equation.

Strictly these laws apply only to first order programs in HOPE but their extension to higher order functions is only technical and we will not use these extensions in this paper.

These rules, which have become known as the unfold/fold system preserve the meaning of any program they are applied to except that they may make a program fail to terminate when it terminated before, i.e. they preserve partial correctness. [21] contains a theoretical study of this system and rules for avoiding non-termination.

2. Transformation Capabilities

The transformation system introduced in the previous section is able to achieve a wide variety of improvements. However, when we come to consider transformation as a practical software tool, transformations considered solely in terms of the rules outlined above become very detailed. In this section we outline several important high level transformation types in terms of which transformations can be planned and explicated. However, it is important that all these transformations ultimately rest for their implementation on the simple rules outlined above. Thus the correctness of each transformation is ultimately guaranteed by the correctness of the basic rules.

(i) *Loop combinations.* Programs written as specifications tend to have many independent computations kept apart for reasons of clarity. One of the main transformation tasks is to interweave these separate iterations whenever possible.

Two main sorts of loop combinations can be identified. We have the case where one inner loop builds up a data structure that an outer one traverses, these can often be compressed to a single loop removing the need for the intermediate structure. Alternatively we may have two or more similar loops in separate parts of a program which can be brought together and executed as one loop. Further details of these transformations can be found in [4,9].

(ii) *Automatic implementation of abstract data types.* Abstract data types provide a powerful tool for structuring program development. Guttag [17] shows how the behaviour of such abstract data types can be specified equationally before any implementation is considered, allowing programs to be developed at suitable conceptual levels. The use of a functional language and transformation allows this technique to be exploited to the full. Within a functional language the equational specification often constitutes a preliminary implementation allowing 'abstract' programs employing the data type (for example priority queues) to be tested in isolation. When a designer is satisfied with this program and decides to proceed to an implementation in a more machine oriented data type (say for example binomial trees) using transformation all he would have to do is write a simple mapping function showing how he intends to use the lower data type (binomial trees) to represent the higher (priority queues). Given this information efficient implementations for all the higher level functions can be produced automatically using the transformation techniques outlined in [8], an extended example being given in [25].

(iii) *Synthesis.* Viewing HOPE as a specification vehicle there is a continuous spectrum of increasingly inefficient programs. For example using the set notation one may easily define sets that are of infinite cardinality or not explicitly constructable. These are however legitimate HOPE programs and can be converted to runnable versions using the methods outlined earlier. Another form of specification that is very useful is that of general equations i.e. equations with several functions on the left hand side defining some function implicitly. For example say we have a

function f defined normally, then its inverse $f$inv can be defined implicitly using the equation

$$f\text{inv}(f(x)) = x$$

Such an equation cannot be run in HOPE, however, transformation can be used to convert it to an explicit definition of $f$inv that can be run.

Further details of these applications can be found in [7,8].

(iv) *Computation sequence re-organisation.* Transformation can be used to re-arrange the order in which operations are performed during a computation. For example from a version of factorial defined thus

$$\text{fact}(0) = 1$$
$$\text{fact}(n+1) = (n+1) * \text{fact}(n)$$

alternative versions can be produced that carry out the computation in different orders, of these perhaps the most important is the iterative version viz.

$$\text{fact}(n) = \text{factit}(n, 1)$$
$$\text{factit}(0, \text{acc}) = \text{acc}$$
$$\text{factit}(n+1, \text{acc}) = \text{factit}(n, n+1 * \text{acc})$$

which can be translated directly to an imperative program using a while loop as explained in [23].

Further details of these transformations are in [4,9].

(v) *Structure sharing.* Conventional languages that include assignment and explicit control over storage allocation allow the user to alter structures in place and affect changes using side effects. This style of programming is recognised as being very efficient but also notoriously difficult to understand and prone to error. Transformation allows this sort of behaviour to be introduced in a controlled and systematic way. Starting from a specification in a functional language (which of course cannot employ side effects) the way storage is used can be considered and optimised, introducing side-effects guaranteed not to change the meaning of the program. Details can be found in [25].

3. A Transformation Meta-language

The idea of using a meta-language to control a general theorem proving system was first used in the LCF Project [15]. Here a meta-language, ML, was developed to allow the writing of structured plans to assist in the proving of theorems about programs. This meta-language idea was first applied in the program transformation context by Feather [12]. Here the meta-language was not a full programming language but a collection of methods whereby a user could guide a general transformation system based on the unfold/fold methodology.

Our approach builds on both these projects. We have chosen HOPE itself to be our meta-language. Thus the objects manipulated by our HOPE meta-language are HOPE object programs and the operators of the meta-language, or tactics in the LCF terminology, act on HOPE object programs and return HOPE object programs.

(This rather reflexive decision has several advantages, not least of which is the fact that we do not regress into a language definition project. By defining HOPE abstract structures within HOPE and writing a HOPE parser in HOPE we have the skeleton of a HOPE compiler. These benefits apart HOPE has proved a good choice for a meta-language having powerful structure defining and manipulating facilities.)

3.1. *First level tactics*

Our basic tactics, out of which all others are written, consist of the basic rules of the unfold/fold system. Thus we define the following HOPE distfix operators that work on HOPE object programs

```
distfix instantiate __ occuring-in __ with __
dec instantiate:variable # equation # expression→equation
distfix unfold __ using __
dec unfold:equation # equation→equation
distfix fold __ with __
dec fold:equation # equation→equation
distfix abstract __ within __
dec abstract:expression # equation→equation
distfix rewrite __ as __ because __
dec rewrite:equation # equation # rule→equation
distfix define __ by __
```

dec define: set(function-name) # set(equation)→set(equation)
distfix replace __ within __ by __
dec replace: function-name # program # set(equation)→program

Most of these operators should be understandable from their previous definition. All work on equations returning equations, except define and replace. Define is really an identity function just returning the set of equations defining the functions named in the first argument. Replace is the only operator that allows us to alter programs. Programs are just sets of equations and replace acts upon the program given in the second argument by replacing those equations defining the function named in the first argument by the equations given in the third argument.

The use of some of these operators can be seen in the following teletype session showing them being used to perform the optimisation used as a simple example in [4]. In the meta-language there are two ways one may refer to object program equations. One may give the whole equation in quotes and parse it using the function eqparse (available as ⟨: :⟩ distfix operators) or one may select an equation from a program by giving the left hand side in quotes using the distfix operator eqn __ within __, e.g. either

⟨: "fact($n+1$) = $n+1$ * fact(n)" :⟩

or

eqn "fact($n+1$)" within P

evaluate to the same equation.

```
let append ==
        define {"append","g"}
        by {"append(nil,y) = y",
            "append(cons(x,y),z) = cons(x, append(y,z))",
            "g(x,y,z) = append(append(x,y),z)"}
in
        replace "g(x,y,z)" within append
        by {unfold
              instantiate "x" occuring-in
              eqn "g(x,y,z)" within append
              with "nil"
            using
                eqn "append(nil,y)" within append,
```

fold
unfold
instantiate "x" occuring-in
eqn "$g(x, y, z)$" within append
with "cons(u, v)"
using eqn "append(cons$(x, y), z)$" within append
with eqn "$g(x, y, z)$" within append};

GIVING
append(nil, y) = y
append(cons$(x, y), z$) = cons$(x$, append(y, z))
g(nil, y, z) = append(y, z)
g(cons$(u, v), y, z$) = cons$(u, g(v, y, z))$

3.2. *Second level tactics*

The operators outlined above are adequate to describe a wide range of different transformations. However, as explained earlier, significant transformations expressed solely at this level are very cumbersome. The next step in expanding the expressive power of our transformation meta-language is to define a set of higher level operators. These correspond to the different types of transformation outlined in Section 2. The important point is that these higher level operators are written in terms of the lower level ones and when applied attempt to construct a transformation step of the required type in terms of the unfold/fold operators. These tactics can thus fail in that they are unable to perform the required transformation but can never produce a wrong program as they are working within a correctness preserving formalism.

Each of the transformation types in Section 2 has a set of heuristics to guide the transformation tactic and a set of criteria to judge its success or failure. Thus they involve a limited amount of search but this search is confined to within the tactic avoiding a combinatorial explosion. We can cut down the amount of search needed by giving more information in the tactic.

Thus for example corresponding to the merge loops of Section 2(i) we have the following tactic

distfix mergeloops for __ on __ within __
dec mergeloops: equation # variable # program→program

The previous example can now be done with a single meta-language command

let P == mergeloops for eqn "$g(x,y,z)$" within append
on "x" within append

The code for mergeloops instantiates the variable mentioned according to its type and then performs a series of unfolds followed by folds to express the composition as a single direct recursion.

The syntax of some of the other operations is

(i) Change data type

distfix implement __ within __ using __ name __
dec implement: function __ name # program # function __ name # function __ name→program.

The first parameter names the 'abstract' function for which a 'concrete' version is required. The third parameter names the representation function that shows how one wishes to represent this abstract data type and the final parameter is the name one wishes to give to the concrete function to be synthesised. The result, if successful, is the original program augmented with equations for the concrete function.

Thus using the example from [4] given that the following program, P, had been defined

data pair == nilp ++ conspi(num,pair) ++ conspp(pair,pair)
data tree == niltree ++ constree(num,tree,tree)

dec rep: pair→tree
dec twist: tree→tree

twist(niltree) = niltree
twist(constree(i, $t1$, $t2$)) = constree(i, twist($t2$), twist($t1$))

(Returns mirror image of tree)

rep(nilp) = niltree
rep(conspi(i, conspp($t1$, $t2$))) = constree(i, rep($t1$), rep($t2$))

(Simple representation of a labelled tree in terms of pairs.)

the meta-language command is

let P == implement twist within P using rep name conctwist

and the result is P augmented with the equations

$$\begin{aligned}&\text{conctwist(nilp)} = \text{nilp}\\&\text{conctwist(conspi(}a,\ \text{conspp(}p1,p2\text{)))}\\&\qquad\qquad = \text{conspi(}a, \text{conspp(conctwist(}p2\text{), conctwist(}p1\text{)))}\end{aligned}$$

(ii) Change computation sequences. The tactic to convert a function defined recursively into an iterative version is

distfix convert-to-iteration __ within __ using __ name __
dec convert-to-iteration: function __ name # program # equation
function __ name→program

The first parameter names the function for which an iterative version is required. The third is the equation defining the iterative form implicitly in terms of the recursive form, the 'eureka' step of [4], and the last parameter is the name to be given to the iterative form. The result, if successful, is an enhanced program containing equations, in iterative form, for the derived function. Thus given a program, P, for factorial similar to the one defined in Section 2 the meta-language command to convert this to an iterative form is

let P == convert-to-iteration factorial within P
using ⟨: "factit(n, acc) = acc * factorial(n)" :⟩
name factit

(iii) Synthesise. The tactic to produce a runnable program for a function defined implicitly is

distfix synthesise __ within __ using __
dec synthesise: function __ name # program #
equation→program

The first parameter names the function which is defined implicitly by the equation given as the last parameter. The result, if successful, is a program containing equations for the function that can be run.

For example if a program P contains equations for a function f its inverse, $f$inv, can be produced via the meta-language command

let P == synthesise $f$inv within P using ⟨: "$f$inv($f(x)$) = x" :⟩

(iv) Structure sharing. This is accomplished via the tactic

distfix make-destructive __ within __ on __ name __

```
dec make-destructive: function __ name # program # variable
                        # function __ name→program
```

The effect of this tactic is to attempt to produce a version of the named function that overwrites the argument named.

3.3. *Paradigm algorithms*

It is becoming increasingly recognised that there are not all that many totally different fundamental algorithms and that many seemingly different algorithms are just variations on a single theme. Correspondingly advice on how to program a solution to a particular problem often consists of simply naming a general technique thought appropriate. Given such advice programmers are usually fairly adept at instantiating the particular paradigm to fit their circumstances. It would be nice to be able to instruct our meta-language system in such a direct way. In this section we will outline a tentative approach that goes some way towards providing this capability. The technique we will use owes a great deal to the ideas put forward by Backus and others concerning the power of higher order functions, and borrows directly from an example given by Ronan Sleep. An allied approach is outlined in [14].

We will use as an example binary search. This is perhaps the simplest of all paradigm algorithms. The central idea is very simple, if one is searching a structure for an occurrence of an item performing a binary search consists of splitting the structure in half and deciding after each split that one half can be discarded as it could not contain the sought after item. The splitting is continued until the item is found whereupon success is indicated together with some value computed from the item (e.g. the information stored under some particular key) or until the structure cannot be decomposed further whereupon failure is indicated. We see that this algorithm depends upon being able to split a structure roughly in half and being able to decide in which half to continue searching. This idea of a generalised binary search can be captured by the following higher order HOPE function g__b__s.

```
dec g__b__s: alpha # beta # (alpha→truval) # (alpha→alpha # alpha)
                # (alpha # alpha # beta→truval) #
                (alpha # beta→truval # gamma)

                →truval # gamma
```

The meaning of the various arguments in

g__b__s(str, *s*, stop, split, decision, test)

is

str	is the structure being searched (of type alpha)
s	is the item we are looking for (of type beta)
stop	is the test that tells us when we can decompose the structure no more
split	is the function that decomposes our structure
decision	is the test that tells us in which of the two structures resulting from the decomposition to continue searching
test	tells us whether we have found the item we are looking for and if so returns a computed value (of type gamma)

g__b__s can be defined thus,

g__b__s(str, *s*, stop, split, decision, test) =

```
let tv, v == test(str, s)
    in tv, v if tv
      else false, undefined if stop(str)
      else let str1, str2 == split(str)
        in g__b__s(str1, s, stop, split, decision, test)
                                          if decision(str1, str2, s)
            else g__b__s(str2, s, stop, split, decision, test)
```

Producing a particular realisation of this generalised binary search can be achieved by instantiating the functional parameters. This is a two stage process. Firstly stop and split depend on the particular structure we wish to perform binary search over and secondly decision and test are dependent on the particular problem we are applying binary search to, in particular what kind of total order we have over the structure. Thus we are expanding the notion of structure and expect to be able to access within the module defining the structure the appropriate functions for stop and split, a concept already present in CLU [19] and CLEAR [5].

Our meta-language command therefore is

```
distfix use __ over __ on __ within __
dec use: algorithm-name # structure-name # function-name
            # program→program
```

The first parameter names some general algorithm for which we have a generic form as above, the second parameter names a structure appropriate to this algorithm and the third a function that it is desired to implement using this algorithm. The result, if successful, is a version of the function employing an instance of the particular paradigm algorithm. This tactic would work by first combining the particular generic form with the structure named instantiating as many of the functions as required and then fitting the resulting structure specific algorithm to the problem mentioned.

To return to our example of binary search, say in a program P we had the following simple function over arrays

$$\text{search}(a, i) = m \textbf{ st } 1 \leq m \leq \text{size}(a) \ \& \ a[m] = i$$

and we know that a is ordered in ascending sequence. To implement this using binary search the meta-language command could be

use g__b__s over arrays-with-interval on search within P

arrays-with-interval is the name of a structure consisting of a triple of an array and two integers, the lower and upper bounds respectively. For this structure the functions stop and split are

$$\text{stop}(a, i, j) = j \leq i$$
$$\text{split}(a, i, j) = (a, i, \text{mid}),\ (a, \text{mid}, j)$$
$$\textbf{where } \text{mid} == \left\lfloor \frac{i+j}{2} \right\rfloor$$

($\lfloor n \rfloor$ is n rounded down to the nearest integer)

Having thus specialised g__b__s the implementation of the use meta-language operator must work out decision and test. Test could simply be

$$\text{test}(a, i, j, s) = \text{true},\ j \textbf{ if } a[j] = \text{s}$$
$$\textbf{else false, undefined.}$$

To produce decision the operator must first show that searching in one of the substructures must always return false and then produce code to decide which it is. It does not require great knowledge of total orderings to come up with

$$\text{decision}(a1, i1, j1, a2, i2, j2, s) = a1[j1] > s$$

Having produced our required functions in meta-language operator would replace the body of search in P by a call to g__b__s using the above functions.

We must again emphasise that the ideas presented in this section are tentative and we are not yet at the stage to start implementing the use tactic. However this approach does feel to correspond to our intuitive notions about algorithms and structures.

3.4. *General strategies*

The meta-language being a full programming language (HOPE) should enable us to program development strategies that apply over a wide class of programs. For example the techniques used by Feather in his program transformation system [12] can be implemented using the apparatus we have developed. One could also perhaps program a strategy corresponding to an applicative version of the Jackson Design Technique [18] that has been so successful in the area of commercial data processing.

4. An Example, Hamming's Problem

In this section we would like to show some of the apparatus introduced earlier at work on a simple problem. The one we have chosen is fairly well known, having been discussed in [11]. We want to produce a list of all the numbers that can be formed by multiplying 2, 3 and 5 together any number of times and would like the list to be in ascending sequence.

4.1. *Specification*

Our initial specification is the following HOPE program,

```
dec   hamming: →list(num)
dec   g: →set(num)
dec   order: set(num)→list(num)
dec   min: set(num)→num
dec   −: set(num) # num→set(num)
infix −: 6
```

$\text{hamming} = \text{order}(g)$

$g = \{2^i 3^j 5^k \mid 0 \leq i, j, k\}$

$\text{order}(S) = \min(S) :: \text{Order}(S - \min(S))$

$\min(S) = s$ **st** $s \in S$ & $\forall s1 \in S \quad s \leq s1$

$S - s = \{s1 \mid s1 \in S \;\&\; s1 \neq s\}$

Note that g is an infinite set and hamming an infinite list. Such structures are perfectly acceptable in languages with lazy evaluation [13]. However, in the above program we ask for the minimum of an infinite set so min would never terminate. Nevertheless such a program is perfectly acceptable as a specification.

4.2. *Meta-language program*

We next present the meta-language program that will take the above specification and produce a runnable program. Although we are forced to present things linearly we are not claiming that the development can be understood solely by examination of the meta-language program. This can best be gained by examining intermediate forms of the program a process best aided by a VDU rather than paper.

There are three main steps in the development corresponding to three main ideas in the final algorithm. We first unfold the definition to produce the initial value of the list, 1, and re-arrange the set g into the union of three sets that are multiplied throughout by 2, 3, 5 respectively. We then promote the ordering process into the creation of these sets. This is done by introducing two functions: merge a refinement of order that works on ordered lists instead of sets and $*1$ an infix operator over ordered lists that corresponds to the operation of multiplying every element of a set by a given number. Merge and $*1$ are introduced via equations defining them implicitly and the last two steps of the meta-language program synthesise runnable versions of them.

In the following meta-language program we have ommitted type declarations, all new meta-language operators are used in distfix form.

transformhamming $P =$

let newhamming == expand hamming within P

in

let newhamming == fold newhamming
with ⟨: "merge(order($S1$), order($S2$), order($S3$))
= order($S1 \cup S2 \cup S3$)" :⟩

in

let newhamming == fold newhamming
with ⟨: "$n * 1$ order(S) = order($\{n, s \mid s \in S\}$)" :⟩

in

let newhamming == fold newhamming with eqn "hamming" within P

in

let P == synthesise merge within P
using ⟨: "merge(order($S1$), order ($S2$), order($S3$))
= order($S1 \cup S2 \cup S3$)" :⟩

in

let P == synthesise $*1$ within P
using ⟨: "$n * 1$ order(S) = order($\{n, s \mid s \in S\}$)" :⟩

in replace hamming within P by {newhamming}

expand ham within P =

let ham ==
unfold
unfold
unfold
unfold ham
using eqn "order" within P
using ⟨: "min(g) = 1" :⟩
using eqn "g" within P
using eqn "-" within P

in

fold
rewrite ⟨: "hamming = 1::order($\{2^i 3^j 5^k \mid 0 \le i, j, k \& i, j, k \ne 0\}$)" :⟩

as ⟨: "hamming = 1 :: order($2 * s\{2^i 3^j 5^k \mid 0 \le i, j, k\} \cup$
$3 * s\{2^i 3^j 5^k \mid 0 \le i, j, k\} \cup$
$5 * s\{2^i 3^j 5^k \mid 0 \le i, j, k\}$)
where $n * s \quad S = \{n, s \mid s \in S\}$" :⟩

because setrule.
with eqn "g" within P

4.3. *Final program*

The program produced by the application of the above meta-language to the specification of Section 4.1 is essentially the one given in [20]. In HOPE this is,

hamming = 1 :: merge(2 *1 hamming, 3 *1 *hamming*, 5 *1 *hamming*)

merge($m1 :: L1, m2 :: L2, m3 :: L3$)

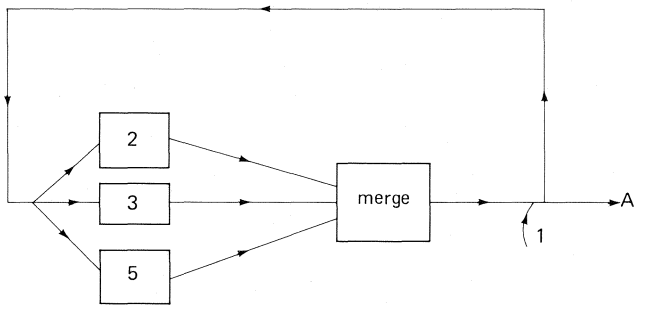
$$= \min :: \text{merge}(\text{remove}(\min, m1 :: L1), \text{remove}(\min, m2 :: L2), \text{remove}(\min, m3 :: L3)) \textbf{ where } \min == \text{minof3}(m1, m2, m3)$$

$n *1 (m :: L) = n . m :: (n *1 L)$

$\text{remove}(m, n :: L) = L$ **if** $m = n$ **else** $n :: L$

$\text{minof3}(m1, m2, m3) = m1$ **if** $m1 \leq m2$ & $m1 \leq m3$ **else** $m2$ **if** $m2 \leq m1$ & $m2 \leq m3$ **else** $m3$

The behaviour of this program can best be understood pictorially in terms of the Kahn, MacQueen communicating processes. Here streams, corresponding to the list in the above program, connect process boxes.



The 2, 3 and 5 boxes remove the number at the head of their input stream multiply it by 2, 3 or 5 respectively and pass it on. The merge box passes on the smallest of the numbers at the head of its three input streams removing any duplicates from the head of the other input stream. Thus if 1 is injected

into the system as shown the required infinite stream of numbers is produced at A.

4.4. *Detailed transformation*

In this section we show the detailed transformation induced by the meta-language program of Section 4.2 that takes the specification of Section 4.1 to the runnable program of Section 4.3.

We will write the top-level meta-language command on the left and the detailed steps of the transformation on the right.

expand hamming within P

$$\begin{aligned}
\text{hamming} &= \text{order}(g) \\
&= \min(g) :: \text{order}(g - \min(g)) && \text{Unfolding order} \\
&= 1 :: \text{order}(\{2^i 3^j 5^k \mid 0 \le i, j, k\} - 1) && \text{unfolding } g \\
& && \text{rewriting } \min(g) \text{ as } 1 \\
&= 1 :: \text{order}(\{2^i 3^j 5^k \mid 0 \le i, j, k \ \&\ i, j, k, \neq 0\}) && \text{unfolding } - \\
&= 1 :: \text{order}(2 *s \{2^i 3^j 5^k \mid 0 \le i, j, k\} \cup \\
&\qquad 3 *s \{2^i 3^j 5^k \mid 0 \le i, j, k\} \cup \\
&\qquad 5 *s \{2^i 3^j 5^k \mid 0 \le i, j, k\}) \\
&\qquad \textbf{where } n *s \quad S = \{n.s \mid s \in S\} && \text{rewriting} \\
&= 1 :: \text{order}(2 *s\ g \cup 3 *s\ g \cup 5 *s\ g) \\
&\textbf{where } n *s \quad S = \{n.s \mid s \in S\} && \text{folding with } g
\end{aligned}$$

fold newhamming
with ⟨: "merge(order($S1$), order($S2$), order ($S3$))
$= \text{order}(S1 \cup S2 \cup S3)$" :⟩

$$\text{hamming} = 1 :: \text{merge}(\text{order}(2 *s\ g), \text{order}(3 *s\ g), \text{order}(5 *s\ g))$$
$$\textbf{where } n *s\ S = \{n.s \mid s \in S\}$$

fold newhamming
with ⟨: "n *1 order (S) = order($\{n.s \mid s \in S\}$)" :⟩

$$\text{hamming} = 1 :: \text{merge}(2 *1 \text{ order}(g), 3 *1 \text{ order}(g), 5 *1 \text{ order}(g))$$

fold newhamming with eqn "hamming" in P

$$\text{hamming} = 1 :: \text{merge}(2 * 1\ \text{hamming}, 3 * 1\ \text{hamming}, 5 * 1\ \text{hamming})$$

synthesise merge within P

using ⟨: "merge(order ($S1$), order($S2$), order($S3$))
$= \text{order}(S1 \cup S2 \cup S3)$" :⟩

$\text{RHS} = \text{order}(S1 \cup S2 \cup S3)$

$= m :: \text{order}((S1 \cup S2 \cup S3) - m)$
where $m == \min(S1 \cup S2 \cup S3)$

Unfolding order

$= m :: \text{order}((S1 - m) \cup (S2 - m) \cup (S3 - m))$
where $m == \text{minof3}(\min(S1), \min(S2), \min(S3))$

Properties of min, $\cup$

$= m :: \text{merge}(\text{order}(S1 - m), \text{order}(S2 - m), \text{order}(S3 - m))$
where $m == \text{minof3}(\min(S1), \min(S2), \min(S3))$

Folding with definition of merge

$= m :: \text{merge}(\text{rem}(m, \text{order}(S1)), \text{rem}(m, \text{order}(S2)), \text{rem}(m, \text{order}(S3)))$
where $m == \text{minof3}(\min(S1), \min(S2), \min(S3))$

Properties of order, min

$= m :: \text{merge}(\text{rem}(m, \text{order}(S1)), \text{rem}(m, \text{order}(S2)), \text{rem}(m, \text{order}(S3)))$
where $m == \text{minof3}(\text{head}(\text{order}(S1)), \text{head}(\text{order}(S2)), \text{head}(\text{order}(S3)))$

folding with order $(\text{head}(n :: 1) = n)$

Thus LHS = RHS i.e.

$\text{merge}(\text{order}(S1), \text{order}(S2), \text{order}(S3))$
$= m :: \text{merge}(\text{rem}(m, \text{order}(S1)), \text{rem}(m, \text{order}(S2)), \text{rem}(m, \text{order}(S3)))$
where $m == \text{minof3}(\text{head}(\text{order}(S1)), \text{head}(\text{order}(S2)), \text{head}(\text{order}(S3)))$

$\text{merge}(OS1, OS2, OS3)$
$= m :: \text{merge}(\text{rem}(m, OS1), \text{rem}(m, OS2), \text{rem}(m, OS3))$
where $m == \text{minof3}(\text{head}(OS1), \text{head}(OS2), \text{head}(OS3))$

Generalising

Synthesise $*1$ in P
  using ⟨: "$n *1 \text{ order}(S) = \text{order}(\{n.s \mid s \in S\})$" :⟩

$$\text{RHS} = \text{order}(\{n.s \mid s \in S\})$$

$$= m :: \text{order}(\{n.s \mid s \in S\}) - m)$$
  where $m == \min(\{n.s \mid s \in S\})$

Unfolding order

$$= n.m :: \text{order}(\{n.s \mid s \in S\}) - n.m)$$
  where $m == \min(S)$

Fact about min

$$= n.\min(S) :: \text{order}(\{n.s \mid s \in (S - \min(S)))\})$$

$$= n.\min(S) :: (n *1 \text{ order}(S - \min(S)))$$

Folding with definition of $*1$

$$\text{LHS} = n *1 (\min(S) :: \text{order}(S - \min(S)))$$

Thus

$$n *1 (\min(S) :: \text{order}(S - \min(S))) = n.\min(S) :: (n *1 \text{ order}(S - \min(S)))$$

$$n *1 (m :: OS) = n.m :: (n *1 OS)$$

Generalising

Thus when the new equations developed are used to replace the old equation for hamming in P we have the program given in Section 4.3. We hope the reader will agree that the detailed derivations shown above although tedious do not require any great insights once the overall structure of the development has been set out by the meta-language program. At a few points rewritings dependent on the properties of the functions were used but these were not very deep theorems. Although we could not expect the machine to discover these facts for itself we could hope that it could verify that they were in fact true. Note the great similarity in the structure of the transformations for the two synthesise operations which lends credence to the claim that this level of problem solving can be largely left to the machine.

5. Advanced Applicative Programming Environments

There is a growing interest in the development of systems to support the development and maintenance of large software projects [22,26]. A transformation system, particularly a meta-language driven one, would seem to form a natural component of such a system.

One advantage of writing program specifications in a functional language is that in many cases these specifications can be run, albeit slowly, to test that they do in fact specify what is required before proceeding to an implementation. We also consider that it has been demonstrated that the automatic or semi-automatic verification of programs is much easier if these programs are written in a functional language, see for example [2]. In fact within the functional languages there is a continuous spectrum, running from execution through symbolic execution to formal proof and we can envisage a set of tools that would enable a user to establish, with more or less certainty, that his specification has all the desired properties.

Having satisfied himself with the specification the task of the designer would be to plan the transformations needed to take this specification to a runnable program and write the meta-language program described earlier. The system would then carry out the transformation, possibly providing estimates of the efficiency of the programs produced. After an acceptably efficient program has been produced the specification and successful meta-language program would be stored. There should never be any need for any one to examine the code of the final program.

The fact that the specification is written for maximum clarity and understandability should have important consequences when we come to consider the question of maintenance, modification and portability. Any changes required to meet changing circumstances would of course be made to the specification, which being more modular than conventional programs should greatly reduce the likelihood of errors being introduced. Unless the change is a major one it is likely that the original meta-language program will still achieve an efficient program without further user intervention. Even if modification is required to the meta-language program there is no possibility of introducing error. Changes in the implementation of a given specification, say for the purposes of implementing on a different target machine, would be achieved by suitably modifying and re-running the meta-language program.

Just as conventional programs can share functions or procedures so

meta-language programs can share sub-programs or tactics in the LCF terminology. Thus a 'tool box' of generally applicable tactics can be built up to collect and codify knowledge about algorithm design, giving the designer more and more powerful tools for program development. As the system becomes more powerful so the vocabulary available to the designer would become more sophisticated.

Another advantage of working totally within applicative languages is that these are much more suited to the new generation of parallel architectures that are being developed (see for example [10, 16]). We feel that the arrival of these machines would go a long way towards making the languages and ideas discussed here practical.

In the future we hope to extend the specification/transformation approach to include earlier parts of the software life cycle. We would do this by providing specialised user languages and transformation systems. These would enable an application specialist to state his requirements precisely, but in his own terms, and then have these converted to runnable programs. It would be possible to provide such languages and systems, for particular domains, right now. However, we hope to achieve this in general by developing ways of systematically extending a general specification language to a specialised requirements language and at the same time extending the transformation system to cope with requirements written in the language provided. Thus we envisage that 'programs' in the future will, more and more, be written by the people who originally conceive of the need for these programs and that the work of present day systems analysts or designers will consist of providing the specialised languages and systems that enable such programs to be specified and efficiently implemented.

6. Conclusion

At the time of writing, July 1981, the status of the meta-language is that after only 2 months of work all the first level tactics have been implemented and design work is proceeding on the second level ones. We have produced a HOPE in HOPE parser which is being extended to provide various HOPE compilers particularly one for our parallel machine. We do not see any great impediment to implementing the second level tactics and we look forward to using this richer set of operators to conduct experiments in program development and maintenance. We have conducted studies in

expressing the development of several 'classical' algorithms using the meta-language including topological sort, longest upsequence, and the Fisher-Galler algorithm.

We hope that in this and related papers we have gone some way towards convincing readers that programming can progress from being an art to a science and be formalised sufficiently to allow at least semi-automation. Much remains to be done but we are convinced that the combination of applicative languages and transformation techniques offers the best hope for overcoming the problems that plague software development at the moment.

Acknowledgements

It is a great honour to be invited to present a paper at a conference in honour of Professor van Wijngaarden. As a relative newcomer I will leave it to others to document his substantial contributions to the study of algorithms and the development of higher level languages. As a recent member of IFIP WG2.1 I have very much enjoyed the friendly and constructive atmosphere he engenders. The work presented here grew out of the working group's investigations into the feasibility of developing languages, called 'Abstracto' and 'Constructo' respectively, suitable for communicating algorithms and their developments although of course it does not represent the working group's collective view.

This work owes a great deal to the ideas developed in the LCF project by Michael Gordon, Robin Milner and Christopher Wadsworth. My colleagues at Imperial College have been directly involved in much of this work particularly Ian Moor and Victor Wu who have been responsible for many of the ideas in the meta-language and all the implementation. Our work on transformation springs from very fruitful earlier collaboration with Rod Burstall who with David MacQueen and Don Sannella is responsible for the language HOPE used here. This work has been consistently supported by the UK SERC.

References

[1] J. Backus, Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs, Turing lecture, Comm. ACM 21 (8) (1978) 613–641.

[2] R. Boyer and J. Moore, Proving theorems about LISP functions, J. ACM 22 (1975) 129–144.

[3] R.M. Burstall, Design considerations for a functional programming language, Proc. of Infotech State of the Art Conference, Copenhagen (1977) 54–57.

[4] R.M. Burstall and J. Darlington, A transformation system for developing recursive programs, J. ACM (1977) 44–67.

[5] R.M. Burstall and J.A. Goguen, Putting theories together to make specifications, Proc 5th Joint Conference on Artificial Intelligence (1977).

[6] R.M. Burstall, D.B. MacQueen and D.T. Sanella, HOPE an experimental applicative language, Int. Rep. Dept. Computer Science, University of Edinburgh (1980).

[7] J. Darlington, Application of program transformation to program synthesis, Proc. Int. Symp. on Proving and Improving Programs, Arc et Senans, France (1975).

[8] J. Darlington, Synthesis of implementations for abstract data types, Rep. 80/4, Dept. of Computing, Imperial College London (1980).

[9] J. Darlington, An experimental program transformation and synthesis system, Artificial Intelligence J. 16 (1981) 1–46.

[10] J. Darlington and M. Reeve, ALICE a multi-processor reduction machine for the parallel evaluation of applicative languages, Proc. ACM/MIT Conf. on Functional Programming languages and Computer Architectures (1981).

[11] E.W. Dijkstra, A Discipline of Programming (Prentice-Hall, Englewood Cliffs, NJ 1976).

[12] M.A. Feather, 'ZAP' program transformation system, primer and users manual, Rep. No. 54, Dept. of Artificial Intelligence, University of Edinburgh (1979).

[13] D.P. Friedman and D.S. Wise, CONS should not evaluate its arguments, in: S. Michaelson and R. Milner (Eds.), Automata, Languages and Programming (Edinburgh University Press, 1976).

[14] G.H. Gonnet, A handbook of algorithms and data structures, Rep. CS-80-23, University of Waterloo, Canada (1980).

[15] M.J. Gordon, R. Milner and C. Wadsworth, Edinburgh LCF, Rep. CSR-11-77, (Part 1), Dept. of Computer Science, Edinburgh University (1977).

[16] J. Gurd, A. Watson and A. Glauert, A multi layered data flow computer architecture, Int. Rept., Dept. of Computer Science, University of Manchester (1978).

[17] J.V. Guttag, Abstract data types and the development of data structures, Comm. ACM 20 (1977) 397–404.

[18] M.A. Jackson, Principles of Program Design (Academic Press, London, 1975).

[19] B.H. Liskov, E. Moss, C. Schaffert, B. Scheifler and A. Snyder, CLU reference manual, Techn. Rep. MIT Laboratory for Computer Science (1979).

[20] G. Kahn and D. McQueen, Coroutines and networks of parallel processes, in: Information Processing 77 (North-Holland, Amsterdam, 1977).

[21] L. Kott, About a transformation system: a theoretical study, Proc. Third Symposium on Programming, Paris (1971).

[22] M.M. Lehmann, The environment of program development and maintenance-programs, programming and programming support, Rep. 81/2, Dept. of Computing, Imperial College London (1981).

[23] J. McCarthy, A basis for a mathematical theory of computation, in: P. Brafford and D. Hirschuerg (Eds.), Computer Programming and Formal Systems (North-Holland, Amsterdam, 1963).

[24] R.A. Milner, A theory of type polymorphism in programming, CSR-9-77, Dept. of Computer Science, University of Edinburgh (1977).

[25] I. Moor and J. Darlington, A formal synthesis of an efficient implementation for an abstract data type, Int. Rep., Dept. of Computing, Imperial College (1980).

[26] W.E. Riddle and R.E. Fairley, Software development tools, in: Proc. Pingree Park Workshop (Springer-Verlag, New York, 1980).

Algorithmic Languages, de Bakker/van Vliet (eds.)

HYPERLISP

Masahiko Sato and Masami Hagiya

Department of Information Science, Faculty of Science, University of Tokyo, Bunkyo-ku, Tokyo, Japan

A new programming language called HYPERLISP is presented, whose domain of symbolic expressions is mathematically neater than that of LISP. The semantics of HYPERLISP is defined in a strictly constructive manner. The correctness of a meta-circular interpreter for HYPERLISP is also provable by a constructive method.

0. Introduction

In this paper, we first introduce a new domain **S** of symbolic expressions (sexps, for short) which is mathematically neater than the classical domain of LISP symbolic expressions. All the sexps are constructed from the initial sexp 0 by successive applications of two pairing functions *cons* and *snoc*. Moreover, our domain **S** enjoys the set theoretic isomorphism $\mathbf{S} \simeq \mathbf{S} \times \mathbf{S} + \mathbf{S} \times \mathbf{S}$, while for that of LISP we have $\mathbf{S} \simeq \mathbf{A} + \mathbf{S} \times \mathbf{S}$ where **A** is the set of atoms. It then becomes possible to define *car* and *cdr* as total functions.

We then introduce a programming language which we call HYPERLISP. The language is LISP-like in the sense that any sexp is a meaningful HYPERLISP program. Hence, taking into account the possibilities of nontermination of evaluation and erroneous termination, the semantics of HYPERLISP will be given as a binary relation $eval \subset \mathbf{S} \times \mathbf{S}$ such that $eval(x, y)$ and $eval(x, z)$ implies $y = z$ (i.e., *eval* is a partial function: $\mathbf{S} \to \mathbf{S}$). The intended meaning of $eval(x, z)$ is that the sexp x is evaluated to z. We define eval formally as the least set satisfying a constructively given set of inductive clauses. Since inductive definition is the most basic way of definition in constructive mathematics, we think that, from the foundational point of view, this is the most unproblematic and fundamental way of defining the semantics of HYPERLISP. The practical and theoretical usefulness of our semantics may be well illustrated by the following facts:

(i) The semantics worked as a complete *specification* of the language in the implementation of the interpreter.

(ii) The *correctness* of an interpreter (written in HYPERLISP) is provable in a constructive manner.

The domain **S** has an interesting algebraic structure which we cannot explain here due to the limitation of space. For this, as well as for a more detailed exposition of the syntax and semantics of HYPERLISP, we refer the reader to [8].

1. Sexp

1.1. *Definition of a sexp*

Imagine an infinite leaf-free binary tree like Fig. 1, where a small circle is drawn at each node. The topmost node is called the *root.*

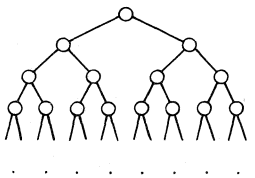


Fig. 1.

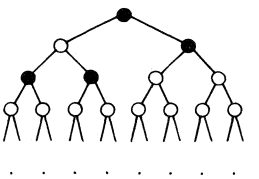


Fig. 2.

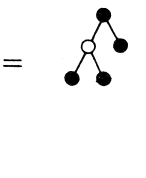


Fig. 3.

Choose a *finite* number of nodes arbitrarily and mark them black as in Fig. 2. The resulting figure is called a *sexp* (for *s*ymbolic *exp*ression). We assume those nodes that do not appear in the drawings are not marked. Since only finitely many nodes are marked, any sexp may be represented as a finite binary tree. Thus as a sexp, Fig. 2 is equal to Fig. 3.

The sexp with no marked nodes is denoted by 0. Fig. 1, considered as a sexp, is 0. The sexp whose only marked node is the root is denoted by 1.

We use $x, y, z, \ldots$, possibly indexed, as variables for sexps.

The set of all the sexps is denoted by **S**.

1.2. *Recognizer*

We define the predicate *atom*:

$$atom(x) \leftrightarrow \begin{cases} true & \text{if the root of } x \text{ is marked,} \\ false & \text{otherwise} \end{cases}$$

e.g. *atom*(1) is true, but *atom*(0) is not.

We set

$$\mathbf{A} = \{x \in \mathbf{S} \mid atom(x)\},$$

$$\mathbf{M} = \mathbf{S} - \mathbf{A}.$$

An element of **A** is called an *atom* while an element of **M**, a *molecule.*

1.3. *Selectors*

We define *car, cdr*: $\mathbf{S} \to \mathbf{S}$

$$car\left(\begin{array}{c} \circ \\ x \quad y \end{array}\right) = car\left(\begin{array}{c} \bullet \\ x \quad y \end{array}\right) = x,$$

$$cdr\left(\begin{array}{c} \circ \\ x \quad y \end{array}\right) = cdr\left(\begin{array}{c} \bullet \\ x \quad y \end{array}\right) = y,$$

e.g.

$$car\left(\begin{array}{c} \bullet \\ \circ \quad \bullet \\ \bullet \quad \bullet \end{array}\right) = \begin{array}{c} \circ \\ \bullet \quad \bullet \end{array},$$

$$cdr\left(\begin{array}{c} \bullet \\ \circ \quad \bullet \\ \bullet \quad \bullet \end{array}\right) = \bullet = 1,$$

$$car(0) = cdr(0) = car(1) = cdr(1) = 0.$$

Remark that *car* and *cdr* are total functions on **S**.

1.4. *Constructors*

We define *cons, snoc*: $\mathbf{S} \times \mathbf{S} \to \mathbf{S}$

$$cons(x, y) = \begin{array}{c} \circ \\ x \quad y \end{array},$$

$$snoc(x, y) = \begin{array}{c} \bullet \\ x \quad y \end{array},$$

e.g.

$$cons(0,0)=0,$$

$$snoc(0,0)=1.$$

1.5. *Some properties*

(1) $car(cons(x,y))=car(snoc(x,y))=x,$
$cdr(cons(x,y))=cdr(snoc(x,y))=y.$

(2) $cons(x,y)\in\mathbf{M},$
$snoc(x,y)\in\mathbf{A}.$

(3) $x\in\mathbf{M}\rightarrow x=cons(car(x),cdr(x)),$
$x\in\mathbf{A}\rightarrow x=snoc(car(x),cdr(x)).$

(4) $cons:\mathbf{S}\times\mathbf{S}\rightarrow\mathbf{M}$ *is bijective,*
$snoc:\mathbf{S}\times\mathbf{S}\rightarrow\mathbf{A}$ *is bijective.*

(1)–(3) are obvious from the definition. (4) follows from (1)–(3). By (4) we have the following set theoretic isomorphisms:

(5) $\mathbf{A}\simeq\mathbf{S}\times\mathbf{S},$

$\mathbf{M}\simeq\mathbf{S}\times\mathbf{S},$

$\mathbf{S}\simeq\mathbf{A}+\mathbf{M}\simeq\mathbf{A}+\mathbf{S}\times\mathbf{S}\simeq\mathbf{S}\times\mathbf{S}+\mathbf{S}\times\mathbf{S}.$

Since in a sexp, only finitely many nodes are marked,

(6) *Every sexp can be constructed in terms of* 0
and a finite number of applications of cons and snoc.

E.g.

$=snoc(cons(snoc(0,0),snoc(0,0)),snoc(0,0)).$

By (6) we have the following induction schema:

(7) $$\frac{A(0)\quad A(x)\&A(y)\rightarrow A(cons(x,y))\&A(snoc(x,y))}{A(z)}.$$

1.6. *Notation*

Dot notation:

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

$$[x\,.\,y] = snoc(x, y).$$

List notation:

$$(x, y, z) = (x\,.\,(y\,.(z\,.\,0))),$$

$$[x, y, z] = [x\,.\,[y\,.\,[z\,.\,0]]].$$

Particularily,

$$(\,) = [\,] = 0.$$

Some auxiliary notations are prepared:

$$'x = [1, x],$$

$$x(\ldots) = (x, \ldots), \qquad x[\ldots] = [x, \ldots],$$

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

$'x$ has the highest precedence and $x : y$, the lowest: e.g.

$$'0[0, 0] : 1$$

$$= [1, 0][0, 0] : 1$$

$$= [[1, 0], 0, 0] : 1$$

$$= ([[1, 0], 0, 0], 1).$$

$x(\ldots)$ and $x[\ldots]$ associate to the left: i.e.

$$x(\ldots)(\ldots) = ((x, \ldots), \ldots),$$

$$x(\ldots)[\ldots] = [(x, \ldots), \ldots].$$

The intention of these notations will be clear in Section 2.5.

A semicolon may replace a comma:

$$[x; y; z] = [x, y, z].$$

1.7. *Literal*

Let **L** be the set of all the lowercase letters:

abcdefghijklmnopqrstuvwxyz

Let ϱ be an injection: $\mathbf{L} \rightarrow \mathbf{A}$. In this paper, we define ϱ as follows:

$$\varrho(\mathrm{a}) = [1,1,0,0,0,0,1],$$

$$\varrho(\mathrm{b}) = [1,1,0,0,0,1,0],$$

$$\cdots$$

$$\varrho(\mathrm{z}) = [1,1,1,1,0,1,0].$$

(The ascii code of a is 141 in octal, 1100001 in binary.)

A *literal* is a nonempty string of letters in **L**. A literal denotes a sexp as follows: let $l = \sigma_1 \cdots \sigma_n$ be a literal, where $\sigma_i \in \mathbf{L}$, then l denotes

$$[\varrho(\sigma_1), \ldots, \varrho(\sigma_n)].$$

We identify a literal and the sexp it denotes:

$$\mathrm{ab} = [[1,1,0,0,0,0,1],\ [1,1,0,0,0,1,0]],$$

$$(\mathrm{b}\ .\ \mathrm{c}) = ([[1,1,0,0,0,1,0]]\ .\ [[1,1,0,0,0,1,1]]).$$

Remark that a literal is an atom.

2. Eval

In this section, we define a binary relation on **S**, denoted by *eval*. *eval* will be a partial map: $\mathbf{S} \rightarrow \mathbf{S}$, in the sense that *eval*(x, y) and *eval*(x, z) implies $y = z$. *eval*(x, z) means that x is *evaluated* to z. We write $x \vdash z$ for *eval*(x, z).

The definition of *eval* is 'mutually recursive' with those of *apply, evlis* and *evcon*. *apply* is a tertiary relation on **S** and *evlis* and *evcon* are binary relations. *apply*(f, x, z) means that the function f applied to the argument list x yields z as its value. As was said in the introduction, their definitions (or rules) take the form of the inductive definition, so they may be hard to understand in the first reading. Those who wish to understand our intention first may skip to Section 2.5 and then come back here.

We assume the extremal clause in each of the following definitions.

2.1. *Eval*

(*e*1) $x \in \mathbf{A}, \quad apply(car(x), cdr(x), z) \rightarrow x \vdash z,$

(*e*2) $x \in \mathbf{M}, \quad evlis(cdr(x), y), apply(car(x), y, z) \rightarrow x \vdash z.$

This corresponds to the following ALGOL-like statement:

$eval(x)$

$= \textbf{if } x \in \mathbf{A} \textbf{ then } apply(car(x), cdr(x))$

elif $x \in \mathbf{M}$ **then** $apply(car(x), evlis(cdr(x)))$ **fi**.

2.2. *Evlis*

(*el*1) $evlis(0, 0),$

(*el*2) $x \neq 0, car(x) \vdash z_1, evlis(cdr(x), z_2) \rightarrow evlis(x, cons(z_1, z_2)).$

2.3. *Apply I*

(*a*1) $apply(0, x, 0),$

(*a*2) $apply(1, x, car(x)),$

(*a*3) $car(x) = car(cdr(x)) \rightarrow apply(\text{eq}, x, 1),$

(*a*4) $car(x) \neq car(cdr(x)) \rightarrow apply(eq, x, 0),$

(*a*5) $evcon(x, z) \rightarrow apply(\text{cond}, x, z),$

(*a*6) $car(x) \in \mathbf{A} \rightarrow apply(\text{atom}, x, 1),$

(*a*7) $car(x) \in \mathbf{M} \rightarrow apply(\text{atom}, x, 0),$

(*a*8) $car(x) = 0 \rightarrow apply(\text{null}, x, 1),$

(*a*9) $car(x) \neq 0 \rightarrow apply(\text{null}, x, 0),$

(*a*10) $apply(\text{car}, x, car(car(x))),$

(*a*11) $apply(\text{cdr}, x, cdr(car(x))),$

(*a*12) $apply(\text{cons}, x, cons(car(x), car(cdr(x)))),$

(*a*13) $apply(\text{snoc}, x, snoc(car(x), car(cdr(x)))).$

eq, cond, atom, null, car, ... are literals.

2.4. *Evcon*

(*ec*1) $evcon(0, 0)$,

(*ec*2) $x \neq 0, car(car(x)) \vdash y, y \in \mathbf{A}, car(cdr(car(x))) \vdash z \rightarrow evcon(x, z)$,

(*ec*3) $x \neq 0, car(car(x)) \vdash y, y \in \mathbf{M}, evcon(cdr(x), z) \rightarrow evcon(x, z)$.

2.5. *Properties and examples*

Since $car(car([x])) = car(x)$, we have $apply(\mathrm{car}, [x], car(x))$ by (*a*10). From this and (*e*1) we have $\mathrm{car}[x] \vdash car(x)$, since $car(\mathrm{car}[x]) = \mathrm{car}$ and $cdr(\mathrm{car}[x]) = [x]$.

(*car*) $\mathrm{car}[x] \vdash car(x)$.

Likewise

(*cdr*) $\mathrm{cdr}[x] \vdash cdr(x)$,

(*cons*) $\mathrm{cons}[x, y] \vdash cons(x, y)$,

(*snoc*) $\mathrm{snoc}[x, y] \vdash snoc(x, y)$,

(*eq*1) $x = y \rightarrow \mathrm{eq}[x, y] \vdash 1$,

(*eq*2) $x \neq y \rightarrow \mathrm{eq}[x, y] \vdash 0$.

Similar for atom and null. By (*a*2) *etc.*,

(*id*) $1[x] \vdash x$ i.e. $'x \vdash x$

For *evlis,* we have

(*l*) $x_i \vdash y_i (1 \leq i \leq n) \rightarrow \mathrm{evlis}((x_1, \ldots, x_n), (y_1, \ldots, y_n))$.

(*e*2) says that when the sexp to be evaluated is a molecule, its argument list (i.e. its cdr) should be evaluated by *evlis*:

(*e*) $x_i \vdash y_i (1 \leq i \leq n), f[y_1, \ldots, y_n] \vdash z \rightarrow f(x_1, \ldots, x_n) \vdash z$.

By (*a*1) etc.,

(*z*) $0 \vdash 0$.

For *evcon,*

(*c*1) $x_i \vdash z_i\ (1 \leq i \leq k), z_i \in \mathbf{M}\ (1 \leq i < k), z_k \in \mathbf{A}, y_k \vdash z$

$\rightarrow \mathrm{cond}[x_1 : y_1; \ldots; x_n : y_n] \vdash z,$

(*c*2) $x_i \vdash z_i \in \mathbf{M}\ (1 \leq i \leq n) \rightarrow \mathrm{cond}[x_1 : y_1; \ldots; x_n : y_n] \vdash 0.$

An atom represents truth while a molecule represents falsity.

From the above properties, the following evaluations may be obvious:

'a ⊢ a,

car[(b . c)] ⊢ b,

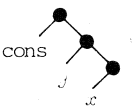
cons('a, car[(b . c)]) ⊢ (a . b),

cond[eq[a, a] : 0; '1 : '1] ⊢ 0,

cond[eq[a, b] : 0; '1 : '1] ⊢ 1.

2.6. *Lambda abstraction*

Consider $\lambda xy\,.\,cons(y,x)$. Let us represent this function by a sexp. First note that by (*cons*) in Section 2.5,

$cons(y,x] =$ 

$\vdash (y\,.\,x) = cons(y,x).$

But in cons[y,x], x and y are the variables for which the actual arguments are to be substituted. We try to represent cons[y,x] by two sexps as in Fig. 4.

Fig. 4.

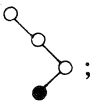
The right sexp represents the place of the variables, but it does not tell which variable is where. Remember that x is the first argument and y is the second. Since the first argument is identified as the car of the argument list,

it is represented by the sexp

the second argument is represented by


;

the third by


;

etc. Adding the information on variables to Fig. 4, we get Fig. 5.

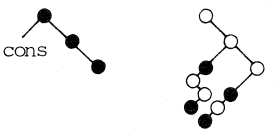


Fig. 5.

Finally, we get our *lambda expression,* the sexp named *xcons* in Fig. 6.

xcons =

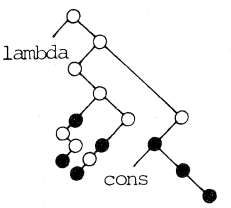


Fig. 6.

We expect for $x, y \in \mathbf{S}$

$$xcons[x, y] \vdash (y.x).$$

To realize this, we add the following rule:

$(a\lambda)$ $\quad f \in \mathbf{M}, \quad car(f) = \text{lambda}, \ subst(x, param(f), body(f)) \vdash z$

$$\rightarrow apply(f, x, z),$$

where *param*, *body* and *subst* are total functions defined as follows:

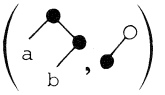
$$param(f) = car(cdr(f)),$$

$$body(f) = car(cdr(cdr(f))),$$

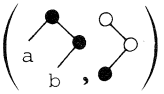
$$subst(x, p, b) = \textbf{if } p = 0 \textbf{ then } b$$
$$\textbf{elif } p \in \mathbf{A} \textbf{ then } point(x, car(p))$$
$$\textbf{elif } b \in \mathbf{M} \textbf{ then } cons(subst(x, car(p), car(b)),$$
$$subst(x, cdr(p), cdr(b)))$$
$$\textbf{elif } b \in \mathbf{A} \textbf{ then } snoc(subst(x, car(p), car(b)),$$
$$subst(x, cdr(p), cdr(b))) \textbf{ fi},$$

$$point(x, q) = \textbf{if } q = 0 \textbf{ then } 0$$
$$\textbf{elif } q \in \mathbf{A} \textbf{ then } x$$
$$\textbf{elif } cdr(q) = 0 \textbf{ then } point(car(x), car(q))$$
$$\textbf{elif } cdr(q) \neq 0 \textbf{ then } point(cdr(x), cdr(q)) \textbf{ fi}.$$

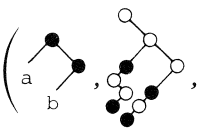
We did not define *subst* and *point* inductively for the sake of readability. $point(x, q)$ is used to extract a certain part (specified by q) from the argument list x. E.g.

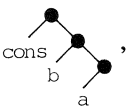
$point\left(\right.$ 

$\left.\right) = \mathrm{a},$

$point\left(\right.$ 

$\left.\right) = \mathrm{b},$

$subst\left(\right.$ 

$\left.\right) =$ 

,

$subst([\mathrm{a}, \mathrm{b}], param(xcons), body(xcons)) = \mathrm{cons}[\mathrm{b}, \mathrm{a}].$

Since cons[b, a] $\vdash$ (b . a), by ($a\lambda$) above

$$apply(xcons, [a, b], (b . a)),$$

$$xcons[a, b] \vdash (b . a).$$

In our lambda expressions, (bound) variables are literally anonymous so that the usual problem of avoiding the conflict of variables will never occur. But to really write down meaningful lambda expressions, the notation so far is too poor; we need more sophisticated notation to express lambda expressions. (See Section 2.8.)

2.7. *Label*

($a\Lambda$) $f \in \mathbf{M}$, $car(f) = \text{label}$,

$$apply(subst(f, param(f), body(f)), x, z) \rightarrow apply(f, x, z).$$

2.8. *Notation for lambda and label expressions*

xcons in Section 2.6 is denoted by

$$\lambda([X, Y]; cons[Y, X])$$

where cons[Y, X] expresses the function body and [X, Y] declares that X is the first argument and Y is the second. X and Y are called *metaliterals*. A metaliteral is an alphanumeric string beginning with an uppercase letter and plays the role of a bound variable. Obviously

$$\lambda([X, Y]; cons[Y, X]) = \lambda([U, V]; cons[V, U])$$

they both denote the same sexp, *xcons* in Fig. 6.

Examples:

$$\lambda([X, Y]; cons[Y, X])[a, b] \vdash (b . a),$$

$$\lambda([X, Y]; cons('X, car[Y]))[a, (b . c)] \vdash (a . b),$$

$$(\cdot . \cdot\ cons('a, car[(b . c)]) \vdash (a . b)),$$

$$\lambda([X, Y]; cond[eq[X, Y] : 0; '1 : '1])[a, a] \vdash 0,$$

$$\lambda([X, Y]; cond[eq[X, Y] : 0; '1 : '1])[a, b] \vdash 1.$$

[X, Y] above is called a *metaterm*; a metaterm declares metaliterals.

Examples of metaterms:

(1) [X],

(2) [X, Y],

(3) [[X1 . X2]],

(4) [X = [X1 . X2], Y].

In (3) X1 is the car of the first argument and X2 is the cdr. In (4) X is the first argument, X1 its car, X2 its cdr and Y is the second argument. $\alpha = \beta$ in a metaterm means that α and β occupy the same place in the argument list.

$\Lambda(\ldots)$ is for a label expression. E.g.

append = Λ(APPEND;
  λ([X, Y];
   cond[null[X]:'Y;
    '1:cons(car[X], APPEND(cdr[X], 'Y))]))

which is the ordinary append function; e.g.

append[(a, b, c), (d, e)] $\vdash$ (a, b, c, d, e).

Using the metaterm (4) above, let

$append_2$ = Λ(APPEND;
  λ([X = [X1 . X2], Y];
   cond[null[X]:'Y;
    '1:cons('X1, APPEND[X2,Y])])).

append and $append_2$ are extensionally equal.

The notations explained so far (in Sections 1.6, 1.7 and 2.8) comprise what we call the *reference language* of HYPERLISP. The *semantics* of the reference language is given by the rules how a grammatically correct program is translated to a sexp. The syntax and semantics of the reference language may be defined in a constructive manner as we are defining *eval,* but here we will not go into details. (See [8].)

2.9. *Function definition*

We can give a function definition to an arbitrary atom. E.g. let us give to append, which is a literal, i.e. an atom, the following definition:

λ([X = [X1 . X2], Y];

cond[null[X] : 'Y;

'1 : cons('X1, append[X2, Y])]).

We expect

append[(a, b, c), (d, e)] $\vdash$ (a, b, c, d, e).

This will be realized by the rule:

($a\Delta$) $f \in \mathbf{A}$, $f \neq 1$, $f \neq$ eq, $f \neq$ cond,

f is defined to be d, $apply(d, x, z)$
$\rightarrow apply(f, x, z)$.

A function definition is written as follows:

Δappend[X = [X1 . X2], Y]

= cond[null[X] : 'Y;
'1 : cons('X1, append[X2, Y])];

Because of ($a\Delta$), we seldom need ($a\Lambda$).

2.10. *Apply II*

We update the definition of *apply*.

($a1$)–($a5$): as in Section 2.3,

($a\lambda$): as in Section 2.6,

($a\Lambda$): as in Section 2.7,

($a\Delta$): as in Section 2.9,

($ac\mathbf{A}$) $f \in \mathbf{A}$, $f \neq 1$, $f \neq$ eq, $f \neq$ cond, *f is not defined*,
$f \vdash g$, $apply(g, x, z)$
$\rightarrow aply(f, x, z)$,

($ac\mathbf{M}$) $f \in \mathbf{M}$, $car(f) \neq$ lambda, $car(f) \neq$ label,
$f \vdash g$, $apply(g, x, z)$
$\rightarrow apply(f, x, z)$.

Why we omitted ($a6$)–($a13$) in Section 2.3 will be clear in Section 3.2. (ac**A**) and (ac**M**) are for *computed functions*. (See Section 3.6.)

Because of ($a\Delta$), *eval* depends on a set of function definitions. We write $x \vdash_D z$ to mean that x is evaluated to z under the set of definitions D.

We could have defined D formally as a sexp like an association list of LISP1.5 (see [5]). But it would have made the definitions messy.

3. Characteristic Features and Examples

3.1. *Quasi-quotation*

Let

$$xcons_2 = \lambda([X, Y]; '(Y . X)).$$

Since $'(y . x) \vdash_\phi (y . x)$, $xcons_2$ is extensionally equal to *xcons*; i.e.

$$xcons_2[x, y] \vdash_\phi (y . x].$$

We do not need the constructor cons here.

In lambda expressions, we can write (X . Y), [X . Y], (X) etc. as above. This is called Quine's *quasi-quotation*. (See [7].)

Because of the quasi-quotation and the metaterm, we can dispense with explicit use of constructors and selectors in many cases. Take naive reverse as an example. In LISP1.6 (see [6]), we write

```
(DE REVERSE (X)
 (COND ((NULL X) NIL)
       (T (APPEND (REVERSE (CDR X))
          (CONS (CAR X) NIL))))).
```

In HYPERLISP,

```
Δreverse[X = [X1 . X2]]
= cond[null[X]:0;
       '1:append(reverse[X2], '(X1))];
```

3.2. *Definability of primitives*

```
Δatom[X] = cond['X:'1];
```

Δnull[X] = eq[X, 0];

Δcar[[X1 . X2]] = 'X1;

Δcdr[[X1 . X2]] = 'X2;

Δcons[X, Y] = '(X . Y);

Δsnoc[X, Y] = '[X . Y];

3.3. *Definability of eval and apply*

Let D consist of:

Δeval[X] = X;

Δapply[F, X] = [F . X];

Then for any $x, f, z \in \mathbf{S}$,

$\text{eval}[x] \vdash_{\overline{D}} z$ iff $x \vdash_{\overline{D}} z$,

$\text{apply}[f, x] \vdash_{\overline{D}} z$ iff *apply*(f, x, z) under D.

3.4. *Special form*

Since the caller determines whether to evaluate the arguments or not, we do not have to distinguish between *expr* and *fexpr*. Let us define or; let D consist of:

Δor[. X = [X1 . X2]]

= cond[eq[X, 0] : 0;

X1 : '1;

'1 : [or . X2]];

where [. X = [X1 . X2]] means that X is the whole argument list, X1 the first argument and X2 the argument list but the first.

$\text{or}[\text{eq}[a, b], \text{eq}[b, b]] \vdash_{\overline{D}} 1$,

$\text{or}[\text{eq}[a, b], \text{eq}[b, c]] \vdash_{\overline{D}} 0$.

Another example, list:

Δlist[. X] = 'X;

3.5. *Funarg*

Since the arguments are actually substituted in the function body, we have no *funarg problem.*

For an example of a functional argument, see Section 3.6.

3.6. *Paradoxical combinator*

Curry's $Y = \lambda h.\ (\lambda x.\ h(xx))(\lambda x.\ h(xx))$ can be easily simulated. (See [2].) Let D consist of:

$$\Delta y[H] = \lambda([X];\ H[X[X]])[\lambda([X];\ H[X[X]])];$$

$$\Delta null[X] = eq[X,\ 0];\ \Delta cons[X,\ Y] = '(X\ .\ Y);$$

Then:

$$y[\lambda([APPEND];$$

$$'\lambda([X = [X1\ .\ X2],\ Y];$$

$$cond[null[X] : 'Y;$$

$$'1 : cons('X1,\ APPEND[X2, Y])])][(a,\ b,\ c),\ (d,\ e)]$$

$$\vdash_D (a,\ b,\ c,\ d,\ e).$$

4. Bootstrap

By Section 3.3 we know that eval and apply are easily defined in HYPERLISP. Here we try a more instructive set of definitions. Let D consist of the following definitions:

$$\Delta eval[E = [F\ .\ X]]$$

$$= cond[atom[E] : apply[F,\ X];$$

$$'1 : apply('F,\ evlis[X])]$$

$$\Delta evlis[X = [X1\ .\ X2]]$$

$$= cond[null[X] : 0;$$

$$'1 : cons(eval[X1],\ evlis[X2])]$$

$$\Delta apply[F = [L,\ P,\ B],\ X = [X1,\ X2]]$$

```
=cond[null[F]:0;
      atom[F]:
        cond[eq[F,1]:'X1;
             eq[F, eq]:eq[X1, X2];
             eq[F, cond]:evcon[X];
             '1:apply(eval[F], 'X)];
      eq[L, lambda]:eval(subst[X, P, B]);
      eq[L, label]:apply(subst[F, P, B], 'X);
      '1:apply(eval[F], 'X)]

⊿evcon[X=[[E1 . E2] . X2]]
=cond[null[X]:0;
      atom(eval[E1]):eval[E2];
      '1:evcon[X2]]

⊿subst[X, P=[P1 . P2], B=[B1 . B2]]
=cond[null[P]:'B;
      atom[P]:point[X, P1];
      atom[B]:snoc(subst[X, P1, B1], subst[X, P2, B2]);
      '1:cons(subst[X, P1, B1], subst[X, P2, B2])]

⊿point[X=[X1 . X2], Q=[Q1 . Q2]]
=cond[null[Q]:0;
      atom[Q]:'X;
      null[Q2]:point[X1, Q1];
      '1:point[X2, Q2]]

⊿atom[X]=cond['X:'1];
⊿null[X]=eq[X, 0];
⊿cons[X, Y]='(X . Y);
⊿snoc[X, Y]='[X . Y];
```

Theorem. *For any* $x, z \in \mathbf{S}$

$$\mathrm{eval}[x] \vdash_{\mathrm{D}} z \textit{ iff } x \vdash_{\phi} z.$$

Proof. By induction on the evaluation of x. Refer to [8] for the details.

Compare our constructive approach with that of Gordon [3], where he proves the correctness of the universal functions of Pure LISP by means of denotational semantics.

5. Implementation

A tiny interpreter is implemented on PDP11 and VAX11 under UNIX. The technique in [4] is used to implement **S**. (See also [1, p. 402].) Function definitions are semi-compiled so that the arguments are not actually substituted in the body.

It took 17.2 seconds to compute 92 solutions of the eight queens puzzle on VAX11/780.

References

[1] J. Allen, Anatomy of LISP (McGraw-Hill, New York, 1978).
[2] H.B. Curry and R. Feys, Combinatory Logic, Vol. 1 (North-Holland, Amsterdam, 1968).
[3] M. Gordon, Models of Pure LISP, Department of Machine Intelligence, Experimental programming reports: No. 30, University of Edinburgh (1973).
[4] E. Goto, Monocopy and associative algorithms in an extended LISP, TR74-03, Information Science Laboratories, Faculty of Science, University of Tokyo (1974).
[5] J. McCarthy, et al., LISP 1.5 Programmer's Manual (The M.I.T. Press, Cambridge, MA, 1962).
[6] L.H. Quam and W. Diffie, Stanford LISP 1.6 Manual, Stanford Artificial Intelligence Laboratory Operating Note. No. 28.4 (1968).
[7] W. Quine, Mathematical Logic (Harvard, 1955).
[8] M. Sato, Theory of symbolic expressions, TR80-16, Department of Information Science, Faculty of Science, University of Tokyo (1980).

Algorithmic Languages, de Bakker/van Vliet (eds.)

Symbolic Evaluation of LISP Functions with Side Effects for Verification

Dennis de Champeaux and Jos de Bruin

Faculty of Economics, Informatica Department, Jodenbreestraat 23, 1011 NH Amsterdam, The Netherlands

In this paper we present a symbolic evaluator of LISP functions. It can handle data-altering functions of the RPLACA type, i.e. functions that change one data-structure by replacing parts of it by other structures that will themselves not be changed further, at least not permanently. The state description languages uses first-order predicate calculus. It is argued that symbolic evaluation in terms of this language, although theoretically adequate, is not feasible in general, since it may require extremely complicated specifications for real-life functions with side effects. Examples are given of the specifications needed to verify several versions of SUBSTAD, a non-copying SUBST.

1. Introduction

In 1978 we published SUBSTAD, a non-copying version of SUBST (see [1]). Comparison of these two functions in the context of a unification algorithm showed some very favorable results. Two years later we found out that the results were biased by a bug in our machine implementation of SUBST.

This experience increased our interest in verification, in particular of functions with side effects, such as SUBSTAD. These functions pose a challenge to verifiers. One simple RPLACA can have consequences for every data-structure around.

Very few practical, ready-to-use techniques are available at present. The theoreticians of program verification (for an overview, see [5]) are developing languages (Dynamic Logic e.g.) that abstract away from real appli-

cation, concern toy-like programming languages and tend to be considered as interesting objects by themselves.

More promising seem concrete efforts like that of Topor [8], who verified the correctness of the Schorr–Waite marking algorithm, an algorithm somewhat similar to SUBSTAD. His proof by hand is reasonable to follow, but we are interested in actually automating the verification process as much as possible.

We developed a program that can keep track of the many details involved when checking all possible branches of computation trees. We have chosen the method of symbolic evaluation [3, 6], because it guarantees that every branch is visited and that all preconditions to operations are considered.

Symbolic evaluation requires the addition of input/output specifications to the program code and of invariants to each loop in that code. The code is evaluated with symbolic input values that conform to the input specification, producing a symbolic output value for each branch through the code. The symbolic evaluator should embody the semantics of the operators used in the code, in our case (at least) the subset of LISP primitives used in SUBSTAD. For each of those operators it should be able to transform the description of the state in which this operator is called into a description of the state it creates.

It has to be verified that all of the output values produced are in accordance with the output condition. This, as well as checking entry and loop conditions, can be done 'manually' or by a theorem prover. Although we have been experimenting with COGITO, our theorem prover (for results see [2]), our concern here is the automatic updating concerning functions with side effects, like RPLACA. For details on the actual proofs (by hand) see [2].

2. The State Description Language

In order to facilitate deduction, the state description language uses first-order predicate calculus. We start off with a countable domain of cells C and a countable domain of atoms A, where C and A are disjoint. Let D be their union: $D = C \cup A$. We will have the partial functions:

- car and cdr, with domain C and range D; and
- addr, with domain D and range N, the natural numbers.

We will have the partial predicate:
- atom, with domain D, and which, where defined, coincides with the characteristic predicate of A.

Using the addr-function, we define the relation eqa with:

$$(d)(e)\{\text{eqa}(d,e)\leftrightarrow\text{addr}(d)=\text{addr}(e)\},$$

for d, e in D where addr is defined.

Axiom 1.

$$(d)(e)\{\text{eqa}(d,e)\rightarrow[\text{atom}(d)\rightarrow d=e]\} \quad \text{for } d, e \text{ in } D.$$

Axiom 2.

$$(d)(e)\{[\sim\text{atom}(d)\ \&\ \text{eqa}(d,e)\ \&\ \text{car}(d)=\text{car}(e)\ \&\ \text{cdr}(d)=\text{cdr}(e)]\rightarrow d=e\}$$

for d, e in D.

Axiom 1 ensures that e is also non-atomic.

We define a *data object* D_r to be an element of the power set of D:
(1) with D_r of finite size,
(2) with C_r and A_r the elements of D_r respectively in C and A,
(3) with car(C_r) and cdr(C_r) subsets of D_r, and
(4) with a unique element r in D_r, the root of D_r, which has the property that all other members of D_r can be reached from r by finite car/cdr chains.

From now on we mention data objects by referring to their roots.

Recursive definitions on data objects run the risk of being undefined due to infinite regress, since data objects may contain cycles – a cell can reach itself along a car/cdr chain. The finiteness of data objects is the way out of this problem. Most recursive definitions that we will give in the sequel apply to data objects that have the special format of a tree. For generalizations to arbitrary data objects, see [2].

Recursive definitions on trees invoke in proofs an appeal to the so-called car/cdr induction. Whenever a formula $P(x)$ reduces to a formula $P(\text{car}(x))$ and/or $P(\text{cdr}(x))$ then car/cdr induction allows the conclusion that $P(x)$ has been inferred. This is justified by the observation that a well-founded relation can be constructed (in most cases the number of cells

reachable from x) that decreases on each recursive reference. Handled carefully, this also applies to recursive definitions with non-tree arguments.

Next we give definitions of the predicates partof and loopfree. The definition of partof works only on trees (+ is the sequentially read disjunction connective):

$$
\begin{aligned}
&(d)(e)\{\text{partof}(d,e)\leftrightarrow\\
&\qquad[\text{partofcar}(d,e)+\text{partofcdr}(d,e)]\},\\
&(d)(e)\{\text{partofcar}(d,e)\leftrightarrow\\
&\qquad[\sim\text{atom}(e)\ \&\\
&\qquad\ (d=\text{car}(e)+\text{partof}(d,\text{car}(e)))]\},\\
&(d)(e)\{\text{partofcdr}(d,e)\leftrightarrow\\
&\qquad[\sim\text{atom}(e)\ \&\\
&\qquad\ (d=\text{cdr}(e)+\text{partof}(d,\text{cdr}(e)))]\},\\
&(d)\{\text{loopfree}(d)\leftrightarrow\text{loopfree1}(d,0)\},\\
&(d)(V)\{\text{loopfree1}(d,V)\leftrightarrow\\
&\qquad[\text{atom}(d)+\\
&\qquad\ \{\sim(d\ \text{in}\ V)\ \&\\
&\qquad\quad\text{loopfree1}(\text{car}(d),\{d\}\cup V)\ \&\\
&\qquad\quad\text{loopfree1}(\text{cdr}(d),\{d\}\cup V)\}]\}.
\end{aligned}
$$

The expression partof(d,e) signifies that the data object e contains a cell or atom identical to the root of d. Loopfree defines the property that a data object does not contain a cycle.

A *state description* is a conjunction of facts referring to a finite number of data objects, always containing the data objects nil and t, corresponding with NIL and T, members of A, for which holds: atom(nil), atom(t) and $\sim(t=\text{nil})$.

A state description may refer to 'virtual' data objects, which existed during earlier states. Two data objects are compatible, if they can co-exist:

$$
\begin{aligned}
&(d)(e)\{\text{compatible}(d,e)\leftrightarrow\\
&\qquad[\text{atom}(d)+\text{atom}(e)+\\
&\qquad\ (\text{eqa}(d,e)\ \&\ d=e)+\\
&\qquad\ (\sim\text{eqa}(d,e)\ \&\\
&\qquad\quad\text{compatible}(d,\text{car}(e))\ \&\ \text{compatible}(d,\text{cdr}(e))\ \&\\
&\qquad\quad\text{compatible}(\text{car}(d),e)\ \&\ \text{compatible}(\text{cdr}(d),e))]\}.
\end{aligned}
$$

When two data objects are non-compatible at least one has to be virtual. The RPLACX operations are responsible for making data objects virtual.

Definition. An *alist* is a finite list of pairs $((a_1, r_1), \ldots, (a_n, r_n))$ with a_i atoms unequal nil and r_i the roots of data objects, while for each pair r_i, r_j we have: compatible(r_i, r_j).

The alist contains the current bindings of the atoms. A data object is virtual with respect to an alist if it is non-compatible with an r_i from that alist. An atom may occur more than once as a first element of a pair, for instance as a consequence of recursion. LISP functions retrieve and update leftmost occurrences. Side effects may propagate to the right in the alist. Extensions and contractions, as a consequence of entering a higher or lower stack level, also occur at the left.

Definition. A *state configuration* is a pair (AL, FL) with AL an alist and FL (the factlist) a state description. Atomicity of nil, t and all atoms a_i on the alist is implicitly assumed.

3. The Symbolic Evaluator

When given LISP-code and a state configuration the symbolic evaluator generates a tree of state configurations, corresponding to all possible computation paths through the code. The symbolic evaluator works like a real LISP evaluator. It has a code pointer, corresponding to a program counter, to that part of the code which has to be executed, it contains modules which correspond to built-in LISP functions and it knows what to do with user defined functions.

A non-numerical atomic form is evaluated by retrieving the most recent (i.e. leftmost) binding from the current alist.

For built-in functions, the recipe consists of checking whether pre-conditions, parametrized for the current arguments, are fulfilled and, if the check succeeds, updating the state configuration. An exception is made for COND. The COND-module generates one or more bifurcations of the current state configuration. The correctness of a bifurcation (satisfiability of a test expression and its negation) is not proven by means of the deduction machinery but by constructing or having available two models

that possess opposite truth values with respect to the test expression but are both consistent with the current state configuration. To construct these models one could ask the user to provide several examples, which are processed concurrently with the symbolic input specification for the code (not implemented). Testing by running examples and formal verification should not be seen as mutually exclusive, but should go hand in hand.

Modules are implemented for the following subset of standard LISP functions: ATOM, CAR, CDR, COND, CONS, EQ, EQUAL, GO, NOT, NULL, PROG, PROGN, QUOTE, RETURN, RPLACA, RPLACD and SETQ. The functions COND, GO, PROG, PROGN, QUOTE and SETQ are of type FSUBR, i.e. evaluation of their arguments is to their own discretion. The other functions have automatic – left to right – argument evaluation before module-specific actions are taken.

An essential requirement for the modules is that the compatibility property of state configurations is preserved. Our only worry is RPLACA, RPLACD and SETQ because only those functions affect the alist. We will describe some of the modules.

ATOM. Let the argument of ATOM evaluate to x. A new symbolic value will be generated, say g1, which will be returned as the value, while the fact list will be expanded with:

$$\{\text{g1} = t \;\&\; \text{atom}(x)\} + \{\text{g1} = \text{nil} \;\&\; \sim\text{atom}(x)\}.$$

The implemented version deals immediately with the atomicity of x. It returns t or nil when atomicity or non-atomicity of x can easily be derived from the given fact list, otherwise the user is asked to indicate whether t, nil or both possibilities are to be pursued. In this last case, it generates a bifurcation of the current computation branch with t in one and nil in the other branch, adding either atom(x) or ~atom(x) to the respective factlist.

CAT (and analogously CDR). Let the argument of CAR evaluate to x. In contrast with ATOM there is a precondition check for CAR: ~atom(x) should be derivable from the current fact list. If that derivation succeeds a new symbolic value, say g2, is generated and returned and g2 = car(x) is added to the fact list.

COND. This function leads to bifurcation(s) of the current computation branch, as described for the implemented version of ATOM.

CONS. Let the arguments of CONS evaluate to x and y. A new symbolic value, say g3, is generated and will be returned, while the fact list will be extended with: ~atom(g3), car(g3) = c and cdr(g3) = y.

GO. We assume only backward jumps. The loop invariant associated with the label to which GO refers, provided by the user and parametrized for the current bindings by the evaluator, should be derivable from the current fact list. A non-looping check, based on a well founded relation should also be performed. Because jumps are always backwards, we do not have to consider the current computation branch any further.

RPLACA (and analogously RPLACD). Let the arguments of RPLACA evaluate to x and y. The precondition for RPLACA is ~atom(x). A new symbolic value, say g6, is generated and returned, while the fact list is extended with: eqa(x, g6), car(g6) = y and cdr(g6) = cdr(x).

Any non-atomic binding z1 on the alist, identical to x or 'above' x, will be affected indirectly by the RPLACA operation and has to be replaced by a new binding z2 for which minimally holds: eqa(z1, z2). In general: when a RPLACX operation causes x1 to be replaced by x2 then each binding on the alist, y1, will be replaced by a fresh binding, y2, while the fact list will grow with: eqaupto(y1, y2, x1, x2), which says: y2 is identical with y1 unless there is a substructure of y1 that is identical with x1. The predicate eqaupto is defined as:

$$
\begin{aligned}
&(y1)(y2)(x1)(x2)\{\text{eqaupto}(y1, y2, x1, x2) \leftrightarrow \\
&[\text{eqa}(y1, y2)\ \&\ \\
&\ \{y1 = x1 \rightarrow y2 = x2\}\ \&\ \\
&\ \{[\sim(y1 = x1)\ \&\ \sim\text{atom}(y1)] \rightarrow \\
&\quad [\text{eqaupto}(\text{car}(y1), \text{car}(y2), x1, x2)\ \&\ \\
&\quad\ \text{eqaupto}(\text{cdr}(y1), \text{cdr}(y2), x1, x2)]\}]\}.
\end{aligned}
$$

Remark. When the original binding y1 is atomic then according to Axiom 1 the new binding y2 is identical with y1.

Lemma 1.

$$\{x1 = x2\ \&\ \text{eqaupto}(y1, y2, x1, x2)\} \rightarrow y1 = y2.$$

Lemma 2.

$$\{\sim(x1 = y1)\ \&\ \sim\text{partof}(x1, y1)\ \&\ \text{eqaupto}(y1, y2, x1, x2)\} \rightarrow y1 = y2.$$

These lemmas can be used to curb updating activities. For proofs of these and other lemma's and theorems, see [2].

Theorem 1. *Let* y1 *and* z1 *be old bindings which are respectively replaced by* y2 *and* z2 *due to an RPLACX-operation that caused* x1 *to be changed into* x2, *thus with* eqa(x1, x2), *then* compatible(y1, z1), eqaupto(y1, y2, x1, x2) *and* eqaupto(z1, z2, x1, x2) *implies* compatible(y2, z2).

SETQ. Let the second argument evaluate to x. The precondition for SETQ is that the non-evaluated first argument is atomic, say A. The binding of the leftmost occurrence of A on the alist will be replaced by x. If A does not occur on the alist – i.e. when A is a global variable – then $(A \,.\, x)$ will be added at the righthand side of the alist. Preservation of alist-compatability is ensured when the evaluation of the second argument yields a value compatible with the current bindings.

The modules not described trigger obvious updatings. (For the equal predicate needed by the EQUAL module, see [2].)

3.1. *User functions*

Most LISP functions to be verified will contain functions other than the above mentioned primitive ones. These are provided either by the user or are built-in. They can be handled by the evaluator if they are accompanied by an input and an output condition.

The symbolic evaluator first asks for (and tries to assist with) a check that the input condition is fulfilled and then looks whether the user wishes this function to be verified. If so, she will have to provide its body. Recursive user functions will be opened at most once, for obvious reasons. A well-founded relation, user provided, should be used when verifying that arguments of a recursive call score strictly less with respect to that well-founded relation than the arguments at the top level call. This was not implemented.

An output condition should describe the resulting state in terms of the values used in the input condition to enable the symbolic evaluator to update the state configuration in which the function was called. This updating is straightforward when the function does not have side effects and just returns a value, but built-in and user functions of RPLACX-type

need even more complicated alist updating schemes than the one given above for RPLACX.

Suppose we execute (NCONC LIS S1), where the bindings of LIS and S1 are respectively lis and s1. The rightmost leaf of LIS, which must be NIL, will be replaced by a pointer to its second argument S1. Any data-structure containing a pointer to lis or to a cell lying on its 'spine' (i.e. the cdr chain starting at lis) will be changed as a consequence of this NCONC operation.

We will describe an alist update scheme for a class of side effect generating functions, including NCONC, EFFACE and our SUBSTAD support functions SUBSTAD1 and SUBSTAD2. It applies to those functions which cause replacement of a cell, say x1, by a cell, say x2, (thus we have eqa(x1, x2)).

Every binding, z1, on the alist is replaced by a fresh binding, z2, and the fact list is expanded with: transf(z1, z2, x1, x2). The predicate transf and its supporting predicate tr1 and tr2 works by double recursion. First, it is checked whether z1 is identical with x1 or – using tr1 – with a cell reachable from x1. If the tr1-case applies the predicate tr2 is invoked to relate z1 and z2. Second, when z1 is not identical with x1 or a subcell of x1 then transf is called recursively to test whether subcells of z1 are affected by the x1-x2 replacement.

The predicate transf is defined as:

$$
\begin{aligned}
&(y1)(y2)(x1)(x2)\{transf(y1, y2, x1, x2) \leftrightarrow \\
&[eqa(y1, y2)\ \& \\
&\ \{x1 = y1 \rightarrow y2 = x2\}\ \& \\
&\ \{[\sim atom(y1)\ \&\ \sim(x1 = y1)\ \&\ tr1(y1, x1, x2)] \rightarrow \\
&\quad tr2(y1, y2, x1, x2)\}\ \& \\
&\ \{[\sim atom(y1)\ \&\ \sim(x1 = y1)\ \&\ \sim tr1(y1, x1, x2)] \rightarrow \\
&\quad [transf(car(y1), car(y2), x1, x2)\ \& \\
&\qquad transf(cdr(y1), cdr(y2), x1, x2)]\}]\},
\end{aligned}
$$

with tr1 defined as:

$$
\begin{aligned}
&(y1)(x1)(x2)\{tr1(y1, x1, x2) \leftrightarrow \\
&[\sim atom(x1)\ \& \\
&\ eqa(x1, x2)\ \& \\
&\ \{y1 = x1 + \\
&\quad tr1(y1, car(x1), car(x2)) + \\
&\quad tr1(y1, cdr(x1), cdr(x2))\}]\},
\end{aligned}
$$

and with tr2 defined as:

$$
\begin{aligned}
&(y1)(y2)(x1)(x2)\{tr2(y1, y2, x1, x2) \leftrightarrow \\
&[\{y1 = x1 \rightarrow y2 = x2\} \ \& \\
&\ \{\sim(y1 = x1) \rightarrow \\
&\ \ [\{tr1(y1, car(x1), car(x2)) \rightarrow \\
&\ \ \ tr2(y1, y2, car(x1), car(x2))\} \ \& \\
&\ \ \ \{tr1(y1, cdr(x1), cdr(x2)) \rightarrow \\
&\ \ \ \ tr2(y1, y2, cdr(x1), cdr(x2))\}]\}].
\end{aligned}
$$

The meaning of the transf(z1, z2, x1, x2) formula can be phrased as: let y1 be z1 or a subcell of z1, let u1 be x1 or a subcell of x1, while u1 has been replaced by u2 (so u2 is identical with x2 or with a subcell of x2), then, when y1 is identical with u1, there is a corresponding cell in z2, which is identical with u2.

In analogy with Lemma 1 and Lemma 2, we have:

Lemma 3.

$$\{x1 = x2 \ \& \ transf(y1, y2, x1, x2)\} \rightarrow y1 = y2.$$

Lemma 4.

$$
\begin{aligned}
&[(z)\{[z = x1 + partof(z, x1)] \rightarrow \\
&\qquad [\sim(z = y1) \ \& \ \sim partof(z, y1)]\} \ \& \\
&\ transf(y1, y2, x1, x2)] \rightarrow \\
&y1 = y2.
\end{aligned}
$$

Theorem 2. *Let* y1 *and* z1 *be old bindings which are respectively replaced by* y2 *and* z2 *due to a side-effect operation causing* x1 *to be changed into* x2, *thus with* eqa(x1, x2), *then* compatible(y1, z1), transf(y1, y2, x1, x2) *and* transf(z1, z2, x1, x2) *implies* compatible (y2, z2).

The limitations of this updating scheme can be seen from the function NCONC2, defined as:

```
(NCONC2(LAMBDA(LIS1 LIS2 S1)
        (NCONC LIS1(NCONC LIS2 S1)))).
```

A binding referring to the 'spine' of the input binding of LIS2 cannot be recognized and therefore will not be updated, although it is not up-to-date anymore.

We conclude that the user must be given the option to specify a specific, idiosyncratic alist update mechanism for any function having side effects. This will considerably increase the verification burden, since it will have to be shown that the compatibility requirement for the updated alist is fulfilled.

4. Evaluating SUBSTAD

To give an impression of the feasibility of the method of symbolic evaluation as introduced above, we will discuss our effort to verify SUBSTAD. This function is called with three arguments: S1, LAT and S3. It replaces all occurrences of LAT in S3 by S1. The value of LAT should be a non-numeric atom. This is checked by SUBSTAD, which also handles the case that S3 is atomic. Otherwise it calls a support function with one argument, S3.

The support function published in [1] uses pointer reversal to avoid the use of a stack, as is done in garbage collectors. Before discussing this function, we will make some remarks on the verification of two simpler versions, to show how the method works and to illustrate how a slight modification in a program can lead to substantial differences in its verification.

4.1. *SUBSTAD1*

First of all, the recursive SUBSTAD1:

```
(SUBSTAD1(LAMBDA(S3)(PROG2
      (COND((ATOM(CAR S3))
            (COND((EQ LAT(CAR S3))(RPLACA S3 S1))))
           (T(SUBSTAD1(CAR S3))))
      (COND((ATOM(CDR S3))
            (COND((EQ LAT(CDR S3))(RPLACD S3 S1))))
           (T(SUBSTAD1(CDR S3))))
))).
```

The preconditions are:
- the binding of S3, say vs3, is not atomic;
- the binding of LAT, say lat, is atomic; and

– lat is not a leaf of the binding of S1, say vs1. This last precondition is meant to prevent the introduction of cycles.

To simplify the proofs, we will assume that vs1 does not share substructure with vs3. Consequently, Lemma 4 will apply and therefore updating of the S1 binding will never happen. (When vs1 does share structure we can still invoke Lemma 2, since lat is not a leaf of vs1.)

Since we assume the preconditions to hold, the fact list will (implicitly) contain:

atom(lat) & ~atom(vs3) & ~partof(lat, vs1).

The input alist is:

((S1 . vs1)(LAT . lat)(S3 . vs3)).

Assume the output alist to be:

((S1 . vs1)(LAT . lat)(S3 . nvs3)).

The output assertion to be verified will be:

replacedd(vs1, lat, vs3, nvs3),

with replacedd (replacement with potential destruction of vs3) defined as:

(x1)(x2)(x3)(ot){replacedd(x1, x2, x3, ot)↔
[eqa(x3, ot) &
{atom(car(x3))→
[(x2 = car(x3)→car(ot) = x1) &
(~(x2 = car(x3))→car(ot) = car(x3))]} &
{~atom(car(x3))→replacedd(x1, x2, car(x3), car(ot))} &
{atom(cdr(x3))→
[(x2 = cdr(x3)→cdr(ot) = x1) &
(~(x2 = cdr(x3))→cdr(ot) = cdr(x3))]} &
{~atom(cdr(x3))→replacedd(x1, x2, cdr(x3), cdr(ot))}]}.

There are nine different paths through the code. We will work our way along one of the paths.

Initially the fact list contains:

atom(lat) & ~atom(vs3) & ~partof(lat, vs1).

Assuming that (ATOM(CAR S3)) yields T we get in addition:

xa = car(vs3) & atom(xa).

Assuming that (EQ LAT(CAR S3)) yields T we get:

lat = xa.

RPLACA generates a new value, say nv1, adding:

eqa(nv1, vs3) & car(nv1) = vs1 & cdr(nv1) = cdr(vs3).

The alist update scheme for RPLACA generates a new binding for S3, say ivs3, so the alist becomes:

((S1 . vs1)(LAT . lat)(S3 . ivs3)),

while the fact list grows with:

eqaupto(vs3, ivs3, vs3, nv1).

Assuming that (ATOM(CDR S3)) yields NIL we get:

xd = cdr(ivs3) & ~atom(xd).

The next action concerns the recursive call on the CDR. Its parametrized and simplified input condition:

~atom(xd) & atom(lat) & ~partof(lat, vs1),

is trivially satisfied. The function will not be opened, but instead the fact list grows with:

replacedd(vs1, lat, xd, nxd) & transf(ivs3, jvs3, xd, nxd),

while the alist changes into:

((S1 . vs1)(LAT . lat)(S3 . jvs3)).

The output assertion to be proven for this particular path is:

replacedd(vs1, lat, vs3, jvs3).

We will not give proofs. The general strategy in this and following cases is a combination of subproblem recognition, case reasoning, expansion of recursive definitions and application of car/cdr induction.

4.2. *SUBSTAD2*

The treatment of SUBSTAD1 is given above was slightly incorrect, although this did not affect the result. Upon entry of SUBSTAD1 the alist is in fact:

((S3 . vs3)(S1 . vs1)(LAT . lat)(S3 . vs3)),

where the first occurrence of S3 comes from SUBSTAD1 and the second one from SUBSTAD. The output assertion of SUBSTAD1 did refer to the *second* occurrence of vs3. This more subtle treatment of the alist is essential for the half recursive half iterative support function SUBSTAD2.

```
(SUBSTAD2(LAMBDA(S3)(PROG(HH)
AGAIN
        (COND((ATOM(SETQ HH(CAR S3)))
                (COND((EQ LAT HH)(RPLACA S3 S1))))
               (T(SUBSTAD2 HH)))
        (COND((ATOM(SETQ HH(CDR S3)))
                (COND((EQ LAT HH)(RPLACD S3 S1))))
               (T(SETQ S3 HH)
                 (GO AGAIN)))
)))
```

Because of the assignment of the local S3 to its CDR just before the loop, we no longer have a handle on the data-structure as a whole, to which we must be able to refer in order to specify the loop invariant and to enable a correct update of the calling environment after existing SUBSTAD2. The problem is solved by refering to the *global* S3, the argument with which SUBSTAD2 is called. (In general a pre-processor should take care that all arguments given to user defined functions are explicitly assigned on the alist.)

Verifying SUBSTAD2 requires deducing the loop invariant when control reaches the label AGAIN upon entering the function, deducing the output assertion for six paths through the code and deducing the loop invariant for three paths.

The input alist is as given above. The output alist, after exiting from SUBSTAD2 will be:

((S1 . vs1)(LAT . lat)(S3 . nvs3)).

The input and output assertion are the same as for SUBSTAD1. We have to provide a loop invariant with the label AGAIN. This loop assertion will refer to the current bindings of the variables, so we also have to specify an alist at the label:

((HH . vhh)(S3 . ls3)(S1 . vs1)(LAT . lat)(S3 . gs3)).

The value ls3 is the local value of S3, and gs3 is the global value of S3. The

loop assertion will be:

$$\text{atom(lat)} \;\&\; \sim\text{atom(ls3)} \;\&\; \sim\text{atom(vs3)} \;\&\; \sim\text{partof(lat, vs1)} \;\&\; \text{spine(vs1, lat, vs3, gs3, ls3)}.$$

We will not give the definitions of spine and other support predicates. Giving a general description of the situation at the label is rather complicated, since it is not enough to say that every tree hanging off the spine above the local S3 has been checked and replaced if necessary. Structure sharing may have led to changes in the part of the tree that is still to be investigated. It may even have caused the replacement of the right most leaf of vs3 by a pointer to vs1, so S3 may eventually point to a cell for which there is no corresponding cell in the original vs3.

We will just give one definition as an example, for the others we again refer to [2]. The predicate sidefct is used to describe that xp and xq, which are parts of the not yet visited subtrees x3 and x1 of the original (xo) and current (xn) incarnation, are the same unless structure sharing has led to side effects.

$$\begin{aligned}
&(\text{xo})(\text{xn})(\text{x3})(\text{xso})(\text{xsn})(\text{xp})(\text{xq})\\
&\qquad\{\text{sidefct(xo, xn, x3, xso, xsn, xp, xq)}\leftrightarrow\\
&[\text{eqa(xp, xq)} \;\&\\
&\{\text{xso} = \text{x3}\rightarrow\\
&[\{\text{atom(car(xp))}\rightarrow\text{car(xp)} = \text{car(xq)}\} \;\&\\
&\{\sim\text{atom(car(xp))}\rightarrow\\
&\text{sidefct(xo, xn, x3, xo, xn, car(xp), car(xq))}\} \;\&\\
&\{\text{atom(cdr(xp))}\rightarrow\text{cdr(xp)} = \text{cdr(xq)}\} \;\&\\
&\{\sim\text{atom(cdr(xp))}\rightarrow\\
&\text{sidefct(xo, xn, x3, xo, xn, cdr(xp), cdr(xq))}\}]\} \;\&\\
&\{\sim(\text{xso} = \text{x3})\rightarrow\\
&[\{\text{car(xso)} = \text{xp}\rightarrow\text{car(xsn)} = \text{xq}\} \;\&\\
&\{\sim(\text{car(xso)} = \text{xp})\rightarrow\\
&[\{\text{tr1(xp, car(xso), car(xsn))}\rightarrow\\
&\text{tr2(xp, xq, car(xso), car(xsn))}\} \;\&\\
&\{\sim\text{tr1(xp, car(xso), car(xsn))}\rightarrow\\
&\text{sidefct(xo, xn, x3, cdr(xso), cdr(xsn), xp, xq)}\}]\}]\}]\}.
\end{aligned}$$

Symbolic evaluation of SUBSTAD2 generates fact lists that are much longer than those generated for SUBSTAD1, since the alist in this case contains three arguments (HH, local S3 and global S3) that have to be

updated after an RPLACX or a recursive call to SUBSTAD2. This, and the greater amount of predicates needed to specify the loop invariant, made verification of this function just barely feasible. The great difference in verification effort caused by a slight change in the code, challenges the claim that once a program is verified, modifications will require very little additional effort.

4.3. *SUBSTADP*

The disparity between amount of code and amount of ad hoc definitions is even worse for SUBSTADP:

```
(SUBSTADP(LAMBDA(S3)(PROG(EX HH)
      (SETQ EX $)
L2
      (SETQ HH(CAR S3))
      (COND((NOT(ATOM HH))
            (MARK S3 1)
            (RPLACA S3 EX)
            (SETQ EX S3)
            (SETQ S3 HH)
            (GO L2))
           ((EQ LAT HH)(RPLACA S3 S1)))
L4
      (SETQ HH(CDR S3))
      (COND((ATOM HH))
           ((NOT(EQ EX $))
            (REPLACD S3 EX)
            (SETQ EX S3)
            (SETQ S3 HH)
            (GO L2))
           (T(SETQ S3 HH)
             (GO L2)))
      (COND((EQ LAT HH)(RPLACD S3 S1)))
      (COND((EQ EX $)(RETURN)))
L5
      (SETQ HH S3)
      (SETQ S3 EX)
      (COND((MARKB S3)
```

```
            (MARK S3 0)
            (SETQ EX(CAR S3))
            (RPLACA S3 HH)
            (GO L4)))
      (SETQ EX(CDR S3))
      (RPLACD S3 HH)
      (GO L5)
)))?end of the pointer reversal SUBSTADP?
```

In this version, the use of a stack is avoided by reversing pointers, i.e. when the car or cdr part of a cell is non-atomic, this part is saved, while the car or cdr is replaced by a pointer back to the parent cell immediately above it. Marking is used to indicate whether the car or the cdr part of the cell contains the reversed pointer. The tree is searched in a depth first manner.

The code contains three labels, so in addition to the input and output assertion we have to set up three loop invariants. Describing the situation at the various loops is extremely complicated because of the much greater number of (temporary) replacements.

We defined the predicates that are necessary to describe the situation at one label, L2, assuming that vs1 is atomic. Even with this drastic simplification, we needed a staggering amount of definitions: eleven predicates, several of them with seven arguments and totalling nearly 200 lines of text (see [2]). To get an impression of what is involved, glance at the definitions of two predicates, lb2at3 and its support lb2at5. They describe the subtrees hanging off the spine above the inverted pointer chain.

```
(vs1)(lat)(ex)(l3)(ol)(nw)
      {lb2at3(vs1, lat, ex, l3, ol, nw)↔
[eqa(ol, nw) &
 {onichain(ex, nw)→lb2at5(vs1, lat, ex, l3, ol, nw)} &
 {~onichain(ex, nw)→
  [{atom(car(ol))→
    [{car(ol) = lat→car(nw) = vs1} &
     {~(car(ol) = lat)→car(nw) = car(ol)}]} &
   {~atom(car(ol))→
     lb2at3(vs1, lat, ex, l3, car(ol), car(nw))} &
   {atom(cdr(ol))→
    [{cdr(ol) = lat→cdr(nw) = vs1} &
     {~(cdr(ol) = lat)→cdr(nw) = cdr(ol)}]} &
   {~atom(cdr(ol))→
     lb2at3(vs1, lat, ex, l3, cdr(ol), cdr(nw))}]}].
```

This predicate mainly looks whether nw – which is already visited – is residing on the inverted pointer chain, which may be caused by structure sharing. If so the predicate lb2at5 will describe the situation.

```
(vs1)(lat)(ex)(l3)(ol)(nw)
      {lb2at5(vs1, lat, ex, l3, ol, nw)↔
[eqa(ol, nw) &
 {ex = nw→
  [{markb(nw)→
    [replacedd(vs1, lat, car(ol), l3) &
     {atom(cdr(ol))→
      [{cdr(ol) = lat→cdr(nw) = vs1} &
       {~(cdr(ol) = lat)→cdr(nw) = cdr(ol)}]} &
     {~atom(cdr(ol))→
      replacedd(vs1, lat, cdr(ol), cdr(nw))}]} &
   {~markb(nw)→
     {atom(car(ol))→
      [{car(ol) = lat→car(nw) = vs1} &
       {~(car(ol) = lat)→car(nw) = car(ol)}]} &
     {~atom(car(ol))→
      replacedd(vs1, lat, car(ol), car(nw))} &
     replacedd(vs1, lat, cdr(ol), l3)]}]} &
 {~(ex = nw)→
  [{markb(nw)→
    [{atom(cdr(ol))→
      [{cdr(ol) = lat→cdr(nw) = vs1} &
       {~(cdr(ol) = lat)→cdr(nw) = cdr(ol)}]} &
     {~atom(cdr(ol))→
      lb2at3(vs1, lat, ex, l3, cdr(ol), cdr(nw))} &
     (∃ icel){onichain(ex, icel) &
                lb2at5(vs1, lat, ex, l3, car(ol), icel) &
                [markb(icel)→car(icel) = nw] &
                [~markb(icel)→cdr(icel) = nw]}]} &
   {~markb(nw)→
    [{atom(car(ol))→
      [{car(ol) = lat→car(nw) = vs1} &
       {~(car(ol) = lat)→car(nw) = car(ol)}]} &
     {~atom(car(ol))→
```

lb2at3(vs1, lat, ex, l3, car(ol), car(nw))} &
($\exists$ icel){onichain(ex, icel) &
lb2at5(vs1, lat, ex, l3, cdr(ol), icel) &
[markb(icel)→car(icel) = nw] &
[~markb(icel)→cdr(icel) = nw]}]}]}].

When nw = ex, we can describe it with replacedd, keeping in mind whether its car (markb) or its cdr (~markb) contains the back pointer.

If nw lies somewhere else on the inverted pointer chain and the non-reversed pointer points to an atomic structure, describing this part is straightforward. However, if it is non-atomic, we have to recursively invoke lb2at3, because structure sharing between that part of nw and the reversed pointer chain is again possible.

To describe the part originally pointed to by the now reversed pointer, we have to use existential quantification. We do not have a direct pointer to it, but we know were to start (at EX) and we know its unique identification: eqa(icel, car(ol)). This identification is part of lb2at5.

Possible structure sharing similarly complicates the description of subtrees on the inverted pointer chain, under l3 or to the right of the inverted pointer chain.

5. Discussion

Although we were able to write a symbolic evaluator that can handle the functions we were interested in (and no doubt a host of others), it was not possible to give a completely general update algorithm to handle all RPLACX-type functions. We defined one for a common class, in which one data-structure is changed by replacing certain subparts by other data-structures that will not themselves be mutated before the function is exited (at least not permanently). To make the verifier a general one, it should allow the user to specify her own update procedures in other cases. Since compatibility will have to be proven by the user in those cases, this places a rather heavy burden on her.

The algorithm given is extremely careful, replacing all bindings on the alist after every call to an RPLACX-type function. This has its price. Updated bindings need potentially complicated proofs to show their invariance, even though it may be very obvious (to us) that in fact they

could not have been changed at all. Of course one could keep the number of updated bindings down by incorporating the lemma's given above and other specific knowledge into the evaluator, but this would amount to pushing the problem around.

The attempt to give correctness proofs for several versions of SUBSTAD revealed that the method of symbolic evaluation – although theoretically adequate – flounders in some cases on a practical problem: formal description of input/output statements as well as loop invariants leads to a proliferation of ad hoc definitions. We expect this to hold for all currently available verification techniques. If so, verification specialists may be adviced to give more attention to the practical implications of their theories, instead of devoting all their energies to esoteric refinements, or even to the design of logics that become an end in themselves.

The bottle-neck lies in the necessity to specify in state-description terms what a function is supposed to do. Whether a function is recursive or not is not even explicitly expressible in such a specification. Somehow people feel more akin to a definition in procedural terms, such as "the terminals equal to lat will be *replaced* by vs1" and "the tree will be *visited* from left to right". Proving correctness of a function would then 'reduce' to showing that the function *behaves* according to expectations rather than that input/output description pairs conform to a certain relation.

The technique we have developed for describing evolving states using an alist, a fact list and predicates like equaupto and transf that capture specific side effects, may be of interest to other areas of A.I. The alist can be considered a collection of individual concepts, where the bindings are the actual extensions. A new situation differs primarily in that some concepts have different extensions, which are described by fresh facts. Outdated facts do not have to be deleted but merely become invisible since they contain arguments not residing on the alist any longer.

This more procedural approach to the frame problem seems to have advantages over the strictly declarative method given in [7]. There is no need for wieldy axioms to express that when $P(x, \ldots, z, s1)$ holds in situation s1 and some conditions are fulfilled, the fact $P(x, \ldots, z, s2)$ can be inferred in s2. Instead we have a different problem. A fact may seem to be obsolete (since an argument has been removed from the alist) while an analogous fact can be inferred for a newly introduced extension. We have encountered this in Lemmas 1–4, where particular circumstances allow one to equate the old and new binding.

Since updatings and the recognition of identities are object centered and therefore may affect many facts simultaneously, this problem seems less obstructive than the original one, but more thinking and/or experimenting is needed to validate this suggestion.

Although we agree with De Millo et al. [4] that the present verification tools do not lend themselves to practical use, we do not share their conclusion that the whole effort should be abandoned. Verifiers will probably always run into resource limitations, but to assume that they will never be able to use mechanisms similar to those that enable humans to circumvent some of these limitations for certain tasks (without sacrificing preciseness) seems premature.

Finally, it pays to have a second look at one's program from a verification perspective. Writing this paper forced us to reconsider the conditions under which the function SUBSTAD is applicable. The specification that we published 5 years ago turned out to be too liberal!

References

[1] D. de Champeaux, SUBSTAD: For fast substitution in LISP, with an application on unification, Inform. Process. Lett. 7(1) (January 1978) 58–62.

[2] D. de Champeaux, Algorithms in AI, Ph.D. Thesis, Economische Faculteit, Universiteit van Amsterdam (1981).

[3] J.A. Darringer and J.C. King, Applications of symbolic execution to program testing, IBM Report RC 6965 (January 1977).

[4] R.A. De Millo et al., Social processes and proofs of theorems and programs, Comm. ACM 22 (5) (May 1979) 271–280.

[5] D. Harel, Proving the correctness of regular deterministic programs: A unifying survey using dynamic logic, IBM Report 7557 (March 1979).

[6] J.C. King, Symbolic execution and program testing, Comm. ACM, 19 (7) (July 1976) 385–394.

[7] J. McCarthy and P.J. Hayes, Some philosophical problems from the standpoint of artificial intelligence, in: B. Meltzer and D. Michie (Eds.), Machine Intelligence 4 (Elsevier, New York, 1969) 463–502.

[8] R.W. Topor, The correctness of the Schorr–Waite list marking algorithm, Acta Informat. 11 (1979) 211–221.

Algorithmic Languages, de Bakker/van Vliet (eds.)

Invited Address

Aad van Wijngaarden's Contributions to ALGOL 60

Peter Naur

Copenhagen University, Copenhagen, Denmark

The events of 1959–1960 leading to the development of the programming language ALGOL 60, with special attention to the contributions of Aad van Wijngaarden, are outlined. While Van Wijngaarden contributed actively to the shaping of most of the central concepts of the language, in particular block structure and procedures, his main influence appears to have been in less tangible aspects of discussion manner and mental style.

Trying to describe and clarify the events that led to the development of ALGOL 60 is a precarious undertaking. Many people and incidents were involved, and many of the views on ALGOL 60 held then and now are emotionally charged. As shown by the discussion provoked by an earlier report on these events by the present writer [5] the likelihood is that a description that goes into the details of the events will be met with angry protests. It should therefore be made clear that the present attempt to identify the particular contributions of Aad van Wijngaarden to ALGOL 60 has been written in response to a specific invitation from the program committee of the International Symposium on Algorithmic Languages, 1981, Amsterdam. Further, although the account will make extensive use of original documents, in many respects it can only present a personal view of the events.

ALGOL 60 was developed by an effort as truly collective as could be conceived. This means that the contribution of each participant in the effort can at best only be understood in connection with, and in relation to, the efforts of a number of other participants. In many cases the contribution of each individual merges with those of others to such an extent that only the collective result can be identified. In the case of the contributions of Aad van Wijngaarden, observed from Copenhagen, it must be clear from the outset that in many cases a further merging will take place with

the contributions of other active workers in Amsterdam, in particular those of Edsger Dijkstra and J.A. Zonneveld. What may be noted in this context is that Van Wijngaarden, being the senior member of the Amsterdam team around 1959, will have been the one who set the direction of the effort and who set up the working conditions that gave the younger members the opportunity to make the contributions for which they are individually recognized.

Van Wijngaarden's contributions to ALGOL 60 belong to the last phase of the development of the language. The first contact of the Amsterdam team with the language development activity, as far as I know, was Dijkstra's participation in a discussion of implementation problems in Mainz, Federal Republic of Germany, on 21 November 1958. At this time ALGOL 58 had already been described in the Zürich report [3], worked out as a result of the meeting in Zürich on 27 May to 2 June 1958, the European participants being F.L. Bauer, H. Bottenbruch, and K. Samelson, of the Federal Republic of Germany, and H. Rutishauser of Switzerland. Van Wijngaarden's name appears first in the official records as participant, together with Dijkstra, in the meeting in Copenhagen, 26–28 February 1959. Thus, like many other Europeans in the final phase of the development, including myself, Van Wijngaarden entered the activity in response to the deliberate effort of the original language committee, centered around GAMM in the Federal Republic of Germany, to expand the geographical basis for the support of the new language.

The meetings in Mainz in November 1958 and in Copenhagen in February 1959 aimed primarily at the discussion of implementation problems, the language described in the Zürich report being taken, at this stage, as essentially final. This is reflected in the fact that the Algol Bulletin [2] that was set up at the meeting in Copenhagen initially addressed "computing centres, who are all actively engaged on using the ALGOL language for facilitating the programming for their respective computers" in order to "facilitate the continued collaboration of these computing centres in all questions related to the practical use of the ALGOL". In the face of the adoption of the Zürich language as virtually established the Amsterdam team remained sceptical and independent, and in fact did not at first join the collaboration chanelled by the Algol Bulletin.

The status of the language as such was taken up for discussion by several groups, including American participants, at the UNESCO Conference on Data Processing in Paris, June 1959. An ad-hoc sub-committee, members

E.W. Dijkstra (The Netherlands), W. Heise (Denmark), A.J. Perlis (U.S.A.), and K. Samelson (Federal Republic of Germany), proposed a time schedule for the preparations of a first, definitive version of the language, and made some specific language proposals. This action extended the scope of the Algol Bulletin so as to include the official European discussion of language modifications. At this time the Amsterdam team entered the mailing list of the Algol Bulletin.

The time schedule for the discussion set November 1 1959 as the last date for acceptance of proposals for the first, definitive version of the language. This deadline brought forth proposals from many sides, collected in the Algol Bulletin (abbreviated AB) 7, mailed on 3 November 1959. The Amsterdam team, Van Wijngaarden and Dijkstra, contributed a series of suggestions, AB 7.31 to 7.35, covering a wide range of topics. The first group, AB 7.31 to 7.33, was concerned mostly with the meaning of names and the dynamics of declarations. These issues had already been the subject of several contributions to the Algol Bulletin, as viewed from two different sides, the first being the need for arrays of dynamically varying sizes and the second the need for some way to control the accessibility of names of the surrounding progam from inside procedure bodies. The Amsterdam proposals in AB 7 brought fresh insight into the discussion, most strikingly in the proposal for level declarations old, new:

> We suggest that the level declaration
>
> **new** (*I*, *I*, ...)
>
> has the effect that, the named entities have no relationship to identically named entities before in the following text, until the level declaration
>
> **old** (*I*, *I*, ...)
>
> which attributes to the entities named herein the meaning that they had before. These level declarations may be nested and form the only way to introduce a new meaning to a name. In particular in a procedure to be compiled along with the main program all variables that should have no relationship etc. should be declared **new** before they have appeared and declared **old** before the end.
>
> The declarations do not only solve the problem of having

> "old" and "new" variables alongside in a procedure, but are also extremely useful in an ordinary program. It should be noted that after **new**(x) the new x is fully independent of the old x and, therefore, type declarations, if necessary, have to be given anew. On the other hand after **old**(x) the type declarations of the old x are still valid.

In AB 7.34 van Wijngaarden and Dijkstra made a proposal for introducing **dummy** as a type declaration:

> This permits among other things to discern between different dummies and apply other declarations to them. Example:
>
> **dummy integer** (e, d)...
>
> $AT[e, d] = A[d, e]$ ...
>
> defines the transpose of a matrix. In here, and this is the next suggestion, the misleading symbol := in the function declaration is replaced by the non-operational symbol =.

Further, in AB 7.35 they suggested that

> It should be possible to declare entities to be other things than real variables, e.g. complex numbers, vectors, matrices, lists (sets) of quantities. A quantity defined by such a declaration may enjoy well defined properties which make it possible to apply operators like +, −, ×, etc. "in the conventional meaning", i.e. in the meaning that is conventional for such types of quantities.

The next step in the European preparation for ALGOL 60 was the meeting in Paris, 12–14 November 1959, attended by 49 persons from nine countries. The findings of this meeting are collected in the reports of five sub-committees, published in AB 8, issued on 12 December 1959. Van Wijngaarden was a member of sub-committee 1, dealing with the identification of objects, which, essentially, recommended that the Amsterdam proposals on dummy variables and on the form of function declarations be considered carefully by the final conference. Sub-committee 2, including Dijkstra as member, dealt with several questions related to declarations. On the dynamic behaviour of declarations the sub-committee report has the following foretelling remark:

> The principal problem is considered to be the range within which a declaration should be valid. The extreme possibilities are the strict limiting by write-up or alternatively by time succession. A further possibility is that of permitting dynamic declarations only when those two extremes are coincident.

Van Wijngaarden was the chairman of the sub-committee 3 concerning **for** and **if** statements, the other active members being K. Samelson and P. Naur. The report of the sub-committee proposes an explanation of the effect of a **for** statement in terms of more elementary statement forms that evaluates all expressions of the **for** clause once before the first repetition, in other words according to a static view. The existence of this report, written by the members named above, is a telling comment on the curious claim made by K. Samelson in 1978 [5, Appendix 7] that certain features of ALGOL 60, including the dynamic **for** clause, were included as a result of the pressure from a party of 'liberalists' or 'trickologists' having as 'hard core' Naur, Perlis, and Van Wijngaarden, against a party of 'restrictionists' that included Samelson himself. As I have explained elsewhere in more detail [5, Appendix 8] I find no support for the claim that such a 'liberalist' party existed, neither in my memory nor in the recorded facts, and the report of sub-committee 3 indicates perfect willingness on the part of two 'hard core liberalists', Van Wijngaarden and myself, to adopt a static **for** clause. If one looks for the explanation why this view of the **for** clause did not prevail in the final version of ALGOL 60 one will find that at the final conference in Paris the **for** statement became predominantly an American issue into which the European members felt it would be tactically unwise to enter strongly.

As a further result of the meeting in Paris, 12–14 November 1959, the European members of the final Algol committee, F.L. Bauer, P. Naur, H. Rutishauser, K. Samelson, B. Vauquois, A. van Wijngaarden, and M. Woodger, were appointed. This group of seven met in Mainz on 14–16 December 1959. During this meeting, in a small group engaged for several days on discussing difficult problems that had already engaged the members for several months previously, the personalities of all participants emerged strongly, although this is not visible from any of the technical documents produced at the time [1, document 2]. Right at the beginning of this meeting, while we were still walking among the university buildings on the way to the room of the meeting, Van Wijngaarden made a move which

in its friendly and polite manner and its subtle significance is highly characteristic of him. He simply said: "I think we should introduce ourselves, I am Aad." For an understanding of the significance of this suggestion it should be realized, first, that in the previous meetings the mode of personal address had conformed to the central European tradition of using surnames, and even titles, as 'Professor So-and-So'. The suggestion to use a more informal mode was therefore a general suggestion to be less formal, more direct and personal. But the suggestion had a more important implication. Until the meeting in Mainz, ALGOL in Europe was the result of the work of the GAMM-centered committee, and was felt to be, in a sense, the intellectual property of that committee. By making his polite suggestion, which conforms more to an Anglo-Saxon than to a central European style of address, Van Wijngaarden made it clear to everyone that from now on the influence on the language was shared equally by all seven members of the European group.

In accordance with the tone set by Van Wijngaarden's proposal of informality and equality the Mainz meeting became a highly effective collective effort. He himself fought valiantly for dynamic declarations based on the new-old idea, but with characteristic alertness dropped them when the arguments in favour of a block structure gathered force. From that moment he contributed cheerfully and actively to the shaping of the details of the block structure, the main result of the meeting [1, document 2].

At the final ALGOL 60 conference in Paris, 11–16 January 1960, Van Wijngaarden was first entrusted, together with Samelson, the important task of convincing the Americans of the merits of the European proposal for block structure. This took place in a committee of four, having Backus and Green as the American members. The result [1, document 11] was a unanimous recommendation of the European proposal with a few minor modifications, a decisive turning point of the conference, brought about, undoubtedly to a large extent by Van Wijngaarden's friendly and polite manner and his flexible intelligence.

The report of the sub-committee [1, document 11], in addition to the notes on block structure has a separate, concluding paragraph saying:

> The Committee recognizes the need for syntactically altering programs in various ways, and recommends that the present Algol Committee meets in Rome in May 1960 to consider the specification of a Meta Algol and Processor.

Needless to say, this recommendation was not followed. It is, however, an interesting evidence of the optimism with regard to the speed with which programming language ideas could be developed, held by some of the members of the ALGOL 60 Committee.

At the later stages of the ALGOL 60 conference in Paris Van Wijngaarden was a member of several sub-committees that had decisive influence on the shaping of the procedure concept. During the early stages of the conference several proposals for the semantics of procedure parameters, based on a distinction between input and output parameters, had been considered and rejected by the full committee. As a result a new sub-committee, having Katz, Van Wijngaarden, and Woodger, as members, was given the task to consider procedure calls with only one list of parameters. The main part of their report [1, document 17] reads as follows:

> For the successful use of a procedure, its purpose must be understood, and parameters appearing in a call of the procedure must be in accordance with the expressed intentions of the author. For this reason no formal rules governing admissible actual parameters should be made. In particular, we need not even specify which are input and which are output parameters, and then the rules for replacement of formal by actual parameters on page 53 of DOCUMENT 4B (line 7–20) [quoted below] continue to apply for simple variables. A procedure may conceivably make use as parameters of all kinds of identifiable entities, and for each of these appropriate replacement rules must be given, whether the proposal to amalgamate input and output parameters is accepted or not.

DOCUMENT 4B (the Zürich report [2, p. 53, line 7–20]) says:

> Within a program, a procedure statement causes execution of the procedure called by the statement. The execution, however, is effected as though all formal parameters listed in the procedure declaration heading were replaced, throughout the procedure, by the actual parameters listed, in the corresponding position, in the procedure statement.
>
> This replacement may be considered to be a replacement of every occurence within the procedure of the symbols, or sets of symbols, listed as formal parameters, by the symbols,

> or sets of symbols, listed as actual parameters in the corresponding positions of the procedure statement, after enclosing in parentheses every expression not enclosed completely in parentheses already.
>
> Furthermore, any **return** statement is to be replaced by a **go to** statement referring, by its label, to the statement following the procedure statement, which, if originally unlabled, is treated as having been assigned a (unique) label during the replacement process.

While these formulations, which were adopted by the ALGOL 60 committee, leave many aspects of procedures open, they do make it clear that the adoption in ALGOL 60 of the parameter replacement mechanism of ALGOL 58, and the rejection of the input/output distinction, was made knowingly and deliberately.

The final decisions concerning procedures were based on a report [1, documents 26 and 27] submitted by a sub-committee whose members seem not to have been recorded, but which I believe included Perlis and Van Wijngaarden. The report on procedure statements describes the handling of parameters as follows:

> The execution is effected as though the values or names respectively of all formal parameters listed in the formal part of the procedure declaration heading were replaced throughout the procedure compound by the values or the names respectively of the actual parameters in the corresponding positions in the procedure statement.
>
> The correspondence between the actual parameter and the formal one is by list position, i.e. list position defines correspondents. The treatment of the correspondents is determined by the name list associated with the formal parameter list in the procedure heading. A name is taken if the corresponding formal parameter appears in the name list; otherwise the value is taken. A procedure statement is only defined if the correspondents are compatible, i.e. when the correspondent is specified by value or name respectively that the types or kinds respectively correspond. The value or name – whichever is indicated – of each of the actual parameters is substituted appropriately in the procedure compound – includ-

> ing declarations – according to the following prescription: if specification is made by name, the name of the actual parameter is substituted for all occurrences of the corresponding formal parameter in the procedure compound. If specification is made by value, the value of the actual parameter is assigned to the corresponding formal parameter as an initialisation of the procedure compound. If the parameter is a label by assignment is meant the same as replacement. If the parameter is an array, the consequences of mismatch of the dimensions of the correspondents are undefined.

As a further help to understanding these rules an addition to the draft report [1, document 31, item 174] was submitted by, I believe, Van Wijngaarden:

> 2.7. Names
>
> The name of an identifier is that identifier.
>
> The name of a variable or expression is the (name of the) identifier associated with that variable or expression, respectively.
>
> The name of an array, function or procedure is that function [sic.] identifier, procedure [sic.] identifier or procedure identifier associated with that array, function or procedure, respectively.

As has been described elsewhere [5] the adoption of these proposals by the full ALGOL 60 committee did not clarify the procedure concept sufficiently for the final formulation, and another round of exchanges of proposals, by letter, followed the ALGOL 60 meeting in Paris, during the time 17 to 25 January 1960.

Although most of the formulations quoted above have authors in addition to Van Wijngaarden, I think they can serve as support of a characterization of his distinctive contribution to ALGOL 60. Running through the proposals in which he has a hand is a keen openness to new solutions, which, however, are always characterized by being based on few, very general notions, and described briefly and elegantly. When this is said it must also be admitted by in pursuing this direction there is a risk that the simplicity and elegance may be deceptive, may cover complications and obscurities. This risk was demonstrated by the discussion and feeling of

uncertainty that was provoked in the ALGOL 60 committee by Van Wijngaarden's description of the concept of name.

At this point we may also note some proposals from the Amsterdam team that were upheld during the Paris meeting by Van Wijngaarden but rejected by the full committee. The most remarkable such proposal was the one for having the possibility to declare dummy variables explicitly, as presented in Algol Bulletin 7.34 quoted above. The trouble about this proposal, and the main reason why it was rejected, was that it was never developed beyond the initial suggestion. It was just an intriguing idea, but one whose concrete implications and relation to other parts of the programming language remained obscure.

The situation with respect to the Amsterdam proposal (AB 7.35, quoted above) for admitting an extended range of types, including complex numbers, vectors, matrices, and lists, was similar. In this case the general, mathematical notions behind the proposal were of course well known, but at the same time is was increasingly clear that it requires a lot more than just a mathematically well-defined notion to achieve a data type notion that is adequate for inclusion in a common programming language. Indeed, even just the clarification of the handling of integers and reals in ALGOL 60 required extensive discussion in a sub-committee of the ALGOL 60 meeting in Paris [1, documents 15, 16].

For use as illustrations in the final ALGOL 60 Report all members of the committee were urged to submit sample programs. Only Van Wijngaarden and Rutishauser responded to this, Van Wijngaarden on 4 February 1960 sending in the procedure euler that appears as Example 1 in the ALGOL 60 Report [4] and also a procedure similar in operation to the one submitted a few weeks later by Rutishauser, which appears as Example 2. Thus if we disregard the examples in the main section of the report, Van Wijngaarden must be the first person to have a numerical algorithm written in ALGOL 60 published.

The final contribution of Van Wijngaarden to the formulation of the ALGOL 60 Report was his and Dijktra's suggestion, made in a telephone call from Amsterdam to Copenhagen on about 10 February 1960, that a sentence be added to the draft report so as to make it clear that the language admits recursive procedure calls. This particular issue has been dealt with at length in an earlier study [6, pp. 159–160] from which it should be clear that the members of the ALGOL 60 committee do not agree on the significance of this incident. However, as far as Van Wijngaarden

himself is concerned it is quite clear that he regarded this action, by Dijkstra and himself, as a contribution to the clarity and completeness of description of the language already fully defined in the draft report, not as a reversal of a committee decision to disallow recursive procedure activations. In any case, the concrete proposal itself, due, I believe, to Dijkstra, is a highly characteristic piece of brevity and elegance:

> Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

While the publication of the ALGOL 60 Report [4] in March 1960 may be said to terminate the creation of the language itself, when considering the contributions of the Amsterdam team to establishing the language in a wider sense the field of ALGOL 60 compiler construction must be mentioned. The fact is that only a few months after the final definition of the new language, in June 1960, Van Wijngaarden could announce the successful completion, by Dijkstra and Zonneveld, of the first compiler for the language, working on the Electrologica X1 computer. This result had an enormous impact as a support of ALGOL 60, both in terms of the actual compiler techniques employed, which were subsequently used and adapted widely, but perhaps even more as a proof that the language could be implemented almost in its entirety by a team of quite modest capital resources. It was, in very concrete terms, a confirmation of the benefits that may be gained from insisting on simplicity and generality in programming language design.

As the conclusion of these notes, although Van Wijngaarden contributed actively to all the central parts of ALGOL 60, it is difficult to identify any definite part of the language as contributed particularly by him. In fact, it is much easier to point to ideas that he proposed for the language, but that were eventually rejected in the language design process. The point is that his main influence on ALGOL 60 was less tangible, but not less strong for that reason. Van Wijngaarden's manner, his friendliness, politeness, cheerfulness, quick comprehension, flexibility of mind, all of these were exceedingly helpful in shaping the concepts and in smoothing the discussion. And the direction of his influence, his mental style, his striving for simplicity and generality, certainly have left their mark on the final language. For these reasons all of us who have benefitted from ALGOL 60, in any way whatever, owe him our recognition and gratitude.

References

[1] ALGOL 60 documents 1959–60. (Unpublished technical memoranda prepared in connection with the ALGOL 60 conference in Paris, 11–16 January 1960.)
2: European representatives to the ALGOL 60 conference 14–16 December 1959. Meeting of the European representatives to the Algol conference, Mainz, 4 pp.
11: J. Backus, J. Green, K. Samelson and A. van Wijngaarden, 13 January 1960, Report of the committee on local, etc., 1 p.
15: P. Naur and A. Perlis, Meaning of types and assignments, 1 p.
16: P. Naur and A. Perlis, 13 January 1960, Types of expressions – assignments, 3 pp.
17: C. Katz, A. van Wijngaarden and M. Woodger, 14 January 1960, Report of the committee on procedure declarations and procedure calls with only one list of parameters, 1 p.
26: Procedure statements, 16 January 1960, 2 pp.
27: Procedure declarations, 16 January 1960, 2 pp.
31: ALGOL 60 committee, 13–16 January 1960, First and second list of suggested changes in document 5. Each item is authored by a member of the committee. The items are numbered 101 to 175, followed by 173–175 (used again) and ∞-1, 33 pp.

[2] Algol Bulletin, P. Naur (Ed.), Mimeographed discussion letters, No. 7, 3 November 1959, 21 pp. No. 8, 12 December 1959, 15 pp. Regnecentralen, Copenhagen.

[3] J.W. Backus, F.L. Bauer, H. Bottenbruch, C. Katz, A.J. Perlis (Ed.), H. Rutishauser, K. Samelson (Ed.), and J.H. Wegstein, Report on the algorithmic language ALGOL, Num. Math. 1 (1959) 41–60. Also: Preliminary report – international algebraic language, Comm. ACM 1 (12) (1958) 8–22.

[4] J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur (Ed.), A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden and M. Woodger, Report on the algorithmic language ALGOL 60, Num. Math. 2 (1960) 106–136. Also: Comm. ACM 3 (5) (1960) 299–314.

[5] P. Naur, The European side of the last phase of the development of ALGOL 60. In: R.L. Wexelblat (Ed.), History of Programming Languages (Academic Press, New York, 1981) pp. 92–139.

[6] R.L. Wexelblat (Ed.), History of Programming Languages (Academic Press, New York, 1981) 758 pp.

Algorithmic Languages, de Bakker/van Vliet (eds.)

On the Notion of Strong Typing

Maarten M. Fokkinga

Twente University of Technology, P.O. Box 217, 7500 AE Enschede, The Netherlands

The usefulness of strong typing is formalized in the following way. Strong typing is a syntactic means to restrict the class of programs so that a pleasant semantic property holds. More precisely, a semantic equivalence of strongly typed programs is proved independent of the representation used to implement abstract entities like numbers, truth values and predefined ones.

Thus a formal content is given to phrases like "typing prevents to employ unintended properties of representations" and "semantically types are redundant".

1. Introduction

It seems widely accepted that so-called strong typing has some undeniable benefits. E.g. the ALGOL 68 designers claim that "ALGOL 68 has been designed in such a way that most syntactic errors and many others can be detected easily before they lead to calamitous results" [19, Section 0.1.3]. Undoubtly it is its mode discipline which plays a major role in this error detection (see [6, 8]). Indeed, "one often pays a price for [the absence of a type system] in the time taken to find rather inscrutable bugs – anyone who mistakenly applies CDR to an atom in LISP, and finds himself absurdly adding a property list to an integer, will know the symptoms" [11].

It is therefore not surprising that the following requirement is included in STEELMAN [2]:

> "3A. Strong Typing. The language shall be strongly typed. The type of each variable, array, record, expression, function and parameter shall be determinable during translation".

But STEELMAN neither provides a formal definition of strong typing, nor

does it give any semantic property aimed at in requiring strong typing. So how could one prove that ADA meets the requirements or desiderata?

In this paper we investigate what formally the usefulness of strong typing might be. To this end we view typing as a purely syntactic way of restricting the class of programs so that a pleasant semantic property holds for that class, and we thus formalize the interplay between the syntactic typing and the semantic properties of programs. This view is in accordance with [13] and [12], and seems consistent with practical implementations of strongly typed languages. Nevertheless one mostly finds types motivated in a setting where semantic entities (like retracts [16] and [3], downward closed directed c.p.o.'s [11] and so on) are assigned to types.

Our paper might be viewed as a continuation of [15] and [3]. They both present a theorem which we call the Correspondence Theorem. Informally this theorem asserts that there is a relation, called correspondence, which relates for any strongly typed program the values denoted under different implementations. However, both assign a semantics to types. We are glad to improve their results in that we show types to be semantically redundant. Moreover we prove a nicer theorem (Theorem 3.10) which asserts that a *semantic equivalence of programs* is independent of the implementation.

The formalization and proofs are carried out in the framework of the typed λ-notation. We define two expressions equivalent with respect to some type t if their values, when used according to t, are the same function – or constant.

The remainder of the paper is organized as follows. In Section 2, we formally define syntactic concepts of the language, and define some axioms which are to characterize the semantics. In Section 3, the formalization of the usefulness of strong typing is presented. Thereafter, in Section 4, we give a specific semantics of the language, satisfying the axioms; that section only serves to provide a concrete example. Finally we conclude with Section 5, discussing the results obtained.

2. The Language

We choose a simple language to illustrate the essential ideas. Obviously, then, the language has to have a construct where type checking is involved, say function application or assignment. Moreover the language has to have a construct for user controlled creation of new values; were this not the case

there would be no problems at all, because one must of course assume that all 'predefined' values behave well. In view of its simple semantics we are led to consider the λ-notation; λ-abstraction is the construct to create new values.

Definition 2.1 (Expressions and Types). Let X be a countably infinite set of *normal identifiers* and let Z be a set of *type identifiers*. Throughout the paper we let x and y vary over X and z over Z. Specific elements of X are e.g.

$$zero,\ one,\ succ,\ pred,\ true,\ false,\ldots$$

and specific elements of Z are

$$int,\ bool,\ldots.$$

The set T of *types* is defined thus

$$t ::= z \mid (t \to t').$$

The set E of *expressions* is defined thus

$$e ::= x \mid (\lambda x : t\,.\,e) \mid e(e').$$

Throughout the paper we let e vary over E and t over T; we sometimes suffix them with digits, primes and letters *f*, *a* and *b* (for *f*unction, *a*rgument and *b*ody). According to common usage we omit parentheses when they are clear from the context; in particular the scope of λ extends as far as possible, and $\to$ associates to the right, so that $t1 \to t2 \to t3 = t1 \to (t2 \to t3)$.

Notice that there are no constants like **0, 1, 2, ...**; predefined identifiers like *zero, one, two, ...* (or even *zero* and *succ* alone) should enable the programmer to use numbers. Other interesting predefined identifiers may be the so-called fixed point operators, $fixpoint_{t,t'}$ of type $((t \to t') \to (t \to t')) \to (t \to t')$, to enable recursive definitions.

Syntactic sugar might be added to make the language more practical. E.g. non-recursive definitions can be introduced as an abbreviation:

$$\textbf{let } x : t = e' \textbf{ in } e$$

and

$$e \textbf{ where } x : t = e'$$

abbreviate $(\lambda x : t \,.\, e)(e')$. Also conditional expressions can be introduced:

if e **then** $e1$ **else** $e2$

where both $e1$ and $e2$ have type t, abbreviates

$$cond_t(e)(\lambda x : null \,.\, e1)(\lambda x : null \,.\, e2)$$

where $cond_t$ has type $bool \to (null \to t) \to (null \to t) \to t$. All this is well known, see e.g. [17].

We now define what expressions are well typed. The formal term used is strong typing. Informally it means that for each application the type of the argument must match the parameter type of the function. In our simple language two types match iff they are equal; in a more elaborate language a less trivial relation may hold.

The type of identifiers depends on the context in which they occur. We model that context by a so-called syntactic environment. Formally, the set S of *syntactic environments* is the set of partial functions $X \to T$. Throughout the paper we let s vary over S. For each s we assume that there exists an identifier x which has not yet a type associated with it; we say that $new(x, s)$ holds in that case. In view of the infinity of X this is no strong requirement.

As usual the suffix $[p \leftarrow q]$ denotes *updating of a function;* in particular

$$s[x \leftarrow t](x') = \text{if } x = x' \text{ then } t \text{ else } s(x').$$

This notation will also be used for semantic environments r, to be introduced below.

Definition 2.2 (Strong Typing). The relation $s \vdash e : t$ (*"e has type t in s"*) is the smallest relation satisfying

(a) if $s(x) = t$, then $s \vdash x : t$;

(b) if $s[x \leftarrow ta] \vdash eb : tb$, then $s \vdash (\lambda x : ta \,.\, eb) : (ta \to tb)$;

(c) if for some ta, $s \vdash ef : ta \to tb$ and $s \vdash ea : ta$, then $s \vdash ef(ea) : tb$.

We say e *is strongly typed in* s if for some t, $s \vdash e : t$.

Now we turn to the semantics of the language. Let V be the set of *values* which serve as meanings for expressions, and let $R = X \to V$ be the set of *semantic environments* giving the meaning of the predefined identifiers. (Throughout we let v and w vary over V and r over R.) The *meaning of an expression* e is then given by $M(e, r)$, where $M \in E \times R \to V$ is the so-called meaning function (a partial function).

Usually the meaning of expressions are taken to be some *abstract entities,* like numbers, truth values or functions. Accordingly the domain of numbers is associated with the type identifier *int,* the domain of truth values is associated to *bool,* and – sometimes mathematically quite sophisticated – functional domains are associated to types $t \to t'$. Actually, however, expressions yield bit patterns, or the like, which in some way or another represent those abstract entities. And accordingly, from the bit pattern alone, say *concrete value,* one can not tell whether it is meant as a number, truth value or function. It is indeed quite possible to execute a bit pattern meant as a number as if it represents a function. Thus semantically types do not enter the picture.

Admittedly, mostly the abstract entities are of interest. But the interpretation of the concrete values cannot be the task of the language designer, i.e. is not incorporated into M. Even if M would produce numbers, then still these numbers represent some more abstract entities like year of birth, salary and so on. The interpretation is really outside the grip of M, and is left to the individual programmer and creator of the standard environment.

Consequently the value denoted by an expression is possibly untyped. We will however not burden the reader/programmer with details of the value space V, but instead specify the meaning of expressions by the axioms which we need in the proofs below.

Definition 2.3 (Axioms for M). For strongly typed expressions the meaning function satisfies the following axioms.

(a) $M(x, r) = r(x)$;

(b) if $v = M(ea, r)$, then $M((\lambda x : ta \,.\, eb)(ea), r) = M(eb, r[x \leftarrow v])$;

(c) if y not free in e, then $M((\lambda x : ta \,.\, e)(ea), r) = M((\lambda y : ta \,.\, e[x/y])(ea), r)$;

(d) if $v = M(e', r)$ and x does not occur free in the scope of some λ within e, then $M(e[x/e'], r) = M(e, r[x \leftarrow v])$.

Above, and in the sequel, we use the postfix $[x/e']$ to denote *substitution* of e' for x – taking care to rename bound identifiers in order to avoid clash of names.

Notice that $M(\lambda x : t \,.\, e, r)$ need not be a function. All we require is that it can be used as a function in the sense of axioms (b) and (c). Indeed, the M given in Section 4 will yield some code of a function, so that e.g. $M(\lambda x : t \,.\, e, r)$ differs from $M(\lambda y : t \,.\, e[x/y], r)$. Actually in Section 4 we take

V to be a set of untyped values, so that any value may be used in any way, and M even satisfies the axioms for not strongly typed expressions.

In the sequel we will use the following abbreviations.

(1) '$v(w)$' abbreviates $M(x(y), r[x \leftarrow v, y \leftarrow w])$, and is thus a concise way of expressing that v is to be used as a function with argument w.

(2) $e1 =_r e2$ abbreviates $M(e1, r) = M(e2, r)$; $e1$ and $e2$ yield the same value in r.

3. Formalizing the Usefulness of Strong Typing

We will first introduce the syntactic concept of primitive expressions. These denote what one might call predefined values and they are used to state assumptions on alternative representations for the same set of abstract entities. Secondly we define the semantic relation of correspondence and some properties of it. The correspondence relation is used in the proof of Theorem 3.10 which expresses our view on the usefulness of strong typing.

Suppose that the standard environment r provides via $zero : int$ and $succ : int \rightarrow int$ an implementation for numbers. Of course, the concrete value denoted by *zero* is not the number zero, but merely represents it in some way or another. We may also consider an alternative implementation $\hat{r}$. Surely $r(zero)$ and $\hat{r}(zero)$ need not be equal, although they both represent the same abstract entity. E.g. the expressions

$$zero,\ succ(zero),\ succ(succ(zero)), \ldots$$

constitute the – unknown – representation of numbers. And if e.g. $pred : int \rightarrow int$ is also present, then

$$pred(zero),\ pred(succ(zero)),\ succ(pred(zero)), \ldots$$

might also contribute to the representation. However note that abstractions like $\lambda x : int \,.\, x$ or $\lambda x : int \,.\, zero$ do not contribute to the representation of abstract entities as far as determined by the environment. Thus we are led to the following definition to get some grip on the representations of abstract entities.

Definition 3.1 (Primitive expressions). For any s the set P of *primitive expressions* consists of all strongly typed expressions p generable by

$$p ::= x \mid p(p).$$

In the sequel p varies over P.

It will turn out that, for fixed s and r, the primitive expressions of type z constitute all expressible 'z-values'. Thus they play the role usually played by constants. However we do not restrict the types of the given function identifiers to first order; a function identifier *mk-int*:$((t \to t) \to int)$ may occur in the primitive expressions and so contribute to the values representing *'int'*s.

Given two environments r and $\hat{r}$, we wish to define a correspondence relation $\sim_t$ on $V \times V$, relating those values which *wrt* r resp $\hat{r}$ represent the same abstract entity. As one concrete value may represent a variety of abstract entities (e.g. 001 may represent both the number one and the truth value true, and many more), we need to indicate with respect to what interpretation the correspondence is to be understood. The type t serves that purpose. Of course we want $M(p, r) \sim_z M(p, \hat{r})$ for p of elementary type z; thus the relation also depends on s.

Definition 3.2 (Correspondence). For any s, r, $\hat{r}$ and t the relation $s, r, \hat{r} \vdash v \sim_t \hat{v}$ (*"v and $\hat{v}$ represent the same abstract entity"*) is defined by induction on t as follows:

(a) $t = z$: $s, r, \hat{r} \vdash M(p, r) \sim_z M(p, \hat{r})$ for any p with $s \vdash p : z$;

(b) $t = ta \to tb$: $s, r, \hat{e} \vdash v \sim_t \hat{v}$ iff for all w, $\hat{w}$ with $s, r, \hat{r} \vdash w \sim_{ta} \hat{w}$, also $s, r, \hat{r} \vdash$ '$v(w)$' $\sim_{tb}$ '$\hat{v}(\hat{w})$'.

We cannot expect to derive any interesting property for the correspondence relation unless we assume consistency between the two environments. In particular the following predicate *Correct*$\sim(s, r, \hat{r})$ is reasonable.

Definition 3.3 (Correct$\sim$). *Correct*$\sim(s, r, \hat{r})$ holds iff for all x, t with $s \vdash x : t$

$$s, r, \hat{r} \vdash M(x, r) \sim_t M(x, \hat{r}).$$

The following lemma shows that a seemingly stronger requirement for *Correct*$\sim(s, r, \hat{r})$ actually already follows from the given definition.

Lemma 3.4. *Let s, r, $\hat{r}$ satisfy Correct*$\sim(s, r, \hat{r})$. *Then, for any p, t with*

$s \vdash p : t$,

$$s, r, \hat{r} \vdash M(p, r) \sim_t M(p, \hat{r}).$$

Proof. By induction on the structure of p.

The following lemma is needed to prove the Stability of Correspondence Lemma below, which in turn is needed in the Correspondence Theorem following it. Both lemmata are of a rather technical nature. They show that updating of s, r, $\hat{r}$ to $s[x \leftarrow t]$, $r[x \leftarrow v]$, $\hat{r}[x \leftarrow \hat{v}]$ under certain circumstances does not change the relation $\sim_t$.

Lemma 3.5. *Let s, r, $\hat{r}$ satisfy Correct$\sim(s, r, \hat{r})$; let w, $\hat{w}$, ty satisfy $s, r, \hat{r} \vdash w \sim_{ty} \hat{w}$; let y be new in s, i.e. new(y, s). Then for any p, t with $s' \vdash p : t$,*

$$s, r, \hat{r} \vdash M(p, r') \sim_t M(p, \hat{r}')$$

where $s' = s[y \leftarrow ty]$, $r' = r[y \leftarrow w]$, $\hat{r}' = \hat{r}[y \leftarrow \hat{w}]$.

Proof. By induction on the structure of p.

Lemma 3.6 (Stability of Correspondence). *Let s, r, $\hat{r}$ satisfy Correct$\sim$ $(s, r, \hat{r})$; let w, $\hat{w}$, ty satisfy $s, r, \hat{r} \vdash w \sim_{ty} \hat{w}$; let y be new in s, new(y, s). Then for any v, $\hat{v}$, t*

$$s, r, \hat{r} \vdash v \sim_t \hat{v} \quad \textit{iff} \quad s', r', \hat{r}' \vdash v \sim_t \hat{v}$$

where $s' = s[y \leftarrow ty]$, $r' = r[y \leftarrow w]$, $\hat{r}' = \hat{r}[y \leftarrow \hat{w}]$.

Proof. By induction on the structure of t.

Case $t = z$, $\Rightarrow$. Assume $s, r, \hat{r} \vdash v \sim_z \hat{v}$. By definition, for some p with $s \vdash p : z$, $v = M(p, r)$ and $\hat{v} = M(p, \hat{r})$. Because *new*(y, s), y does not occur free in p, hence $v = M(p, r')$ and $\hat{v} = M(p, \hat{r}')$ and $s' \vdash p : z$. So by definition $s', r', \hat{r}' \vdash v \sim_z \hat{v}$.

Case $t = z$, $\Leftarrow$. Apply Lemma 3.5.

Case $t = ta \rightarrow tb$. Use the definition of correspondence and the induction hypotheses for both ta and tb.

Theorem 3.7 (Correspondence). *Let s, r, $\hat{r}$ satisfy Correct$\sim(s, r, \hat{r})$. Then for any e, t with $s \vdash e : t$*

$$s, r, \hat{r} \vdash M(e, r) \sim_t M(e, \hat{r}).$$

Proof. By induction on the structure of e.

Case $e = x$. Immediate from the assumption.

Case $e = ef(ea)$. Straightforward by induction.

Case $e = \lambda x{:}ta\,.\,eb$. Then for some tb, $t = ta \to tb$ and $s[x \leftarrow ta] \vdash eb{:}tb$. Now let w, $\hat{w}$ be arbitrary satisfying $s, r, \hat{r} \vdash w \sim_{ta} \hat{w}$. One may easily verify that $`M(e,r)(w)' = `M(\lambda x : ta\,.\,eb, r)(w)' = M(eb[x/y], r[y \leftarrow w])$ where y is chosen such that $new(y,s)$. Setting $s' = s[y \leftarrow ta]$, $r' = r[y \leftarrow w]$ and $\hat{r}' = \hat{r}[y \leftarrow \hat{w}]$, we can show $Correct \sim (s', r', \hat{r}')$ from the Stability of Correspondence Lemma. Hence we may apply the induction hypothesis and find

$$s', r', \hat{r}' \vdash M(eb[x/y], r') \sim_{tb} M(eb[x/y], \hat{r}').$$

As above $M(eb[x/y], \hat{r}') = `M(e, \hat{r})(\hat{w})'$, so that

$$s', r', \hat{r}' \vdash `M(e,r)(w)' \sim_{tb} `M(e,\hat{r})(\hat{w})'.$$

Using once more the Stability of Correspondence Lemma we find

$$s, r, \hat{r} \vdash `M(e,r)(w)' \sim_{tb} `M(e,\hat{r})(\hat{w})'.$$

We conclude therefore $s, r, \hat{r} \vdash M(e,r) \sim_t M(e,\hat{r})$.

Reynolds [15] and Donahue [3] give more or less this theorem as the effect strong typing has on the semantics of expressions. One may interpret the theorem that an implementor of the predefined values, accessible via the predefined identifiers, may freely switch from one representation r to another $\hat{r}$, provided $Correct \sim (s, r, \hat{r})$, without essentially affecting the value denoted by an expression: the two values do correspond and therefore do represent the same abstract entity; in particular if the expression has a non-composite type we know that the two values $M(e,r)$ and $M(e,\hat{r})$ arise from the same primitive expression.

Yet we feel a bit unhappy with this result; it involves too much hand waving to convince an unwilling listener of the importance. Fortunately there is a more appealing semantic property of strongly typed expressions. Switching from one representation to another does not affect the meaning of expressions in the sense that *semantic equivalence* is unaffected. Semantic equivalence need be defined precisely, because there are several reasonable choices, which in general do not coincide (see e.g. [1]). We choose the one in which two expressions e and e' are said equivalent with respect to a type $t1 \to t2 \to \cdots \to tn \to z$ if there is no context of the form $[\cdots](e1)(e2)\cdots(en)$ with $e1 : t1, \ldots, en : tn$ which discriminates between e

and e'; i.e. $e(e1)(e2)\cdots(en)$ and $e'(e1)(e2)\cdots(en)$ yield the same value. Formally, we define this notion by induction on t.

Definition 3.8 (Equivalence). For any s, r, $e1$, $e2$ we define $s,r \vdash e1 \simeq_t e2$ (*"e1 and e2 are equivalent wrt t"*) as follows.

(a) for $t = z$: $s,r \vdash e1 \simeq_z e2$ if $e1 =_r e2$;

(b) for $t = ta \leftarrow tb$: $s,r \vdash e1 \simeq_t e2$ if for all e with $s \vdash e : ta$, $s,r \vdash e1(e) \simeq_{tb} e2(e)$.

Notice that $s,r \vdash e1 \simeq_t e2$ in itself does not require that $s \vdash e1, e2 : t$. Hence it makes sense to consider the question whether any $e1$ and $e2$ are equivalent. In particular we may consider expressions which are not strongly typed, but are *weakly typed* according to [3]. Some simple examples are treated after Theorem 3.10.

An alternative notion of equivalence is the following. Two expressions $e1$ and $e2$ are said equivalent *wrt* type t if there is no strongly typed context $C[\cdots]$ with a hole of type t and as a whole of type z, for some z, such that $C[e1]$ and $C[e2]$ have different values; cf. [10]. Our Theorem 3.10 fails for this notion because of possible pathological values for higher order function identifiers. We might exclude such values by suitable assumptions about r, but we will not pursue this alternative here.

We can of course not expect to prove that equivalence is independent of the environment, unless we assume some consistency requirements between the environments under consideration. In particular the following predicate $Correct \simeq (s,r,\hat{r})$ seems reasonable.

Definition 3.9 (Correct $\simeq$). $Correct \simeq (s,r,\hat{r})$ holds iff for all $p1$, $p2$, z with $s \vdash p1, p2 : z$

$$s,r \vdash p1 \simeq_z p2 \quad \textit{iff} \quad s,\hat{r} \vdash p1 \simeq_z p2.$$

Theorem 3.10 (Representational Independence of Equivalence). *Let s, r, $\hat{r}$ satisfy $Correct \sim (s,r,\hat{r})$ and $Correct \simeq (s,r,\hat{r})$. Then for any $e1$, $e2$, t with $s \vdash e1, e2 : t$*

$$s,r \vdash e1 \simeq_t e2 \quad \textit{iff} \quad s,\hat{r} \vdash e1 \simeq_t e2.$$

Proof. By induction on t.

Case $t = z$, $\Rightarrow$. From $s, r \vdash e1 \simeq_z e2$ we find $e1 =_r e2$ (1)

From $s, r, \hat{r} \vdash M(e1, r) \sim_z M(e1, \hat{r})$ (by the Correspondence Theorem) and similarly for $e2$, we find by the Correspondence Definition

$$\text{for some } p1 \text{ with } s \vdash p1 : z,\ e1 =_r p1 \text{ and } p1 =_{\hat{r}} e1,$$

$$\text{for some } p2 \text{ with } s \vdash p2 : z,\ e2 =_r p2 \text{ and } p2 =_{\hat{r}} e2.$$

Hence by (1) $p1 =_r p2$, so $s, r \vdash p1 \simeq_z p2$, so by *Correct*$\simeq$ $(s, r, \hat{r})$ also $s, \hat{r} \vdash p1 \simeq_z p2$, so $p1 =_{\hat{r}} p2$ and hence $e1 =_{\hat{r}} e2$, i.e. $s, \hat{r} \vdash e1 \simeq_z e2$.

Case $t = z$, $\Leftarrow$. Similar.

Case $t = ta \to tb$. Easy by induction.

It is not difficult to construct counter examples to the conclusion of the theorem in case the condition $s \vdash e1, e2 : t$ is not met. E.g. consider the syntactic environment with $zero : int$ and $true, false : bool$. Now let the representation of booleans be a subset of the representation of the integers. In particular choose r and $\hat{r}$ such that

$$r(zero) = r(true) \neq r(false),$$

$$\hat{r}(zero) = \hat{r}(false) \neq \hat{r}(true).$$

Clearly

$$s, \hat{r} \vdash zero \neq_t true \quad \text{for all noncomposite } t \in Z,$$

but yet

$$s, r \vdash zero \simeq_t true \quad \text{for all } t.$$

Donahue [3] defines a notion of weak typing so that $e = (\lambda x : bool\,.\, x)(zero)$ is weakly typed and has type *bool*. Again we find $s, r \vdash e \simeq_{bool} true$ but $s, \hat{r} \vdash e \neq_{bool} true$. Thus relaxing the requirement $s \vdash e1, e2 : t$ in the theorem to "$e1$ and $e2$ must be weakly typed, with type t say, in s" invalidates the conclusion.

4. A concrete semantics for the language

This section only serves to show that untyped values and coinciding representations are quite reasonable. We will work out the set V and function M, without any sophisticated mathematical constructions as commonly used in the field of denotational semantics, cf. [3, 9, 15, 16].

Our starting point is that values are untyped, like bit patterns, and that each value may be used in any way. This is just the opposite of Definition 2.1.1.2.c of the ALGOL 68 Report [19], and of the postulation by [5]. For ease of presentation we choose a set V which suits our purpose very well.

Definition 4.1 (The value space V). Let C be a fixed set of *constants,* disjoint from X. The set U of *pseudo-values* is defined by BNF:

$$u ::= x \mid (\lambda x.\, u) \mid u(u') \mid c.$$

The set V of *values* is defined thus

$$V = \{u \in U \mid \text{no } x \in X \text{ occurs free in } u\}.$$

Throughout v and w vary over V; specific elements of C are $\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{S}, \mathbf{P}, \ldots$.

Values may be thought to model states of a machine. Possible state transitions are modelled by transformation or reduction rules. A completed transformation of some initial state v into a final state is called the elaboration of v. We choose here a deterministic transformation in applicative order ('call by value'), cf. the SECD machine of [7].

Definition 4.2 (Transformation rules and Elaboration). The deterministic *transformation* $v \to w$ is defined thus:

(a) if $v \to v'$, then $v(w) \to v'(w)$;
(b) if $\forall v'.\ v \not\to v'$ and $w \to w'$, then $v(w) \to v(w')$;
(c) if $\forall w'.\ w \not\to w'$, then $(\lambda x.\, v)(w) \to v[x/w]$;
(d) for each $c \in C$ there is a fixed set of rules

$$c(v1)(v2) \cdots (vn) \to w$$

which respects the deterministic applicative order.

The elaboration $elab \in V \to V$ (a partial function) is given by

$$elab(v) = w \quad \text{if } v \xrightarrow{*} w \text{ and } \forall w'.\ w \not\to w'.$$

In the above framework "fatal errors during elaboration" may be modelled by nontermination. To this end let $\mathbf{error} \in C$ with $\mathbf{error} \to \mathbf{error}$.

Abstract entities like natural numbers N or truth values may be represented in V in a variety of ways, as shown in the next example.

Example 4.3 (Representations of natural numbers). One way is to let $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots \in C$ and to represent $n \in N$ by the obvious constant, say $\mathbf{c}_n$. Further, let $\mathbf{S}, \mathbf{P} \in C$ represent the successor and predecessor function. The following rules are needed: for all n

$$\mathbf{S}(\mathbf{c}_n) \to \mathbf{c}_{n+1},$$

$$\mathbf{P}(\mathbf{c}_{n+1}) \to \mathbf{c}_n.$$

Alternatively, we may represent n by

$$(\lambda x.\, \lambda y.\, x^n(y)) = \lambda x.\, \lambda y.\, x(x(\cdots x(y)\cdots)),$$

and the successor by $\lambda x.\, \lambda y.\, \lambda z.\, y(x(y)(z))$ and the predecessor either by

$$\mathbf{P} \in C \text{ with } \mathbf{P}(\lambda x.\, \lambda y.\, x^{n+1}(y)) \to \lambda x.\, \lambda y.\, x^n(y)$$

or by

$$\lambda z.\, (z(\lambda x.\, \lambda y.\, y((\lambda x.\, \lambda y.\, \lambda z.\, y(x(y)(z)))(x(\lambda x.\, \lambda y.\, x)))$$
$$(x(\lambda x.\, \lambda y.\, x)))(\lambda z.\, z(\lambda x.\, \lambda y.\, y)(\lambda x.\, \lambda y.\, y)))(\lambda x.\, \lambda y.\, y),$$

from [17]. There are various other representations with constant-free values, and which have a lower elaboration complexity (see [14]).

In particular the last representation in the above example shows that values are untyped. $\lambda x.\, \lambda y.\, y$ represents the number zero, but it may be applied to any value. In fact it also represents any function $f \in A \to B \to B$ with

$$f(a) = \text{identity function on } B.$$

Finally we define M. The role of types is to single out the strongly typed expressions, i.e., those for which Theorems 3.7 and 3.10 hold. Semantically "types are redundant."

Definition 4.4 (The meaning function M). The *compilation* $\bar{\ } \in E \to U$ is defined thus (it throws away all types):

(a) $\bar{x} = x$,

(b) $\overline{(\lambda x : t.\, e)} = (\lambda x.\, \bar{e})$,

(c) $\overline{e(e')} = \bar{e}(\bar{e'})$.

The meaning function $M \in E \times R \to V$ is defined

$$M(e, r) = elab(\bar{e}[x/r(x), \text{ for each } x \text{ free in } \bar{e}]).$$

It should be easy to verify the axioms assumed in Definition 2.3, and to construct suitable values for the identifiers $fixpoint_{t,t'}$ and $cond_t$ mentioned in Section 2.

5. Conclusion

We have shown that strong typing may be viewed as a purely syntactic means to restrict the class of expressions so that a nice semantic property holds. This view is consistent with practice where types are semantically (i.e. during run-time) redundant and values are really untyped.

The explicit formulation of the usefulness of strong typing makes it possible to discuss formally whether strong typing is desirable, provides a clear goal to aim at in the design of a type system, and enables a formal proof that a language, which claims to be strongly typed, satisfies that property. Thus we have a framework to discuss the type systems of [15], of ALGOL 68 and of modern languages with highly advanced type systems like LAWINE [18].

For example, [15] extends the λ-notation with a facility to pass types as a parameter. It presents no problems at all to extend our definitions, theorems and proofs to cover that extension too, see [4]. On the other hand the decision in ALGOL 68 that **struct(real** *re, im*) and **struct(real** *rho, phi*) are not equivalent seems irrelevant to maintain the representational independence of equivalence. Here, we think the ALGOL 68 designers have (mis)used the concept of strong typing in order to achieve in this particular case and in an ad-hoc way that those modes are more or less primitive. A facility to declare a type primitive, as in [15], would provide a more general solution, with no need to break the full structural equivalence of modes.

Of course, before we can make precise the above claims, further investigation is needed to extend the concepts of this paper to other language features. The introduction of cartesian product and discriminated union, and of variables and assignment, seems to be straightforward. More attention is needed for subtypes. And recursively defined types are problematic. E.g. the definitions cannot easily be adapted for the type $z = z \to z$. However, we conjecture that adaptations are possible for reducing types [1] like

$$fct = fct \times int \to int$$

which may be used to define the factorial function in the following way.

$$f : fct = \lambda g : fct, i : int \,.\, \textbf{if } i = 0 \textbf{ then } 1 \textbf{ else } i * g(g, i-1);$$

$$fact : int \to int = \lambda i : int \,.\, f(f, i).$$

Acknowledgement

I am grateful to Joost Engelfriet for stimulating and helpful discussions. He has also pointed out a serious error in earlier versions of this paper.

References

[1] E. Astesiano and G. Costa, Languages with reducing reflexive types, in: J.W. de Bakker and J. van Leeuwen (Eds.), Automata Languages and Programming, Lecture Notes in Computer Science, Vol. 85 (Springer, Berlin, 1980) pp. 38–50.

[2] Department of Defense (U.S.A.), Requirements for high order computer programming languages (1978).

[3] J. Donahue, On the semantics of "data type", Siam J. Comput. 8 (4) (1979) 546–560.

[4] M.M. Fokkinga, in preparation.

[5] C.A.R. Hoare, Notes on data structuring, in: O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare (Eds.), Structured Programming (Academic Press, London, 1972).

[6] C.H.A. Koster, The mode-system in ALGOL 68, in: S.A. Shuman (Ed.), New Directions in Algorithmic Languages 1975 (IRIA, 78150 Le Chesnay, 1975) pp. 99–114.

[7] P.J. Landin, The mechanical evaluation of expressions, Comput. J. 6 (1964) 308–320.

[8] L. Meertens, Mode and meaning, in: S.A. Shuman (Ed.), New Directions in Algorithmic Languages 1975 (IRIA, 78150 Le Chesnay, 1975) pp. 125–138.

[9] R.E. Milne and C. Strachey, A Theory of Programming Language Semantics (Chapman & Hall, London, 1976).

[10] R. Milner, Fully abstract models of typed λ-calculi, Theor. Comput. Sci. 4 (1977) 1–22.

[11] R. Milner, A theory of type polymorphism in programming, J. Comput. System Sci. 17 (3) (1978) 348–375.

[12] J.H. Morris, Types are not sets, in: Proc. ACM Symp. on Principles of Programming Languages, Boston, IL (1973) pp. 120–124.

[13] J.H. Morris, Towards more flexible type systems, in: B. Robinet (Ed.), Proc. Programming Symposium, Lecture Notes in Computer Science, Vol. 19 (Springer, Berlin, 1974) pp. 377–384.

[14] W.J. van der Poel, C.C. Schaap and G. van der Mey, New arithmetical operators in the theory of combinators, Indag. Math. 42 (1980) 3.

[15] J.C. Reynolds, Towards a theory of type structure, in: Proc. Programming Symposium, Lecture Notes in Computer Science Vol. 19 (Springer, Berlin, 1974) pp. 408–425.

[16] D. Scott, Data types as lattices, SIAM J. Comput. 5 (3) (1976) 522–587.
[17] J.E. Stoy, Denotational Semantics – The Scott–Strachey Approach to Programming Language Theory (MIT Press, Cambridge, MA, 1977).
[18] S.D. Swierstra, Lawine, an experiment in language and machine design, Doctoral Thesis (Twente University of Technology, The Netherlands, 1981).
[19] A. van Wijngaarden et al., Revised report on the algorithmic language ALGOL 68, Acta Informat. 5 (1975) Fasc 1–3.

Algorithmic Languages, de Bakker/van Vliet (eds.)

Abstract Storage Structures

H.B.M. Jonkers

Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

A novel model for the description of storage structures is presented. It is based on the consideration that a storage structure is completely characterized by two things: the collection of its access paths and a relation which indicates whether two access paths access the same substructure. The model, called a 'structure', is abstract in the sense that it is free of low level concepts such as pointers and garbage, while at the same time it is general in that it allows the description of storage structures with arbitrary sharing and circularities. Operations on structures (such as creation and replacement) can be described very naturally in terms of three primitive operations. These primitive operations are defined using a special partial order, which turns the set of all structures into a complete lattice.

1. Introduction

The question what a 'data structure' is has been a point of dispute for several years. Though not all powder smoke has drifted away yet, a beginning of agreement can now be observed. A data structure is a class of objects which is fully characterized by the operations which can be applied to those objects. There are two aspects to this characterization: an external and an internal aspect. The external aspect deals with the question *what* the effect of the operations is. The concept of an 'abstract data type' [11], which is essentially a heterogeneous algebra [3], has been introduced to model this aspect of a data structure. The internal aspect is concerned with the question *how* the effect of the operations is accomplished. This aspect is usually dealt with by choosing a 'representation' for the data structure and 'implementing' each operation in terms of the well-known operations on the representation. It is generally agreed that the internal aspect of a data structure should be hidden ('encapsulated' [16]) to the user.

The above agreement on what a data structure is does not carry over to

an other crucial question: How should data structures be described, or 'specified'? It is important, both to the user and to the implementer, that a specification of a data structure describes only the external aspect of the data structure. The meaning (in the semantical sense) of a specification of a data structure must therefore be an abstract data type. There are basically two ways to specify data structures (or abstract data types, if you like) [12].

The first, and apparently the most attractive, is the axiomatic (or 'implicit') method [6, 7]. In this method the essential properties of the operations are described through axioms. The major advantage of this method is that it is not necessary to commit oneself to a representation for the data structure. There are also two severe drawbacks, however. Apart from very simple data structures, it is very difficult to construct complete and consistent axiomatic specifications. Specifically data structures involving 'dynamic' and 'shared' data, which are frequently encountered in practice, are very hard to specify. Moreover, axiomatic specifications are usually far from easy to comprehend.

The second way of specifying abstract data types is the 'abstract model' approach [1]. In this approach an abstract representation for the data structure to be specified is chosen. The operations of the data structure are then specified in terms of this representation. This method clearly contrasts the axiomatic method as to its advantages and disadvantages. First of all, specifications are more easily constructed. If the possibility of dynamic creation and sharing is already included in the abstract representations chosen, data structures featuring these properties are readily specified. The specifications also tend to be more readable than axiomatic specifications. The salient disadvantage, of course, is the fact that specifications are not representation-independent. If one is not very careful, details of the representation chosen may permeate into the external world and lead to an 'overspecification' of the data structure. (Contrast this with the problem of writing complete axiomatic specifications.)

It is my firm belief that for realistic applications the future lies in the abstract model approach. A precondition is, however, that the problem of representation-dependence is solved satisfactorily. The key to a solution of this problem lies in the observation that the choice of a representation need not depend on efficiency considerations. The only criteria in choosing a representation should be the clarity and naturalness of the specification. This implies first of all that the representations themselves must be free of implementation detail, or in other words, they should be as abstract as

possible. In particular they should not include such things as pointers, fixed size storage cells, etc. On the other hand, the possibility of dynamic creation and sharing should be inherent (otherwise many applications are ruled out). If we had such abstract representations at our disposal, data structures could be specified relatively representation-independent. The sole purpose of the representation would be to increase the comprehensibility of the specification, and not to suggest a certain implementation.

In this paper representations will be described which are believed to satisfy the requirements mentioned above. These representations can be viewed as abstract 'storage structures'. They can be used as the basis for a specification method, which allows the specification of realistic data structures in a comprehensible and unambiguous way, without undue effort and at various levels of abstraction. Their use is not restricted to specification languages, however. It is envisaged that they can successfully be used in definitions of programming languages as well, especially in definitions of those programming languages which feature sharing ('aliasing') and dynamic creation of data.

The representations, which will be called 'structures', are introduced in the next section, together with some related concepts. In Section 3 three primitive operations which can be applied to structures are defined. For their definition a partial order, which turns the set of all structures into a complete lattice, is introduced first.

2. Structures

The purpose of this section is to define the concept of a 'structure'. A structure can be viewed as an abstract 'storage structure', which can be 'accessed' through special keys called 'accessors'. Accessors will be considered as primitive concepts, usually denoted by strings of letters and digits. By repeatedly applying accessors to a structure one can follow an 'access path'.

An *accessor* is a primitive concept.

$\mathscr{A}$ is the set of all accessors.

$\mathscr{A}^*$ is the set of all finite sequences of accessors.

$\mathscr{A}^+$ is the set of all finite nonempty sequences of accessors.

Λ is the empty sequence of accessors.

The sequence $A_1, \ldots, A_n$ of accessors will be denoted as $A_1 \cdots A_n$.

The following definition of the concept of a structure is based on the consideration that a (storage) structure is completely characterized by two things: First, the collection of all of its access paths and second, a relation which indicates whether two access paths access the same 'substructure'. (Notice that the latter is necessarily an equivalence relation.) Taking into account the properties of access paths as well we arrive at the following definition:

A *structure* S is a pair $\langle \mathscr{P}, \equiv \rangle$, where $\mathscr{P} \subset \mathscr{A}^*$ and $\equiv$ is an equivalence relation on $\mathscr{P}$ such that
(1) $\Lambda \in \mathscr{P}$;
(2) $PA \in \mathscr{P} \Rightarrow P \in \mathscr{P}$ $(P \in \mathscr{A}^*, A \in \mathscr{A})$;
(3) $PA \in \mathscr{P} \wedge P \equiv Q \Rightarrow QA \in \mathscr{P} \wedge PA \equiv QA$ $(P, Q \in \mathscr{P}, A \in \mathscr{A})$.

A $P \in \mathscr{P}$ will be called a *path* of S.

An $X \in \mathscr{P} \mid \equiv$, i.e. an equivalence class of $\equiv$, will be called an *object* of S.

$\mathscr{S}$ is the set of all structures.

Property 1 states that the empty sequence of accessors is a path of S (hence $\mathscr{P} \neq \emptyset$). Property 2 implies that any head piece of a path of S is also a path of S. Property 3 states that equivalent paths have equivalent continuations. This property of an equivalence relation is known as 'right-invariance'. The paths of a structure can be viewed as 'names' for the objects which they represent. As will be seen later, the concept of an object as introduced above is closely related to the intuitive concept of an object.

There are three trivial examples of a structure, which will be called the 'empty structure', the 'convergent structure' and the 'divergent structure' respectively:

$\perp = \langle \{\Lambda\}, \{(\Lambda, \Lambda)\} \rangle$ is a structure called the *empty structure.*

$\mathsf{T}_C = \langle \mathscr{A}^*, \mathscr{A}^* \times \mathscr{A}^* \rangle$ is a structure called the *convergent structure.*

$\mathsf{T}_D = \langle \mathscr{A}^*, \{(P, P) \mid P \in \mathscr{A}^*\} \rangle$ is a structure called the *divergent structure.*

Notice that $\perp$ and T_C contain only a single object, while T_D contains an infinite number of objects (i.e. if $\mathscr{A} \neq \emptyset$, which we will from now on assume). Other examples of structures will be discussed below.

Example 1. Let $S = \langle \mathscr{P}, \equiv \rangle$, where

$$\mathscr{P} = \{\Lambda, a, b, ba\},$$

$$\equiv = \{(\Lambda, \Lambda), (a, a), (a, ba), (ba, a), (ba, ba), (b, b)\},$$

then S is a structure containing the following objects:

$$\mathscr{P} \mid \equiv = \{\{\Lambda\}, \{a, ba\}, \{b\}\}.$$

Notice that the paths a and ba are 'aliases' for one and the same object.

Before continuing some notations have to be introduced. First, if $S = \langle \mathscr{P}, \equiv \rangle$ is a structure, then $\mathscr{P}_S$ and $\equiv_S$ will denote $\mathscr{P}$ and $\equiv$ respectively. Second, if X is an object of a structure S and P is a path of S such that $P \in X$, then, if no confusion can arise, $\bar{P}$ will denote X. This convention fits in with the common mathematical practice of denoting equivalence classes by their representatives. Definitions and lemmas which use this notation for objects must be proved to be independent of the choice of the representatives for the objects.

The definition of a structure does not preclude that structures use an infinite number of accessors or have an infinite number of objects. Structures that use only a finite number of accessors and have a finite number of objects constitute an important subclass. The structures in this subclass will be called the 'finite structures'.

Let S be a structure.

The *accessor set* of S is defined as:

$$\{A \in \mathscr{A} \mid \exists P \in \mathscr{P}_S[PA \in \mathscr{P}_S]\}.$$

S is called *finite* iff the accessor set and the set of objects of S are finite; otherwise S is called *infinite*.

The empty structure $\perp$ is an example of a finite structure, and the divergent structure T_D is an example of an infinite structure. The convergent structure T_C is infinite if and only if $\mathscr{A}$ is infinite.

Finite structures can be pictured in a systematic way as follows:

For each object $\bar{P}$
  |Draw a circle $\mathscr{C}_{\bar{P}}$.
For each pair of objects $(\bar{P}, \bar{Q})$
and each accessor A with $PA \in \bar{Q}$
)Draw an arrow labeled by A from $\mathscr{C}_{\bar{P}}$ to $\mathscr{C}_{\bar{Q}}$.
Label $\mathscr{C}_{\bar{\Lambda}}$ by Λ.

Notice that this drawing algorithm is independent of the choice of the paths for the objects and that it would never terminate if applied to an infinite structure. It is easy to see that the picture thus associated to a finite structure is unique.

Example 2. The empty structure $\perp$ has the following picture:

Fig. 1.

If $\mathscr{A} = \{a, b\}$, then the picture of the convergent structure T_C is:

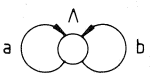


Fig. 2.

If we try the impossible and apply the drawing algorithm to the divergent structure T_D with $\mathscr{A} = \{a, b\}$, then we get:

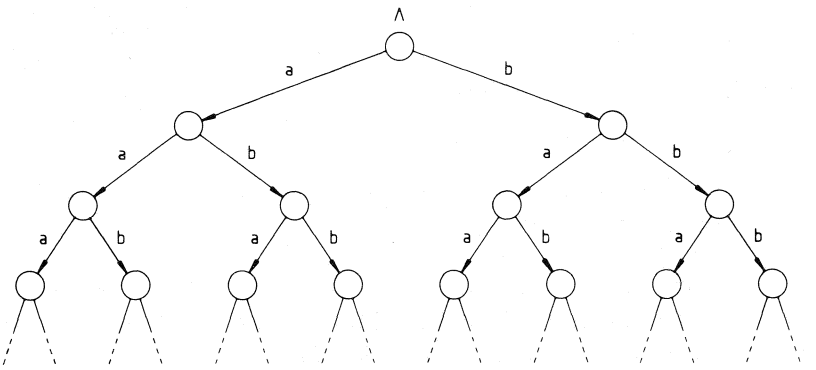


Fig. 3.

The picture of the structure S from Example 1 is:

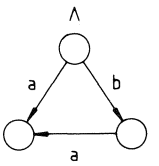


Fig. 4.

The above may raise the question what the difference is between a structure and a rooted graph with labeled edges. At first sight there may not seem to be any difference, yet there is. There are two crucial differences. First, the concept of 'unreachability' is meaningless in a structure. Each object has at least one access path. Second, objects do not have a separate identity. An object simply *is* the collection of its access paths. These two facts will be seen to have a number of important consequences.

An other important observation is that the paths of a structure should *not* be considered as 'pointers': Though a path can be viewed as a name for an object, paths are not objects themselves. Instead, the arrows in the picture of a structure should be regarded as denoting physical inclusion. Since arbitrary kinds of physical inclusion (such as sharing and circularity) can be modeled in a structure, the need to introduce pointers will nowhere arise. The concept of physical inclusion will be made more precise by introducing three relations on the set of objects of a structure:

Let S be a structure.
Let $\bar{P}$ and $\bar{Q}$ be objects of S.

$\bar{P}$ is a *direct component* of $\bar{Q}$ iff there is an $A \in \mathscr{A}$ such that $QA \in \bar{P}$.

$\bar{P}$ is a *component* of $\bar{Q}$ iff there is an $R \in \mathscr{A}^+$ such that $QR \in \bar{P}$.

$\bar{P}$ is *contained* in $\bar{Q}$ iff there is an $R \in \mathscr{A}^*$ such that $QR \in \bar{P}$.

Check that these definitions are independent of the choice of P and Q. The relations 'be a component of' and 'be contained in' are both transitive, while the latter is also reflexive. Neither of them need be an (irreflexive or reflexive) partial order (see Example 3). The meaning of the fact that an object is 'cyclic' can be defined as follows:

|An object of a structure is *cyclic* iff it is a component of itself.

It is easy to see that cyclic objects contain an infinite number of paths.

Example 3. Consider the structure S of Fig. 5.

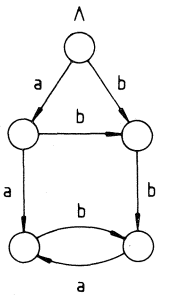


Fig. 5.

The objects of S are:

$$\bar{\Lambda} = \{\Lambda\},$$

$$\bar{a} = \{a\},$$

$$\bar{b} = \{ab, b\},$$

$$\overline{aa} = \{P(ba)^n \mid n \geq 0 \wedge P \in \{aa, abba, bba\}\},$$

$$\overline{bb} = \{P(ab)^n \mid n \geq 0 \wedge P \in \{aab, abb, bb\}\}.$$

The three inclusion relations which are defined between these objects can be described schematically as follows (the plus sign indicates where the relation holds):

$\bar{P}$ is a direct component of $\bar{Q}$:

$\bar{P}$ \ $\bar{Q}$	$\bar{\Lambda}$	$\bar{a}$	$\bar{b}$	$\overline{aa}$	$\overline{bb}$
$\bar{\Lambda}$	−	−	−	−	−
$\bar{a}$	+	−	−	−	−
$\bar{b}$	+	+	−	−	−
$\overline{aa}$	−	+	−	−	+
$\overline{bb}$	−	−	+	+	−

$\bar{P}$ is a component of $\bar{Q}$:

$\bar{P}$ \ $\bar{Q}$	$\bar{\Lambda}$	$\bar{a}$	$\bar{b}$	$\overline{aa}$	$\overline{bb}$
$\bar{\Lambda}$	−	−	−	−	−
$\bar{a}$	+	−	−	−	−
$\bar{b}$	+	+	−	−	−
$\overline{aa}$	+	+	+	+	+
$\overline{bb}$	+	+	+	+	+

$\bar{P}$ is contained in $\bar{Q}$:

$\bar{P}$ \ $\bar{Q}$	$\bar{\Lambda}$	$\bar{a}$	$\bar{b}$	$\overline{aa}$	$\overline{bb}$
$\bar{\Lambda}$	+	−	−	−	−
$\bar{a}$	+	+	−	−	−
$\bar{b}$	+	+	+	−	−
$\overline{aa}$	+	+	+	+	+
$\overline{bb}$	+	+	+	+	+

The relation 'be a component of' is not an irreflexive partial order here, because it is not irreflexive: $\overline{aa}$ is a component of itself. The relation 'be contained in' is not a reflexive partial order because it is not antisymmetric: $\overline{aa}$ is contained in $\overline{bb}$ and $\overline{bb}$ is contained in $\overline{aa}$, but $\overline{aa} \neq \overline{bb}$. This, of course, is caused by the fact that $\overline{aa}$ and $\overline{bb}$ are cyclic objects.

The above example (and especially the expressions for the objects $\overline{aa}$ and $\overline{bb}$) suggests that there is a relation between structures and regular languages. Indeed, the objects of finite structures *are* regular languages:

Lemma 1. *Let S be a finite structure, then each object of S is a regular language over $\mathscr{A}$.*

This can be understood intuitively by considering the picture of a finite structure as the state diagram of a finite state machine and recalling the correspondence between finite state machines and regular languages. A straightforward proof can be obtained by using the fact that each equivalence class of a right-invariant equivalence relation of finite index is a regular language [8]. Another way to prove Lemma 1 is to use the relation

between left-linear grammars and regular languages. (Check that a left-linear grammar, where each nonterminal symbol 'produces' an object, can be associated to each structure.) Due to Lemma 1 a regular expression notation can now be used for the objects of all finite structures.

Example 4. The objects of the structures of Figs. 1, 2, 4 and 5 can be denoted by regular expressions as follows:

Fig. 1: $\bar{\Lambda} = \Lambda$.

Fig. 2: $\bar{\Lambda} = (a+b)^*$.

Fig. 4: $\bar{\Lambda} = \Lambda$, $\bar{a} = a + ba$, $\bar{b} = b$.

Fig. 5: $\bar{\Lambda} = \Lambda$, $\bar{a} = a$, $\bar{b} = ab + b$, $\overline{aa} = (aa + abba + bba)(ba)^*$, $\overline{bb} = (aab + abb + bb)(ab)^*$.

The concept of an object as we introduced it is closely related to the concept of a 'dynamic object', as it is normally conceived in computer science. Dynamic objects are usually considered as 'instances' of 'values'. Two dynamic objects may be instances of the same value and still be different. In mathematical models for dynamic objects this problem is usually solved by associating an 'identity', which is an explicit value, to dynamic objects. As stated before, objects in structures do not have an explicit identity. It is interesting to see how the identity problem for them is solved. The objects in a structure can be viewed as instances of structures (so 'structures' correspond to the 'values' of dynamic objects). This is made more precise by the following definition of the 'structure' of an object:

> Let S be a structure.
> Let $\bar{P}$ be an object of S.
> The *structure* of $\bar{P}$, which will be denoted as $S[\bar{P}]$, is the structure T which is defined as follows:
>
> $$\mathscr{P}_T = \{Q \in \mathscr{A}^* \mid PQ \in \mathscr{P}_S\},$$
>
> $$Q \equiv_T R \Leftrightarrow PQ \equiv_S PR \quad (Q, R \in \mathscr{P}_T).$$

The proof that T is indeed a structure and that T is independent of the choice of P is simple. Two different objects can have the same structure (see Example 5). Hence they can be viewed as instances of that structure.

Example 5. Consider the structure S of Fig. 6.

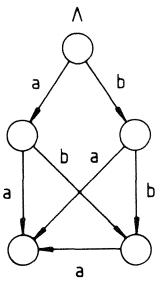


Fig. 6.

In this figure we have (using regular expression notation):

$$\bar{\Lambda} = \Lambda,$$
$$\bar{a} = a,$$
$$\bar{b} = b,$$
$$\overline{aa} = aa + aba + ba + bba,$$
$$\overline{bb} = ab + bb.$$

The structure of $\bar{a}$ is:

$$S[\bar{a}] = \langle \mathscr{P}_0, \equiv_0 \rangle,$$

where

$$\mathscr{P}_0 = \{Q \in \mathscr{A}^* \mid aQ \in \mathscr{P}_S\} = \{\Lambda, a, ba, b\},$$

$$Q \equiv_0 R \Leftrightarrow aQ \equiv_S aR \quad (Q, R \in \mathscr{P}_0),$$

hence

$$\mathscr{P}_0 \mid \equiv_0 = \{\{\Lambda\}, \{a, ba\}, \{b\}\}.$$

The structure of $\bar{b}$ is:

$$S[\bar{b}] = \langle \mathscr{P}_1, \equiv_1 \rangle,$$

where

$$\mathscr{P}_1 = \{Q \in \mathscr{A}^* \mid bQ \in \mathscr{P}_S\} = \{\Lambda, a, ba, b\},$$

$$Q \equiv_1 R \Leftrightarrow bQ \equiv_S bR \quad (Q, R \in \mathscr{P}_1),$$

hence

$$\mathscr{P}_1 \mid \equiv_1 = \{\{\Lambda\}, \{a, ba\}, \{b\}\}.$$

So $\bar{a}$ and $\bar{b}$ have the same structure (the structure of Fig. 4).

Example 6. Consider the structure S of Fig. 7.

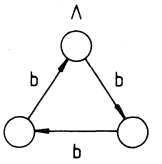


Fig. 7.

All objects have the same structure:

$$S[\bar{\Lambda}] = S[\bar{b}] = S[\overline{b\bar{b}}] = S.$$

3. Operations on Structures

In this section three primitive operations on structures will be defined. They constitute a sufficient set in the sense that all other useful operations on structures can be defined in terms of them. For their definition a special partial order on the set $\mathscr{S}$ of all structures will be introduced first.

The partial order $\sqsubset$ on $\mathscr{S}$ is defined as follows:

$$S \sqsubset T \Leftrightarrow \mathscr{P}_S \subset \mathscr{P}_T \wedge \equiv_S \subset \equiv_T \quad (S, T \in \mathscr{S}).$$

The fact that $\sqsubset$ is indeed a (reflexive) partial order on $\mathscr{S}$ is trivial. In intuitive terms the fact that $S \sqsubset T$ means that all paths of S are also paths of T and that all paths which are 'identified' in S are also identified in T.

Example 7. The structures of Fig. 8 form an ascending sequence.

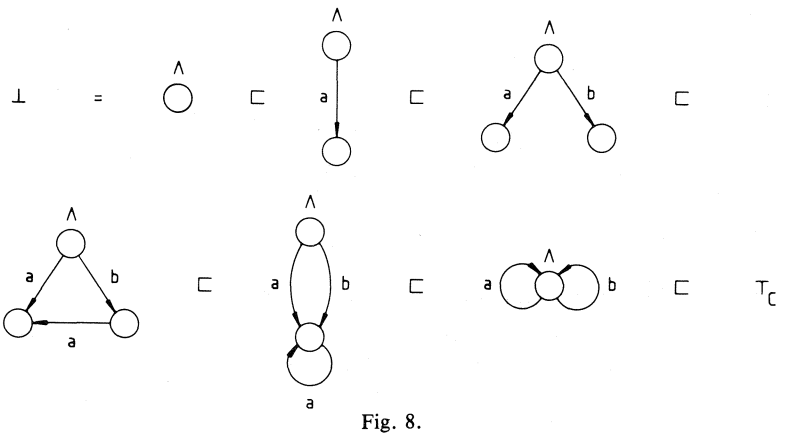


Fig. 8.

Example 8. If we define the partial order $\sqsubset_0$ on $\mathscr{S}$ as:

$$S \sqsubset_0 T \Leftrightarrow S \sqsubset T \wedge \mathscr{P}_S = \mathscr{P}_T \quad (S, T \in \mathscr{S}),$$

then the fact that $S \sqsubset_0 T$ means that S is a 'partial expansion' of T, as illustrated in Fig. 9.

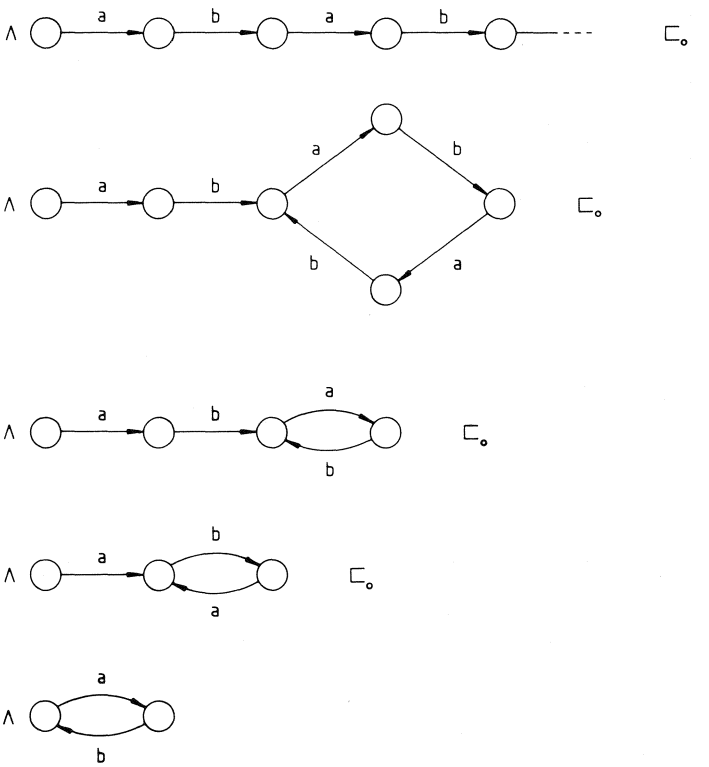


Fig. 9.

Notice that the partial orders $\sqsubset$ and $\sqsubset_0$ are much harder to describe in terms of graphs.

The relation $\sqsubset$ is more than just a partial order: It turns $\mathscr{S}$ into a complete lattice. (A complete lattice is a partially ordered set where each subset has a greatest lower bound.) This is stated in:

Lemma 2. $\langle S, \sqsubset \rangle$ *is a complete lattice.*

The proof of Lemma 2 is simple. First prove that, if S and T are structures, $\langle \mathscr{P}_S \cap \mathscr{P}_T, \equiv_S \cap \equiv_T \rangle$ is also a structure. It is then easy to prove that the greatest lower bound of a set $\mathscr{T}$ of structures is given by $\langle \bigcap_{T \in \mathscr{T}} \mathscr{P}_T, \bigcap_{T \in \mathscr{T}} \equiv_T \rangle$, where $\bigcap_{T \in \mathscr{T}} \mathscr{P}_T = \mathscr{A}^*$ and $\bigcap_{T \in \mathscr{T}} \equiv_T = \mathscr{A}^* \times \mathscr{A}^*$ if $\mathscr{T} = \emptyset$. Notice that the empty structure $\perp$ and the convergent structure T_C are the 'bottom' and 'top' of the complete lattice $\langle \mathscr{S}, \sqsubset \rangle$, i.e. $\perp \sqsubset S \sqsubset \mathsf{T}_C$ for each $S \in \mathscr{S}$. A simple theorem from lattice theory states that apart from a greatest lower bound, each subset also has a least upper bound [2]. The following definitions are therefore in order:

> For each set $\mathscr{T}$ of structures, the structures $\inf \mathscr{T}$ and $\sup \mathscr{T}$ are defined as follows:
>
> $\inf \mathscr{T}$ = greatest lower bound of $\mathscr{T}$ with respect to $\sqsubset$,
>
> $\sup \mathscr{T}$ = least upper bound of $\mathscr{T}$ with respect to $\sqsubset$.

The above will enable us to define the result of operations on structures in terms of inf's and sup's or arbitrary sets of structures without having to worry over the existence of the inf's and sup's.

Example 9. If

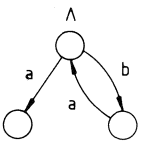
$S =$ 

Fig. 10a.

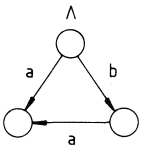
and $T =$ 

Fig. 10b.

,

then

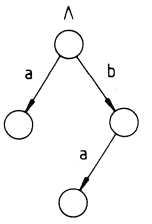
$\inf\{S, T\} =$ 

Fig. 10c.

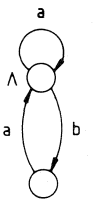
and $\sup\{S, T\} =$ 

Fig. 10d.

Before defining the primitive operations on structures a remark should be made about an other interesting partial order on $\mathscr{S}$. The definition of $\sqsubset$

can be written as:

$$S \sqsubset T \Leftrightarrow \mathscr{P}_S \subset \mathscr{P}_T \wedge \forall P, Q \in \mathscr{P}_S [P \equiv_S Q \Rightarrow P \equiv_T Q] \quad (S, T \in \mathscr{S}).$$

If we reverse the implication sign in this definition we still have a (reflexive) partial order, call it $\sqsubset_1$:

$$S \sqsubset_1 T \Leftrightarrow \mathscr{P}_S \subset \mathscr{P}_T \wedge \forall P, Q \in \mathscr{P}_S [P \equiv_T Q \Rightarrow P \equiv_S Q] \quad (S, T \in \mathscr{S}).$$

Intuitively $S \sqsubset_1 T$ means that all paths of S are also paths of T and that all paths which are 'distinguished' in S are also distinguished in T. The partial order $\sqsubset_1$ has both a bottom (the empty structure $\perp$) and a top (the divergent structure T_D). Yet, in contrast with $\sqsubset$, it does not turn $\mathscr{S}$ into a complete lattice (see Example 10).

Example 10. Consider the structures in Fig. 11.

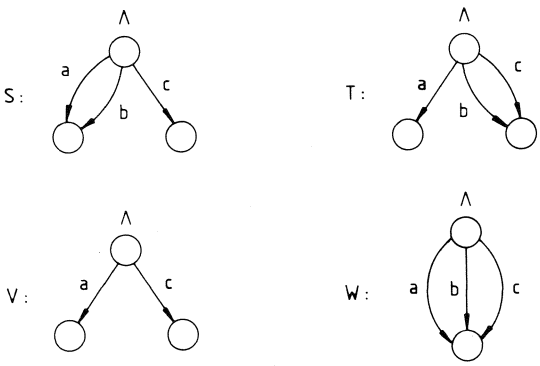


Fig. 11.

Suppose S and T have a greatest lower bound X with respect to $\sqsubset_1$. Since $V \sqsubset_1 S$ and $V \sqsubset_1 T$, we have that $V \sqsubset_1 X$. This implies that $a, c \in \mathscr{P}_X$ and, since $a \not\equiv_V c$, also that $a \not\equiv_X c$. $W \sqsubset_1 S$ and $W \sqsubset_1 T$ imply that $W \sqsubset_1 X$, hence $b \in \mathscr{P}_X$. $X \sqsubset_1 S$ and $a \equiv_S b$ imply that $a \equiv_X b$. Analogously, $X \sqsubset_1 T$ and $b \equiv_T c$ imply that $b \equiv_X c$. Using the transitivity of $\equiv_X$ we get $a \equiv_X c$, which is a contradiction. Hence $\langle \mathscr{S}, \sqsubset_1 \rangle$ is not a complete lattice.

All operations which will be introduced below are considered as partial operators on structures. They may have a number of parameters (usually objects in the structure to which they are applied, or accessors). The result

of applying the operation F with parameters $X_1, \ldots, X_m$ to the structure S will be denoted as $\{S\}F(X_1, \ldots, X_m)$. The notation $F(X_1, \ldots, X_m)$ will be used to denote the (partial) operator $\lambda_{S \in \mathscr{S}}\{S\}F(X_1, \ldots, X_m)$. Concatenation is used to denote functional composition of operators, e.g. $F(X_1, \ldots, X_m)G(Y_1, \ldots, Y_n)$ denotes $\lambda_{S \in \mathscr{S}}\{\{S\}F(X_1, \ldots, X_m)\}G(Y_1, \ldots, Y_n)$.

The first primitive operation on structures which will be introduced amounts to the 'creation' of an object in a structure. The created object has $\perp$ as its structure and is added as a direct component to a given object. The operation, called CRE, has two parameters $\bar{P}$ and A. $\bar{P}$ is an object in the structure S to which CRE is applied and A is an accessor such that PA is not a path of S. The effect of CRE$(\bar{P}, A)$ is pictured in Fig. 12.

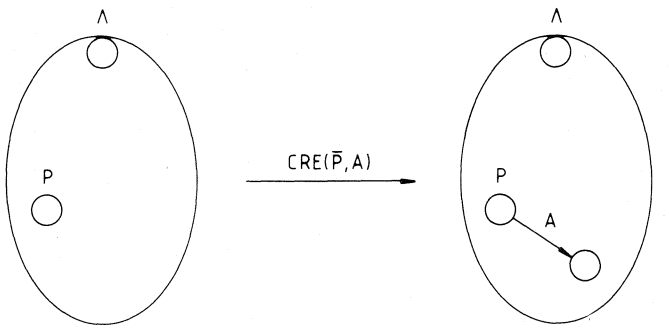


Fig. 12.

The definition of CRE reads:

> Let S be a structure. If $\bar{P}$ is an object of S and $A \in \mathscr{A}$ such that $PA \notin \mathscr{P}_S$, then $\{S\}\text{CRE}(\bar{P}, A)$ is the following structure:
>
> $$\inf\{T \in \mathscr{S} \mid S \sqsubset T \wedge \forall R \in \mathscr{P}_S[R \equiv_S P \Rightarrow RA \in \mathscr{P}_T]\}.$$

It should be clear that CRE$(\bar{P}, A)$ does what Fig. 12 suggests. The fact that 'less' in the partial order $\sqsubset$ implies 'less identification' guarantees that a new object is created and not some old object is taken as the new component of $\bar{P}$.

Example 11. A binary tree can be generated from the empty structure by a sequence of operations such as:

$$\{\perp\}\text{CRE}(\bar{\Lambda}, a)\text{CRE}(\bar{\Lambda}, b)\text{CRE}(\bar{b}, a)\text{CRE}(\overline{ba}, a)\text{CRE}(\overline{ba}, b).$$

The intermediate and final results of this sequence of operations are pictured in Fig. 13.

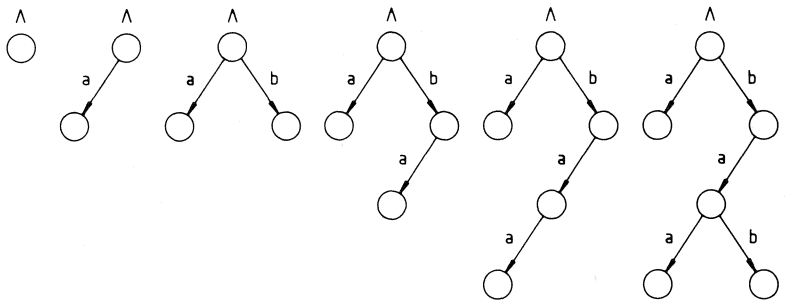


Fig. 13.

The second primitive operation on structures is like CRE, except that it adds an already existing object as a direct component to an object. The operation, called ADD, takes three parameters $\bar{P}$, A and $\bar{Q}$. $\bar{P}$ and $\bar{Q}$ are objects in the structure S to which ADD is applied and A is an accessor such that PA is not a path of S. The effect of ADD$(\bar{P}, A, \bar{Q})$ is pictured in Fig. 14.

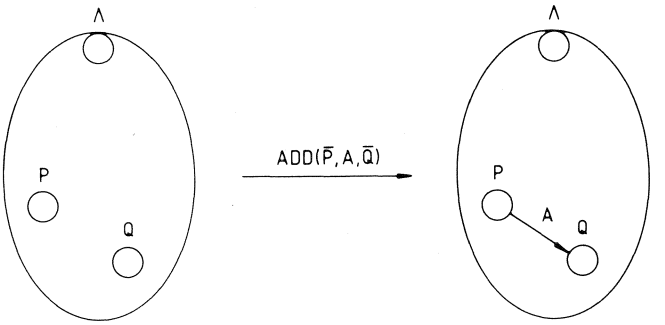


Fig. 14.

The definition of ADD is given below:

> Let S be a structure. If $\bar{P}$ and $\bar{Q}$ are objects of S and $A \in \mathscr{A}$ such that $PA \notin \mathscr{P}_S$, then $\{S\}\text{ADD}(\bar{P}, A, \bar{Q})$ is the following structure:
>
> $$\inf\{T \in \mathscr{S} \mid S \sqsubset T \wedge \forall R \in \mathscr{P}_S[R \equiv_S P \Rightarrow RA \in \mathscr{P}_T \wedge RA \equiv_T Q]\}.$$

The greatest lower bound of the same set of structures as in the definition of CRE is taken here, except that the set is restricted to those structures in which the paths RA with $R \equiv_S P$ and Q are identified. This guarantees that not a new object is created, but that $\bar{Q}$ is added as a new component to $\bar{P}$. Notice that, in contrast with CRE, it is not simple to define ADD without the use of the partial order $\sqsubset$. This is due to the fact that ADD may introduce circularities in a structure.

Example 12. Let S be the structure of Fig. 15, then $\{S\}\text{ADD}(\bar{b}, a, \bar{A})$ is the structure of Fig. 16.

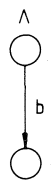


Fig. 15.

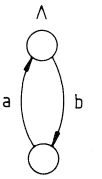


Fig. 16.

The third and final primitive operation can be viewed somehow as the (right) inverse of the other two primitive operations. It amounts to removing a direct component of an object. The operation, called REM, has two parameters $\bar{P}$ and A. $\bar{P}$ is an object in the structure S to which REM is applied and A is an accessor such that PA is a path of S. Fig. 17 pictures the effect of $\text{REM}(\bar{P}, A)$.

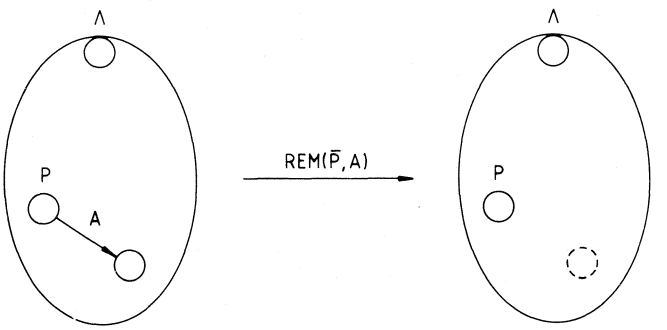


Fig. 17.

The definition of REM is:

> Let S be a structure. If $\bar{P}$ is an object of S and $A \in \mathscr{A}$ such that $PA \in \mathscr{P}_S$, then $\{S\}\mathrm{REM}(\bar{P}, A)$ is the following structure:
>
> $$\sup\{T \in \mathscr{S} \mid T \sqsubset S \wedge \forall R \in \mathscr{P}_S[R \equiv_S P \Rightarrow RA \notin \mathscr{P}_T]\}.$$

Notice that, due to the fact that objects may be shared, $\mathrm{REM}(\bar{P}, A)$ need not remove the object $\overline{PA}$ from a structure. That is why this object is represented by a dotted circle in the right part of Fig. 17. (Strictly speaking the path name P should also be dotted, because the path P (but not the object $\bar{P}$) may be removed from the structure by $\mathrm{REM}(\bar{P}, A)$.) In general, $\mathrm{REM}(\bar{P}, A)$ may reduce the number of objects in a structure by a number varying from zero to all but one (see Example 13).

Example 13. Consider the structure S of Fig. 18.

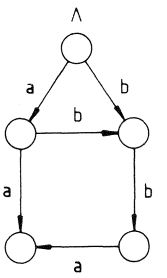


Fig. 18.

The effect of $\mathrm{REM}(\bar{a}, a)$ on S is:

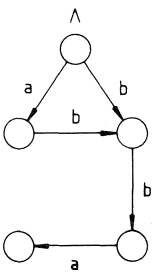


Fig. 19.

Notice: the number of objects has not changed. If REM($\overline{ab}, b$) is applied subsequently to the structure of Fig. 19, we get:

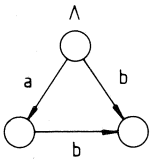


Fig. 20.

Notice: two objects have 'vanished'.

When choosing structures as the basis of the definition of a specification or programming language, the above three primitive operations are sufficient in the sense that all more complex operations can be expressed in terms of them. In order to illustrate this we shall sketch briefly how the meaning of language constructs can be described in terms of the primitive operations. The idea is to represent all values as structures (and their 'instances' as objects of structures). If we consider the variables $X_1, \ldots, X_n$ of an algorithm as accessors, then the 'state' of the algorithm can be represented by a structure as pictured in Fig. 21. In this figure the variables $X_1, \ldots, X_n$ of the algorithm are represented by the paths loc . $X_1, \ldots,$ loc . X_n (dots are used to separate accessors here). The values of the variables are (the structures of) the objects $\overline{\text{loc} \,.\, X_1}, \ldots, \overline{\text{loc} \,.\, X_n}$. Since the latter objects

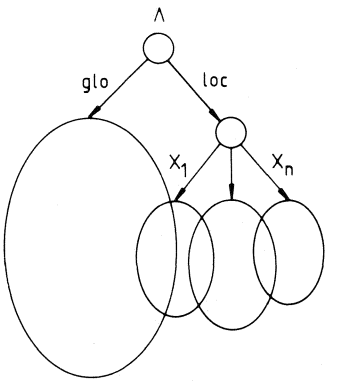


Fig. 21.

may share components, things such as 'aliasing' can readily be described. The component $\overline{\text{loc}}$ of the state constitutes what might be called the 'local environment'. Apart from a local effect an algorithm may also have a global effect ('side effect'). This is modeled by the component $\overline{\text{glo}}$ (the 'global environment') of the state, which is supposed to contain all information global to the algorithm. Since $\overline{\text{glo}}$ and $\overline{\text{loc}}$ may share components, local operations with global side effects can be described very naturally this way.

The meaning of a 'statement' of an algorithm can now be defined as a mapping from states on states, where a state is a structure as in Fig. 21. As an example consider the assignment statement. This statement might have the form "$P.A := Q$", where A is an accessor and PA and Q are paths within the local environment. (The statement should be read as "replace the A-component of $\bar{P}$ by $\bar{Q}$".) The meaning of the assignment statement could be defined as:

$$\mathcal{M}(P.A := Q) = \text{ADD}(\bar{\Lambda}, p, \overline{\text{loc} \,.\, P})\text{ADD}(\bar{\Lambda}, q, \overline{\text{loc} \,.\, Q})$$
$$\text{REM}(\bar{p}, A)\text{ADD}(\bar{p}, A, \bar{q})$$
$$\text{REM}(\bar{\Lambda}, p)\text{REM}(\bar{\Lambda}, q).$$

Notice that the following definition would not be correct:

$$\mathcal{M}(P.A := Q) = \text{REM}(\overline{\text{loc} \,.\, P}, A)\text{ADD}(\overline{\text{loc} \,.\, P}, A, \overline{\text{loc} \,.\, Q}).$$

The reason is that after $\text{REM}(\overline{\text{loc} \,.\, P}, A)$ both the object $\overline{\text{loc} \,.\, Q}$ and the path loc . P need no longer exist. The meaning of language constructs other than the assignment statement can be described in a similar way. For more details about this the reader is referred to [9].

4. Conclusion

In this paper a novel method of characterizing storage structures was discussed. The concept of a 'structure' was introduced, which is basically a simple mathematical model of the access properties of a storage structure. Using this model storage structures with arbitrary sharing and circularities can be characterized without the need to introduce pointers. Creation and replacement become very natural operations which cannot produce any

'garbage', since the concept of unreachability is nonexistent in a structure. Due to the fact that structures are general and yet free of such low level concepts as pointers and garbage, they lend themselves very well as the basis of definitions of realistic specification and programming languages. This is illustrated in [9], in which a specification language for abstract data types is discussed, which is used (in a somewhat informal way) in [10].

The concept of a structure as defined in this paper is believed to characterize storage structures in a way more abstract than other methods. In order to support this assertion let us give a short comparison of structures with some of these other methods. 'Vienna objects' [14] are basically trees with labeled branches. Sharing and circularity can only be modeled by introducing a pointer concept. This is done by allowing 'composite selectors' (which correspond to 'paths') to be used as objects. 'Graphs' [13] were already discussed in Section 2. Graphs are easily seen to be less abstract than structures, because each structure corresponds to many graphs. Also, the unnatural choice of an already existing node as the new node when creating a node in a graph is not necessary in a structure. 'Relational objects' [5] are set-theoretic models of storage structures. They are built from atomic values using set and tuple constructors. Relational objects are more general than graphs (each graph can be described as a relational object), but they inherit many of the disadvantages of graphs. E.g., sharing can only be modeled by representing objects in some way as primitive values (which correspond to the nodes of a graph). The programming language SETL [4] even has a special atomic data type for this purpose. A more comprehensive comparison of structures with other methods of characterizing storage structures can be found in [9].

References

[1] V.A. Bĕrzinš, Abstract model specifications for data abstractions, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA (1979).

[2] G. Birkhoff, Lattice Theory, American Mathematical Society Colloquium Publications, Vol. XXV (Am. Math. Soc., Providence, RI, 1967).

[3] G. Birkhoff and J.D. Lipson, Heterogeneous algebras, J. Combin. Theory 8 (A) (1970) 115–133.

[4] R.B.K. Dewar, The SETL programming language, to appear.

[5] J. Earley, Relational level data structures for programming languages, Acta Inform. 2 (1973) 293–309.

[6] J.A. Goguen, J.W. Thatcher and E.G. Wagner, An initial algebra approach to the specification, correctness and implementation of abstract data types, in: R. Yeh (Ed.), Current Trends in Programming Methodology, Vol. IV (Prentice Hall, Englewood Cliffs, NJ, 1978).

[7] J.V. Guttag and J.J. Horning, The algebraic specification of abstract data types, Acta Inform. 10 (1978) 27–52.

[8] J.E. Hopcroft and J.D. Ullman, Introduction to Automata Theory, Languages, and Computation (Addison-Wesley Publishing Company, Reading, MA, 1979).

[9] H.B.M. Jonkers, Abstract storage structures and the specification of abstract data types, Mathematical Centre, Amsterdam, to appear.

[10] H.B.M. Jonkers, Abstraction, specification and implementation techniques, with an application to garbage collection, Ph.D. Thesis, Mathematical Centre, Amsterdam, to appear.

[11] B. Liskov and S. Zilles, Programming with abstract data types, in: Proc. Symp. on Very High Level Languages, SIGPLAN Notices 9 (4) (1974) 50–59.

[12] B. Liskov and S. Zilles, Specification techniques for data abstractions, IEEE Trans. Software Engrg. SE-1 (1975) 7–19.

[13] M.E. Majster, Extended directed graphs, a formalism for structured data and data structures, Acta Inform. 8 (1977) 37–59.

[14] P. Wegner, The Vienna definition language, Comput. Surveys 4 (1972) 5–63.

[15] R. Yeh (Ed.), Current Trends in Programming Methodology, Vol. IV (Prentice Hall, Englewood Cliffs, NJ, 1978).

[16] S. Zilles, Procedural encapsulation: A linguistic protection technique, in Proc. ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices 8 (9) (1973) 140–146.

Algorithmic Languages, de Bakker/van Vliet (eds.)

Invited Address

The Essence of ALGOL*

John C. Reynolds

Syracuse University, Syracuse, NY, U.S.A.

Although ALGOL 60 has been uniquely influential in programming language design, its descendents have been significantly different than their prototype. In this paper, we enumerate the principles that we believe embody the essence of ALGOL, describe a model that satisfies these principles, and illustrate this model with a language that, while more uniform and general, retains the character of ALGOL.

1. The Influence of Models of ALGOL

Among programming languages, ALGOL 60 [1] has been uniquely influential in the theory and practice of language design. It has inspired a variety of models which have in turn inspired a multitude of languages. Yet, almost without exception, the character of these languages has been quite different than that of ALGOL itself. To some extent, the models failed to capture the essence of ALGOL and gave rise to languages that reflected that failure.

On main line of development centered around the work of Landin, who devised an abstract language of applicative expressions [2] and showed that ALGOL could be translated into this language [3]. This work was influenced by McCarthy's LISP [4] and probably by unpublished ideas of C. Strachey; in turn it led to more elaborate models such as those of the Vienna group [5]. Later many of its basic ideas, often considerably transformed, reappeared in the denotational semantics of Scott and Strachey [6].

In [2], after giving a functional description of applicative expressions,

---

• Work supported by National Science Foundation Grant MCS-8017577 and U.S. Army Contract DAAK80-80-C-0529.

Landin presented a state-transition machine, called the SECD machine, for their evaluation. Then in [3] he extended applicative expressions to 'imperative applicative expressions' by introducing assignment and a label-like mechanism called the *J*-operator. The imperative applicative expressions were not described functionally, but by an extension of the SECD machine called the 'sharing machine'. In later models, such as that of the Vienna group, sharing was elucidated by introducting a state component usually called the 'store' or 'memory'.

For our present concerns, three aspects of Landin's model are especially significant. First, the variety of values that can be assigned to variables is the same as the variety that can be denoted by identifiers or passed as parameters. Landin does not emphasize this fact; it is simply a direct consequence of the typelessness of imperative applicative expressions. Second, no distinction is made between assignments to variables and assignments to locations embedded within data structures. Again, this is inherent in the nature of the model, in which variables themselves are locations embedded within the data structures of the sharing machine.

Finally, since operands are evaluated before operators, the basic method of parameter passing is call by value, and call by name is described in terms of call by value using parameterless functions (in contrast to the ALGOL 60 report [1], where call by value is described in terms of call by name using appropriately initialized local variables). This approach apparently stems from the view that undefined values do not 'exist', so that a function cannot map an undefined value into a defined value (as in LISP, where the conditional must be regarded as a special form rather than a function). This is in contrast with the more recent view of Scott that an undefined value is as legitimate as any other; its only peculiarity is being least in a partial ordering that must be respected by functions.

Directly or indirectly, Landin's model was the basis for a number of programming languages, including his own ISWIM [7], Evans and Wosencraft's PAL [8], and my GEDANKEN [9]. Less obviously, the model influenced ALGOL 68 [10], despite the significant distinction that this language is highly typed. All of these languages inherited from the model the characteristics described above: Anything that can be passed as a parameter can be assigned to a variable, there is no fundamental distinction between assignments to variables and to components of data structures, and call by value is either the only or the basic mode of parameter transmission.

As a consequence, all of these languages are significantly different from ALGOL; in certain respects they are closer to the spirit of LISP. They are all subject to the criticism of references made by Hoare [11]. (Strictly speaking, only ALGOL68 and GEDANKEN use the reference concept, but Hoare's criticism is equally applicable to the sharing or *L*-value approach used in ISWIM and PAL.)

Moreover, except for ALGOL 68, none of these languages obey a stack discipline. It would require a clever compiler to make any use of a stack during program execution, and even then it would be difficult for a programmer to foresee when such use would occur.

In ALGOL 68, a stack discipline is obtained by imposing the restriction that a procedure value becomes undefined upon exit from any block in which a global variable of the procedure is declared. However, this restriction is imposed for the specific purpose of rescuing the stack; a stack discipline is not a natural consequence of the basic character of the language.

Another line of development stemming from ALGOL 60 has led to languages such as PASCAL [121] and its descendents, e.g. EUCLID [13], MESA [14], and ADA [15], which are significantly lower-level than ALGOL. Each of these languages seriously restricts the block or procedure mechanism of ALGOL by eliminating features such as call by name, dynamic arrays, or procedure parameters.

I am not familiar enough with the history of these languages to do more than speculate about the influence of models. However, a desire to be 'closer to the machine' than ALGOL 60 seems evident from the abandonment of features requiring inefficient or 'clever' implementations. In this respect, implementations themselves can be thought of as models influencing language design.

In addition, the influence of program-proving formalisms, particularly the work of Hoare [16], is clear. An axiomatic definition of PASCAL [17] seems to have influenced that language, and the axiomatization of EUCLID [13] was a major goal of its design.

Since Hoare's treatment of procedures [18] does not encompass call by name, procedure parameters, or aliasing, it may account for the weakening of the procedure mechanism in some of these languages. Certainly the view of procedures given by this kind of axiomatization is profoundly different than the copy rule.

2. Some Principles

The preceding somewhat biased history is intended to motivate a new model that I believe captures the essence of ALGOL and can be used to develop a more uniform and general 'Idealized ALGOL' retaining the character of its prototype. Although its genesis lies in the definition of the simple imperative language given in [19], the crux of the model is a treatment of procedures and block structure developed by F.J. Oles and myself.

This paper only describes the basic nature of the model, and it avoids the mathematical sophistication, involving universal algebra and category theory, that is needed to reveal its elegance. A complete and mathematically literate description is given in [20).

It should also be emphasized that the description of 'Idealized ALGOL' in this paper is extremely tentative and only intended to illustrate the model.

Before delving into the details, we state the principles that we believe embody the essence of ALGOL:

(1) ALGOL is obtained from the simple imperative language by imposing a procedure mechanism based on a fully typed, call-by-name lambda calculus.

In other words, Landin was right in perceiving the lambda calculus underlying ALGOL, but wrong in embracing call by value rather than call by name.

The qualification 'fully typed' indicates agreement with Van Wijngaarden that all type errors should be syntactic errors, and that this goal requires a syntax with an infinite number of phrase classes, themselves possessing grammatical or (more abstractly) algebraic structure. (I believe that this characteristic will be the most influential and long lasting aspect of ALGOL 68.) The failure of this property for ALGOL 60 is a design mistake, not part of its essence.

When carried to the extreme, this principle suggests that the lambda calculus is the source of all identifier binding. More precisely, except for syntactic sugar (language constructs than can be defined as abbreviations in terms of more basic constructs, as the **for** statement is defined in the ALGOL60 Report), the only binding mechanism should be the lambda expression.

(2) There are two fundamentally different kinds of type: *data types,* each of which denotes a set of values appropriate for certain variables and expressions, and *phrase types,* each of which denotes a set of meanings appropriate for certain identifiers and phrases.

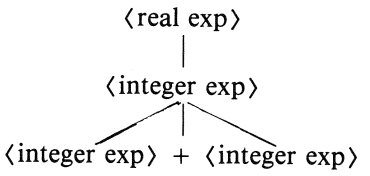
This syntactic distinction reflects that fact that in ALGOL values (which can be assigned to variables) are inherently different from meanings (which can be denoted by identifiers and phrases, and passed as parameters). Thus ALGOL-like languages contradict the principle of completeness [9].

Moreover, in ALGOL itself data types are limited to unstructured types such as *integer* or *Boolean,* while structuring mechanisms such as procedures and arrays are only applicable to phrase types.

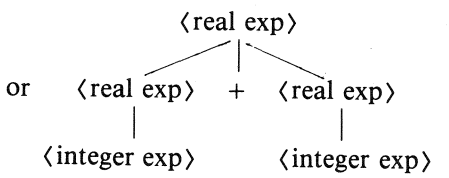
(3) The order of evaluation for parts of expressions, and of implicit conversions between data or phrase types, should be indeterminate, but the meaning of the language, at an appropriate level of abstraction, should be independent of this indeterminacy.

By 'appropriate' we mean a level of abstraction where overflow and roundoff are ignored and termination with an error message is regarded as equivalent to nontermination. This principle prohibits expressions with side effects such as assignments to nonlocal variables or jumps to nonlocal labels, but not expressions that cause error stops.

If types are described grammatically, the indeterminacy of implicit conversions will cause ambiguity. For example, in a context calling for a real expression, 3 + 4 might be parsed as either



or



Except for overflow and (with unfortunate hardware) roundoff, both parses should have the same meaning.

(4) Facilities such as procedure definition, recursion, and conditional and case constructions should be uniformly applicable to all phrase types.

This principle leads to procedures whose calls are procedures, but under a call-by-name regime such procedures do not violate a stack discipline in the way that, for example, function-returning functions in GEDANKEN violate such a discipline. More interestingly, this principle leads to conditional variables and procedures whose calls are variables; indeed arrays can be regarded as a special case of the latter.

(57 The language should obey a stack discipline, and its definition should make this discipline obvious.

Almost any form of language definition can be divided into primary and secondary parts, e.g. Table 1.

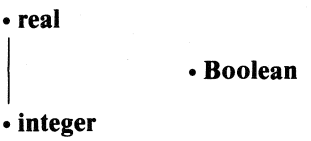
Table 1

	Primary	Secondary
Denotational semantics	Domain equations	Semantic equations
Algebraic semantics	Definition of the target algebra carrier	Definition of the target algebra operations
Operational semantics	Definition of the set of states of the interpreter	Definition of the state-transition function

By "should make the stack discipline obvious" we mean that the stack discipline should be a consequence of the primary part of the language definition. Specifically, the primary part should show that the execution of a statement never changes the 'shape' of the store, i.e. the aspect of the store that reflects storage allocation.

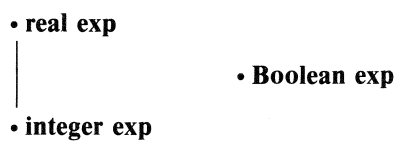
3. Data Types and Expressions

To stay close to ALGOL 60, we take {**integer, real, Boolean**} as the set of data types. To introduce an implicit conversion from **integer** to **real**, we define the partial ordering



and say that τ is a subtype of τ' when $\tau \leq \tau'$.

For each data type τ there is a phrase type τ **exp**(ression), and these phrase types inherit the subtype relation of the data types:



When $\theta \leq \theta'$ we again say that θ is a subtype of θ', now meaning that any phrase of type θ can appear in any context requiring a phrase type θ', e.g. any integer expression can occur in any context requiring a real expression.

A **type assignment** is a function from some finite set of identifiers to phrase types. To describe the syntax of our language we will use phrase class names of the form $\langle \theta, \pi \rangle$, where θ is a phrase type and π is a type assignment, to denote the set of phrases P such that

(1) The identifiers occurring free in P belong to the domain of π.

(2) When its free identifiers are given the phrase types indicated by π, P has phrase type θ.

We will describe syntax by production schemas (in the spirit though not the notation of Van Wijngaarden) in which the metavariables τ, θ, π, and ι range over data types, phrase types, type assignments, and identifiers respectively. A fragment of an appropriate syntax for expressions is

$$\langle \theta, \pi \rangle ::= \langle \theta', \pi \rangle \quad \text{when } \theta' \leq \theta,$$

$$\langle \theta, \pi \rangle ::= \iota \quad \text{when } \iota \in \mathrm{dom}(\pi) \text{ and } \pi(\iota) = \theta,$$

$$\langle \text{integer exp}, \pi \rangle ::= 0 \mid 1 \mid \langle \text{integer exp}, \pi \rangle + \langle \text{integer exp}, \pi \rangle,$$

$$\langle \text{real exp}, \pi \rangle ::= 0.5 \mid \langle \text{real exp}, \pi \rangle + \langle \text{real exp}, \pi \rangle,$$

$$\langle \text{Boolean exp}, \pi \rangle ::= \textbf{true} \mid \textbf{false} \mid \langle \tau \text{ exp}, \pi \rangle = \langle \tau \text{ exp}, \pi \rangle$$
$$\mid \langle \text{Boolean exp}, \pi \rangle \textbf{ and } \langle \text{Boolean exp}, \pi \rangle.$$

(Here $\mathrm{dom}(\pi)$ denotes the domain of the type assignment π.)

This is an abstract syntax to the extent that precedence considerations are ignored. One could 'concretize' it by adding parentheses around the right side of each production, but a realistic concrete syntax would require far fewer parentheses. In fact, we will use fewer parentheses in our examples of programs, trusting the reader's intuition to supply the missing ones sensibly.

In the first two production schemas θ ranges over all phrase types, not just the types of expressions introduced so far. The first schema shows the purpose of the subtype relationship. The second shows that an identifier assigned some phrase type can always be used as a phrase of that type.

In accordance with Principle 3, the syntax is ambiguous (aside from parenthesization considerations), but this ambiguity must not result in ambiguous meanings. An appropriate method for insuring this requirement is described in [19]; it requires that the syntax possess a property that might be called 'minimal typing':

For any phrase P and type assignment π, if there is any θ such that $P \in \langle \theta, \pi \rangle$, then there is a *minimal* θ_0 such that $P \in \langle \theta, \pi \rangle$ if and only if $\theta_0 \leq \theta$.

(When a phrase class name is used as a set it stands for the set of all phrases that can be derived from that phrase class name.)

To prohibit expressions with side effects, we will forbid any occurrence of statements within expressions (except in vacuous contexts such as parameters of constant procedures) and insist that the bodies of function procedures be expressions. Actually, this is unnecessarily Draconian; one would like to permit block expressions, as in ALGOL W [22], but restricted to avoid side effects. However, this topic is beyond the scope of this paper.

4. The Simple Imperative Language

The next step is to introduce variables for each data type. But here we encounter a surprising complication. As a consequence of Principle 4, we want to have conditional variables. For example, when n is an integer variable and x is a real variable, we should be able to write **if** p **then** n **else** x on either side of an assignment statement. When used on the right side, this phrase must be considered as a real expression, since when p is false it can produce a noninteger value. But when used on the left side, it must be considered an integer variable, since when p is true it cannot accept a noninteger value. Thus there are variables that accept a different data type than they produce.

The first step in dealing with this situation is to realize that, in addition to variables, which accept and produce values, and expressions, which only

produce values, it is natural to introduce phrases called *acceptors,* which only accept values. Thus for each data type τ, we will have the phrase type τ **acc**(eptor). The subtype relation for acceptors is the dual of the subtype relation for data types. For example, since **integer** is a subtype of **real**, integer values can be implicitly converted into real values, so that a real acceptor can be used in any context requiring an integer acceptor, i.e. **real acc** $\leq$ **integer acc.**

The second step is to categorize variables separately by the data types that they accept and produce. Thus for each pair of data types τ_1 and τ_2, we have the phrase type τ_1 (accepting) τ_2 (producing) **var**(iable), which is a subtype of $\tau_1'\tau_2'$ **var** when τ_1 **acc** is a subtype of τ_1' **acc** and τ_2 **exp** is a subtype of τ_2' **exp**, i.e. when the data types satisfy $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$.
subtype of τ_2' **exp**, i.e. when the data types satisfy $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$.

The case construction for variables raises the same difficulty as the conditional. But a further problem arises if the empty construction **case** *n* **of** ( ) is permitted. Of course, it would not be unreasonable to prohibit this construction, but it is consistent to view it as a phrase whose phrase type **univ**(ersal) is a subtype of all phrase types. All phrases of this type have the meaning 'undefined', which implicitly converts into the undefined element of the domain of meanings of any other phrase type.

The only other phrase type needed to describe the simple imperative language is **comm**(and). (Throughout this paper, we will speak of commands rather than statements.) In summary, the phrase types of the simple imperative language, which we will call *primitive* phrase types, are

τ **exp**	**comm**
τ **acc**	**univ**
$\tau_1\tau_2$ **var**	

and the subtype relation is the least partial ordering such that

$\tau \leq \tau'$ implies τ **exp** $\leq \tau'$ **exp**,

$\tau' \leq \tau$ implies τ **acc** $\leq \tau'$ **acc**,

$\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$ implies $\tau_1\tau_2$ **var** $\leq \tau_1'\tau_2'$ **var**

$\tau_1\tau_2$ **var** $\leq \tau_1$ **acc**,

$\tau_1\tau_2$ **var** $\leq \tau_2$ **exp**,

univ $\leq \theta$.

An appropriate syntax is:

$$\langle \text{comm}, \pi \rangle ::= \textbf{skip} \mid \langle \tau \text{ acc}, \pi \rangle := \langle \tau \text{ exp}, \pi \rangle \mid \langle \text{comm}, \pi \rangle ; \langle \text{comm}, \pi \rangle$$
$$\mid \textbf{while } \langle \text{Boolean exp}, \pi \rangle \textbf{ do } \langle \text{comm}, \pi \rangle,$$
$$\langle \theta, \pi \rangle ::= \textbf{if } \langle \text{Boolean exp}, \pi \rangle \textbf{ then } \langle \theta, \pi \rangle \textbf{ else } \langle \theta, \pi \rangle$$
$$\mid \textbf{case } \langle \text{integer exp}, \pi \rangle \textbf{ of } (\langle \theta, \pi \rangle, \ldots, \langle \theta, \pi \rangle).$$

In the last two lines, θ stands for any phrase type, including the nonprimitive types to be introduced later. The minimal typing property holds for these productions if, in the partial ordering of phrase types, every finite set with an upper bound has a least upper bound. In fact, the achievement of this property for primitive phrase types was the real goal of the arguments about acceptors, variables, and **univ** at the beginning of this section.

For mathematical simplicity, it is tempting to make the partial ordering of phrase types into a lattice by introducing a phrase type **ns** (nonsense), of which all phrase types are subtypes. However, although a nonsense type simplifies certain theoretical techniques, as in [19], it is not germaine to the purposes of this paper.

A complete semantic definition of the simple imperative language is given in [19]; here we will only delineate the basic nature of such a definition by giving its domain equations. For each phrase type θ, there is a domain of meanings D_θ, and for each type assignment π, there is a domain of environments Env_π, which is the product $\prod_{\iota \in \text{dom}(\pi)} D_{\pi(\iota)}$ of the domains for the type of each identifier in $\text{dom}(\pi)$. Then for each phrase class $\langle \theta, \pi \rangle$ there is a semantic function from phrases to environments to meanings, i.e. $\mu_{\theta,\pi} \in \langle \theta, \pi \rangle \to (\text{Env}_\pi \to D_\theta)$.

For direct semantics $D_{\textbf{comm}}$ is a domain of state transitions, i.e. $S \to S_\perp$, where S is the set of states of the store (hereafter simply called states), and $S_\perp$ indicates the formation of a domain by adding an undefined element $\perp$ (denoting nontermination) to the set S. Similarly $D_{\tau\,\textbf{exp}}$ is $S \to (V_\tau)_\perp$, where $V_{\textbf{integer}}$ is the set of integers, $V_{\textbf{real}}$ is the set of real numbers, and $V_{\textbf{Boolean}}$ is the set $\{\textbf{true}, \textbf{false}\}$.

There are two ways of treating variables. The more conventional is to say that, for each data type, a state has a component mapping an appropriate set of 'L-values' (or 'names' or 'references' or 'abstract addresses') into values of that data type, i.e.

$$S = (L_{\mathbf{integer}} \to V_{\mathbf{integer}}) \times (L_{\mathbf{real}} \to V_{\mathbf{real}}) \times (L_{\mathbf{Boolean}} \to V_{\mathbf{Boolean}}).$$

Then $D_{\tau\,\mathbf{var}}$ is $S \to (L_\tau)_\perp$.

A preferable approach, however, avoids any commitment to a notion such as L-values or references, and more clearly reveals the relationship among variables, acceptors, and expressions. One regards the meaning of an acceptor as a function mapping each value into the state transformation caused by assigning that value to the acceptor, so that $D_{\tau\,\mathbf{acc}} = V_\tau \to (S \to S_\perp)$. Then the meaning of a variable is a pair of functions describing its meanings in its dual roles of acceptor and expression, so that $D_{\tau_1\tau_2\,\mathbf{var}} = D_{\tau_1\,\mathbf{acc}} \times D_{\tau_2\,\mathbf{exp}}$. The implicit conversion functions from variables to acceptors and expressions are the projections from $D_{\tau_1\tau_2\,\mathbf{var}}$ to $D_{\tau_1\,\mathbf{acc}}$ and $D_{\tau_2\,\mathbf{exp}}$.

These two views of variables provide a nice example of the way in which formal definition can influence language design. As long as we do not impose any structure involving L-values or references upon states, there is no danger of defining anyting, such as call by reference, that involves these concepts. On the other hand, the more abstract approach opens the door to features, such as doublets in POP-2 [21] or implicit 'references' in GEDANKEN [9], that define a variable by giving arbitrary procedures for accepting and producing values.

In fact, the more abstract treatment of variables makes no commitment at all to the structure of states; S is a parameter of the semantics that can sensibly stand for any set at all. To emphasize this generality, we make S an explicit argument of D_θ and Env_π, and regard the semantics of a phrase as a family of functions, indexed by S, from environments to meanings:

$$\text{if } P \in \langle \theta, \pi \rangle, \text{ then } \mu_{\theta,\pi}(P)(S) \in \mathrm{Env}_\pi(S) \to D_\theta(S),$$

where

$$\mathrm{Env}_\pi(S) = \prod_{\iota \in \mathrm{dom}(\pi)} D_{\pi(\iota)}(S),$$

$$D_{\mathbf{comm}}(S) = S \to S_\perp,$$

$$D_{\tau\,\mathbf{exp}}(S) = S \to (V_\tau)_\perp,$$

$$D_{\tau\,\mathbf{acc}}(S) = V_\tau \to D_{\mathbf{comm}}(S),$$

$$D_{\tau_1\tau_2\,\mathbf{var}}(S) = D_{\tau_1\,\mathbf{acc}}(S) \times D_{\tau_2\,\mathbf{exp}}(S).$$

However, although the semantics of a phrase is a family of environment-

to-meaning functions, the members of this family must bear a close relationship to one another. Roughly speaking, whenever a state set S can be 'expanded' into another state set S', the semantics of a phrase for S must be related to its semantics for S'.

To make the notion of expansion precise, we first introduce some useful notation:

(*Identity and composition of functions*) We write I_S for the identity function on S, and $\cdot$ for functional composition in diagrammatic order (so that $(f \cdot g)(x) = g(f(x))$).

(*Strict extension*) When $f \in S \to S'_\perp$, we write $f_\circ$ for the $\perp$-preserving extension of f to $S_\perp \to S'_\perp$. When $f \in S \to S'$, we write $f_\perp$ for the $\perp$-preserving extension of f to $S_\perp \to S'_\perp$.

(*Identity and composition of state-transition functions*) We write J_S for the identity injection from S to $S_\perp$. When $f, g \in S \to S_\perp$, we write $f * g$ for $f \cdot (g_\circ) \in S \to S_\perp$.

(*Diagonalization*) We write D_S for the continuous function from $S \to S \to S_\perp$ to $S \to S_\perp$ such that $D_S(h)(\sigma) = h(\sigma)(\sigma)$.

In the last definition (and later in this paper) we assume that $\to$ is right associative and that function (and procedure) application is left associative.

Then we define an *expansion* of S to S' to be a pair $\langle g, G \rangle$ of functions $g \in S' \to S$, $G \in (S \to S_\perp) \to (S' \to S'_\perp)$ such that

(1) G is continuous and $\perp$-preserving.

(2) $G(J_S) = J_{S'}$.

(3) When $f_1, f_2 \in S \to S_\perp$, $G(f_1 * f_2) = G(f_1) * G(f_2)$.

(4) When $f \in S \to S_\perp$, $g \cdot f = G(f) \cdot (g_\perp)$.

(5) When $h \in S \to S \to S_\perp$, $G(D_S(h)) = D_{S'}(g \cdot h \cdot G)$.

Intuitively, g maps each state in S' into the member of S that is 'embedded' within it, while G maps each state-transition function in $S \to S_\perp$ into the state-transition function in $S' \to S'_\perp$ that 'simulates' it.

More precisely, an expansion of S to S' induces, for each phrase type θ, a function in $D_\theta(S) \to D_\theta(S')$ that maps meanings appropriate to S into meanings appropriate to S'. If we write $D_\theta(\langle g, G \rangle)$ for the function in $D_\theta(S) \to D_\theta(S')$ induced by $\langle g, G \rangle$, then

$$D_{\mathbf{comm}}(\langle g, G \rangle) = G,$$

$$D_{\tau\,\mathbf{exp}}(\langle g, G \rangle)(e \in S \to (V_\tau)) = g \cdot e,$$

$$D_{\tau\,\mathbf{acc}}(\langle g, G\rangle)(a \in V_\tau \to D_{\mathbf{comm}}(S)) = a \cdot G,$$

$$D_{\tau_1\tau_2\,\mathbf{var}}(\langle g, G\rangle)(\langle a, e\rangle) = D_{\tau_1\,\mathbf{acc}}(\langle g, G\rangle)(a), D_{\tau_2\,\mathbf{exp}}(\langle g, G\rangle)(e)\rangle.$$

By pointwise extension, an expansion of S to S' induces, for each type assignment π, a function in $\mathrm{Env}_\pi(S) \to \mathrm{Env}_\pi(S')$ that maps environments appropriate to S into environments appropriate to S'. If we write $\mathrm{Env}_\pi(\langle g, G\rangle)$ for the function in $\mathrm{Env}_\pi(S) \to \mathrm{Env}_\pi(S')$ induced by $\langle g, G\rangle$, then

$$\mathrm{Env}_\pi(\langle g, G\rangle)(\eta \in \mathrm{Env}_\pi(S))(\iota) = D_{\pi(\iota)}(\langle g, G\rangle)(\eta(\iota)).$$

We can now state the fundamental relationship between the semantics of a phrase for different state sets: If P is a phrase in $\langle \theta, \pi\rangle$ and $(\langle g, G\rangle)$ is an expansion of S to S', then

$$\mu_{\theta,\pi}(P)(S) \cdot D_\theta(\langle g, G\rangle) = \mathrm{Env}_\pi(\langle g, G\rangle) \cdot \mu_{\theta,\pi}(P)(S').$$

(In fact, properties (1) to (5) of expansions are sufficient to make this relationship hold for all phrases of the simple imperative language.)

As shown in [20], this development can be described succinctly in the language of category theory. State sets and expansions form a category Σ, with $\langle I_S, I_{S \to S_\perp}\rangle$ as the identity on S and $\langle g, G\rangle \cdot \langle g', G'\rangle = \langle g' \cdot g,\ G \cdot G'\rangle$ as composition. Then each D_θ and Env_π is a functor from Σ to the category *Dom* of domains and continuous functions, and the fundamental relationship given above is that $\mu_{\theta,\pi}(P)$ is a natural transformation from Env_π to D_θ.

5. Procedures and Their Declarations

To provide procedures, we introduce a binary operation $\to$ upon phrase types. A phrase of type $\theta_1 \to \theta_2$ denotes a procedure whose calls are phrases of type θ_2 containing an actual parameter of type θ_1. Multiple parameters will be treated by Currying, i.e.

$$P(A_1, \ldots, A_n) \text{ means } P(A_1)\cdots(A_n)$$

and

$$\lambda(F_1 : \theta_1, \ldots, F_n : \theta_n) \cdot B \text{ means } \lambda F_1 : \theta_1 \cdot \cdots \cdot \lambda F_n : \theta_n \cdot B.$$

This way of desugaring multiple parameters is sufficiently well known that we will not formalize it.

Thus what would conventionally be called a proper procedure (or τ function procedure) accepting parameters of types $\theta_1, \ldots, \theta_n$ is regarded as a phrase of type $\theta_1 \to \cdots \to \theta_n \to$ **comm** (or $\theta_1 \to \cdots \to \theta_n \to \tau$ **exp**). Note that parameterless proper procedures are simply commands (as was recognized in ALGOL W [22], where an actual parameter of this type could be any command), and parameterless function procedures are simply expressions (which is a natural and pleasant consequence of call by name).

It is easy to see that if $\theta_2 \leq \theta_2'$, then $\theta_1 \to \theta_2 \leq \theta_1 \to \theta_2'$. Less obviously, if $\theta_1' \leq \theta_1$, then $\theta_1 \to \theta_2 \leq \theta_1' \to \theta_2$. For example, since an integer expression can appear in any context requiring a real expression, a proper procedure accepting a real expression can also accept any integer expression and is therefore meaningful in any context requiring a proper procedure accepting an integer expression. Thus **real exp**$\to$**comm** $\leq$ **integer exp**$\to$**comm**.

In summary, the set of phrase types is the smallest set containing the primitive phrase types and closed under the binary operation $\to$. Its subtype relation is the least partial ordering satisfying the properties given earlier plus

$$\theta_1' \leq \theta_1 \text{ and } \theta_2 \leq \theta_2' \text{ implies } \theta_1 \to \theta_2 \leq \theta_1' \to \theta_2'.$$

In brief, $\to$ is antimonotone in its first operand and monotone in its second operand.

A suitable syntax for application (procedure statements and function designators), abstraction (lambda expressions), and least fixed-points (recursion) is

$$\langle \theta_2, \pi \rangle ::= \langle \theta_1 \to \theta_2, \pi \rangle (\langle \theta_1, \pi \rangle),$$

$$\langle \theta_1 \to \theta_2, \pi \rangle ::= \lambda \iota : \theta_1 \cdot \langle \theta_2, [\pi \mid \iota : \theta_1] \rangle,$$

$$\langle \theta, \pi \rangle ::= \mathbf{rec} \langle \theta \to \theta, \pi \rangle .$$

Here $[\pi \mid \iota : \theta_1]$ denotes the type assignment similar to π except that it maps ι into θ_1, i.e.

$$\mathrm{dom}([\pi \mid \iota : \theta_1]) = \mathrm{dom}(\pi) \cup \{\iota\},$$

$$[\pi \mid \iota : \theta_1](\iota) = \theta_1,$$

$$[\pi \mid \iota : \theta_1](\iota') = \pi(\iota') \quad \text{when } \iota' \neq \iota.$$

For later developments, it will be convenient to extend this notation by

using the following abbreviations:

$$[\pi \mid \iota_1 : \theta_1 \mid \cdots \mid \iota_n : \theta_n] = [\cdots[\pi \mid \iota_1 : \theta_1]\cdots \mid \iota_n : \theta_n],$$

$$[\iota_1 : \theta_1 \mid \cdots \mid \iota_n : \theta_n] = [\varepsilon \mid \iota_1 : \theta_1 \mid \cdots \mid \iota_n : \theta_n]$$

where ε is the type assignment with empty domain. Note that the latter form can be used to notate any type assignment explicitly.

The obvious approach to semantics is to take the meanings of phrases of type $\theta_1 \to \theta_2$ to be continuous functions from meanings of phrases of type θ_1 to meanings of phrases of type θ_2, i.e. $D_{\theta_1 \to \theta_2}(S) = D_{\theta_1}(S) \to D_{\theta_2}(S)$. However, when we consider variable declarations in the next section we will find that this approach conflicts with Principle 5.

Even in the absence of a definite semantics, meaning can be clarified by equivalences. We write $P \equiv_{\theta,\pi} Q$ to indicate that $\mu_{\theta,\pi}(P) = \mu_{\theta,\pi}(Q)$, i.e. that P and Q have the same meaning when regarded as phrases in $\langle \theta, \pi \rangle$.

First we have the standard equivalences of the (typed) lambda calculus. If $P \in \langle \theta_2, [\pi \mid \iota : \theta_1] \rangle$ and $Q \in \langle \theta_1, \pi \rangle$, then

$$(\lambda \iota : \theta_1 \cdot P)(Q) \equiv_{\theta_2,\pi} P|_{\iota \to Q}, \qquad \text{(beta reduction)}$$

where $P|_{\iota \to Q}$ denotes the result of substituting Q for the free occurrences of ι in P, with appropriate renaming of bound identifiers in P. If $P \in \langle \theta_1 \to \theta_2, \pi \rangle$ and $\iota \notin \mathrm{dom}(\pi)$, then

$$\lambda \iota : \theta_1 \cdot (P(\iota)) \equiv_{\theta_1 \to \theta_2, \pi} P \qquad \text{(eta reduction)}$$

Next, an obvious equivalence describes the fixed-point property of **rec**. If $P \in \langle \theta \to \theta, \pi \rangle$, then

$$\textbf{rec}\ P \equiv_{\theta,\pi} P(\textbf{rec}\ P).$$

Finally, two equivalences relate procedures to the conditional construction. If $P \in \langle \text{Boolean exp}, \pi \rangle$, Q, $R \in \langle \theta_1 \to \theta_2, \pi \rangle$ and $S \in \langle \theta_1, \pi \rangle$, then

$$(\textbf{if}\ P\ \textbf{then}\ Q\ \textbf{else}\ R)(S) \equiv_{\theta_2,\pi} \textbf{if}\ P\ \textbf{then}\ Q(S)\ \textbf{else}\ R(S).$$

If $P \in \langle \text{Boolean exp}, \pi \rangle$, Q, $R \in \langle \theta_2, [\pi \mid \iota : \theta_1] \rangle$, and $\iota \notin \mathrm{dom}(\pi)$, then

$$\lambda \iota : \theta_1 \cdot \textbf{if}\ P\ \textbf{then}\ Q\ \textbf{else}\ R \equiv_{\theta_1 \to \theta_2, \pi} \textbf{if}\ P\ \textbf{then}\ \lambda \iota : \theta_1 \cdot Q\ \textbf{else}\ \lambda \iota : \theta_1 \cdot R.$$

For the declaration of procedures, we prefer the **let** and **letrec** notation of Landin [3] to that of ALGOL 60; it is uniformly applicable to all phrase types (not just procedures), it distinguishes clearly between nonrecursive

and recursive cases, and it doesn't make declarations look like commands. The syntax is

$$\langle \theta, \pi \rangle ::= \textbf{let } \iota_1 \textbf{ be } \langle \theta_1, \pi \rangle \,\&\cdots\&\, \iota_n \textbf{ be } \langle \theta_n, \pi \rangle \textbf{ in } \langle \theta, \pi' \rangle$$
$$\mid \textbf{letrec } \iota_1 : \theta_1 \textbf{ be } \langle \theta_1, \pi' \rangle \,\&\cdots\&\, \iota_n : \theta_n \textbf{ be } \langle \theta_n, \pi' \rangle \textbf{ in } \langle \theta, \pi' \rangle$$

where $\pi' = [\pi \mid \iota_1 : \theta_1 \mid \cdots \mid \iota_n : \theta_n]$. (Note that the types $\theta_1, \ldots, \theta_n$ must be stated explicitly for **letrec** but not **let**.)

This notation can be defined as syntactic sugar in terms of application, abstraction, and **rec**. The nonrecursive **let** construction is straightforward. If $P_1 \in \langle \theta_1, \pi \rangle, \ldots, P_n \in \langle \theta_n, \pi \rangle$, and $P \in \langle \theta, \pi' \rangle$, then

$$\textbf{let } \iota_1 \textbf{ be } P_1 \,\&\cdots\&\, \iota_n \textbf{ be } P_n \textbf{ in } P \equiv_{\theta, \pi}$$
$$(\lambda \iota_1 : \theta_1 \cdot \cdots \cdot \lambda \iota_n : \theta_n \cdot P)(P_1) \cdots (P_n).$$

This equivalence can be used to remove all occurrences of **let** from a program without changing its meaning. Although it is formally similar to the equivalence given by Landin [3], it has a different import since call by name is being used instead of call by value. For example, if E is an integer expression, then **let** x **be** E **in** 3 has the same meaning as (λx: **integer exp.** 3)(E) which, by beta reduction, has the same meaning as 3, even when E is nonterminating. If x and y are integer variables, **let** z **be** x **in** $(x := 4; y := z)$ has the same meaning as (λz: **integer integer var** $\cdot (x := 4; y := z))(x)$ which, by beta reduction, has the same meaning as $x := 4; y := x$.

To treat the recursive **letrec** construction, we will first define the nonmultiple case and then show how multiple declarations can be reduced to this case. For the nonmultiple case we follow Landin: If $P_1 \in \langle \theta_1, [\pi \mid \iota_1 : \theta_1] \rangle$ and $P \in \langle \theta, [\pi \mid \iota_1 : \theta_1] \rangle$, then

$$\textbf{letrec } \iota_1 : \theta_1 \textbf{ be } P_1 \textbf{ in } P \equiv_{\theta, \pi}$$
$$(\lambda \iota_1 : \theta_1 \cdot P)(\textbf{rec } \lambda \iota_1 : \theta_1 \cdot P_1).$$

For the multiple case we give an equivalence, suggested by F.J. Oles, that avoids the use of products of phrase types. If $P_1 \in \langle \theta_1, \pi' \rangle, \ldots, P_n \in \langle \theta_n, \pi' \rangle$, and $P \in \langle \theta, \pi' \rangle$, where $\pi' = [\pi \mid \iota_1 : \theta_1 \mid \cdots \mid \iota_n : \theta_n]$, then

$$\textbf{letrec } \iota_1 : \theta_1 \textbf{ be } P_1 \,\&\cdots\&\, \iota_n : \theta_n \textbf{ be } P_n \textbf{ in } P \equiv_{\theta, \pi}$$
$$\textbf{letrec } \iota_1 : \theta_1 \textbf{ be}$$
$$(\textbf{letrec } \iota_2 : \theta_2 \textbf{ be } P_2 \,\&\cdots\&\, \iota_n : \theta_n \textbf{ be } P_n \textbf{ in } P_1)$$
$$\textbf{in } (\textbf{lectrec } \iota_2 : \theta_2 \textbf{ be } P_2 \,\&\cdots\&\, \iota_n : \theta_n \text{ be } P_n \textbf{ in } P).$$

6. Variable Declarations

To declare variables, we use the syntax

$$\langle \text{comm}, \pi \rangle ::= \textbf{new } \tau \textbf{ var } \iota \textbf{ in } \langle \text{comm}, [\pi \,|\, \iota : \tau\ \tau\ \text{var}] \rangle$$

(Note that declared variables always accept and produce the same data type.) However, since this construction involves binding we want to desugar it into a form in which the binding is done by a lambda expression. The solution is to introduce the more basic construction

$$\langle \text{comm}, \pi \rangle ::= \textbf{newvar}(\tau)\langle \tau\ \tau\ \text{var} \rightarrow \text{comm}, \pi \rangle$$

and to define

$$\textbf{new } \tau \textbf{ var } \iota \textbf{ in } P \equiv_{\text{comm},\pi} \textbf{newvar}(\tau)\ \lambda \iota : \tau\ \tau\ \textbf{var} \cdot P,$$

where $P \in \langle \text{comm}, [\pi \,|\, \iota : \tau\ \tau\ \text{var}] \rangle$.

Semantically, variable declarations raise a serious problem. The conventional approach is to permit the set S of store states to contain states with different numbers of L-values, and to define variable declaration to be an operation that adds an L-value to the state. For example, one might take a state to be a collection of strings of values for each data type

$$S = V^*_{\textbf{integer}} \times V^*_{\textbf{real}} \times V^*_{\textbf{Boolean}},$$

and define the declaration of a τ variable to be an operation that adds one more component of the string of values of type τ.

The problem with this view is that it violates Principle 5 by obscuring the stack discipline. Execution of a command containing variable declarations permanently alters the shape of the store, i.e. the number of L-values or the length of the component strings. In effect, the store is a heap without a garbage collector, rather than a stack. It is hardly surprising that this kind of model inspired languages that are closer to LISP than to ALGOL.

Our solution to this difficulty takes advantage of the fact that the semantics of a phrase is a family of environment-to-meaning functions for different sets of states. Instead of using a single set containing states of different shapes and regarding variable declaration as changing the shape of a state, we use sets of states with the same shape and regard variable declaration as changing the set of states. Specifically, if C is **new** τ **var** ι **in** C', then the semantics of C for a state set S depends upon the semantics of C' for the state set $S \times V_\tau$. Thus, since the semantics of C for S maps an

environment into a mapping in $D_{\mathbf{comm}}(S) = S \to S_\perp$, it is obvious that executing C will not change the shape of a state.

To make this precise, suppose $C' \in \langle \text{comm}, [\pi \mid \iota : \tau \ \tau \ \text{var}] \rangle$, so that $C \in \langle \text{comm}, \pi \rangle$. We first note that S and $S \times V_\tau$ are related by the expansion $\langle g, G \rangle$ in which g is the function from $S \times V_\tau$ to S such that $g(\langle \sigma, v \rangle) = \sigma$ and G is the function from $S \to S_\perp$ to $(S \times V_\tau) \to (S \times V_\tau)_\perp$ such that

$$G(c)(\langle \sigma, v \rangle) = \textbf{if } c(\sigma) = \perp \textbf{ then } \perp \textbf{ else } \langle c(\sigma), v \rangle.$$

This expansion induces a function $\text{Env}_\pi(\langle g, G \rangle)$ from $\text{Env}_\pi(S)$ to $\text{Env}_\pi(S \times V_\tau)$.

Let e be the function from $S \times V_\tau$ to $(V_\tau)_\perp$ such that $e(\langle \sigma, v \rangle) = v$, and a be a function from V_τ to $(S \times V_\tau) \to (S \times V_\tau)_\perp$ such that $a(v')(\langle \sigma, v \rangle) = \langle \sigma, v' \rangle$. Then $\langle a, e \rangle \in D_{\tau\, \tau\, \mathbf{var}}(S \times V_\tau)$ is an appropriate meaning for the variable being declared.

To obtain the meaning of **new** τ **var** ι **in** C' for the state set S and an environment $\eta \in \text{Env}_\pi(S)$, we use $\text{Env}_\pi(\langle g, G \rangle)$ to map η into $\text{Env}_\pi(S \times V_\tau)$ and then alter the resulting environment to map ι into $\langle a, e \rangle$, obtaining

$$\eta' = [\text{Env}_\pi(\langle g, G \rangle)(\eta) \mid \iota : \langle a, e \rangle].$$

Then we take the meaning of C' for the state set $S \times V_\tau$ and the environment η', and compose this meaning, which is a state-transition function from $S \times V_\tau$ to $(S \times V_\tau)_\perp$, with a function that initializes the new variable to some standard initial value $\text{init}_\tau \in V_\tau$, and a function which forgets the final value of the variable:

$$\mu_{\text{comm}, \pi}(\textbf{new } \tau \textbf{ var } \iota \textbf{ in } C')(S)(\eta) =$$

$$= (\lambda\sigma \cdot \langle \sigma, \text{init}_\tau \rangle) \cdot \mu_{\text{comm}, [\pi \mid \iota : \tau\, \tau\, \text{var}]}(C')(S \times V_\tau)(\eta') \cdot (g_\perp).$$

(Our unALGOL-like use of a standard initialization is the simplest way to avoid the abyss of nondeterminate semantics.)

However, this approach to variable declaration has a radical effect on the notion of what procedures mean that forces us to abandon the conventional idea that $D_{\theta_1 \to \theta_2}(S) = D_{\theta_1}(S) \to D_{\theta_2}(S)$. The problem is that variable declarations may intervene between the point of definition of a procedure and its point of call, so that the state set S' relevant to the call is different than the state set S at the point of definition, though there must be an expansion from S to S'.

As a consequence, a member p of $D_{\theta_1 \to \theta_2}(S)$ must be a family of functions describing the meaning of a procedure for different S'. Moreover, each of these functions, in addition to accepting the usual argument in $D_{\theta_1}(S')$ must also accept an expansion from S to S' that shows how the states of S are embedded in the richer states of S'.

As one might expect, the members of the family p must satisfy a stringent relationship (which can be expressed by saying that p is an appropriate kind of natural transformation). A precise definition is the following (where expand(S, S') is the set of expansions from S to S'): $p \in D_{\theta_1 \to \theta_2}(S)$ if and only if p is a state-set-indexed family of functions,

$$p(S') \in \text{expand}(S, S') \times D_{\theta_1}(S') \to D_{\theta_2}(S'),$$

such that, for all $\langle g, G \rangle \in \text{expand}(S, S')$, $\langle g', G' \rangle \in \text{expand}(S', S'')$, and $a \in D_{\theta_1}(S')$.

$$D_{\theta_2}(\langle g', G' \rangle)(p(S')(\langle g, G \rangle, a)) =$$

$$= p(S'')(\langle g' \cdot g, G \cdot G' \rangle, D_{\theta_1}(\langle g', G' \rangle)(a)).$$

To make $D_{\theta_1 \to \theta_2}(S)$ into a domain, its members are ordered pointwise, i.e. $p_1 \sqsubseteq p_2$ if and only if $(\forall S')\ p_1(S') \sqsubseteq p_2(S')$.

Finally, we must say how an expansion from S to S' induces a function from $D_{\theta_1 \to \theta_2}(S)$ to $D_{\theta_1 \to \theta_2}(S')$: If $\langle g, G \rangle \in \text{expand}(S, S')$ and $p \in D_{\theta_1 \to \theta_2}(S)$, then $D_{\theta_1 \to \theta_2}(\langle g, G \rangle)(p) \in D_{\theta_1 \to \theta_2}(S')$ is the family p' of functions such that, for all S'', $\langle g', G' \rangle \in \text{expand}(S', S'')$, and $a \in D_{\theta_1}(S'')$,

$$p'(S'')(\langle g', G' \rangle, a) = p(S'')(\langle g' \cdot g, G \cdot G' \rangle, a).$$

A full description of this kind of semantics is presented in [20]; in particular abstraction and application are defined and the validity of beta and eta reduction is proved. This is done by showing that the above definition of $\to$ makes Dom^{Σ} (the category of functors and natural transformations from Σ to *Dom*) into a Cartesian closed category, which is an extremely general model of the typed lambda calculus.

Despite its apparent complexity, much of which is due to our avoidance of category theory in this exposition, this kind of semantics shows that our language is obtained by adding the typed lambda calculus to the simple imperative language in a way that imposes a stack discipline. The essential idea is that the procedure mechanism involves a 'hidden abstraction' over a family of semantics indexed by state sets.

We suspect that this kind of hidden abstraction may arise in other situations where a formal language is extended by adding a procedural or definitional mechanism based on the lambda calculus. The generality of the idea is indicated by the fact that the definition of $\rightarrow$ and the proof that Dom^{Σ} is Cartesian closed do not depend upon the nature of the category Σ.

7. Call by Value

In the ALGOL 60 report, call by value is explained in terms of call by name by saying that a value specification is equivalent to a certain modification of the procedure body. In fact, however, this modification involves only the body and not the formal parameter list, so that it is equally applicable to commands that are not procedure bodies. In essence, call by value is really an operation on commands rather than parameters.

To capture this idea, we introduce the syntax

$$\langle \text{comm}, [\pi \mid \iota : \tau \text{ exp}] \rangle ::= \tau \textbf{ value } \iota \textbf{ in } \langle \text{comm}, [\pi \mid \iota : \tau\, \tau \text{ var}] \rangle$$

which is desugared by the equivalence

$$\tau \textbf{ value } \iota \textbf{ in } C \equiv_{\text{comm}, [\pi \mid \iota : \tau \text{ exp}]} \textbf{new } \tau \textbf{ var } \iota' \textbf{ in } (\iota' := \iota;\ (\lambda \iota : \tau\ \tau \textbf{ var} \cdot C)(\iota')),$$

where $C \in \langle \text{comm}, [\pi \mid \iota : \tau\ \tau \text{ var}] \rangle$ and $\iota ! \notin \text{dom}(\pi) \cup \{\iota\}$. (This is only a generalization of call by value for proper procedures; an analogous generalization for function procedures would require block expressions.)

Notice that τ **value** ι **in** C has a peculiar binding structure: the first occurrence of ι is a binder whose scope is C, yet this occurrence is itself free. (A similar phenomenon occurs in the conventional notation for indefinite integration.)

Call by result, as in ALGOL W [22], can obviously be treated similarly.

8. Arrays

Arrays of the kind used in ALGOL 60 can be viewed as procedures whose calls are variables. Thus an n-dimensional τ array is a phrase of type

$$\underbrace{\textbf{integer exp} \rightarrow \cdots \rightarrow \textbf{integer exp}}_{n \text{ times}} \rightarrow \tau\ \tau \textbf{ var}.$$

(Notice that this is a phrase type. If arrays were introduced as a data type, one could assign to entire array variables (as in APL) but not to their elements.)

The declaration of such arrays is a straightforward generalization of variable declarations, and can be desugared similarly. The details are left to the reader.

Unfortunately, this kind of array, like that of ALGOL, has the shortcoming that it does not carry its own bounds information. A possible solution is to introduce, for each $n \geq 1$, a phrase type **array** (n, τ) that is a subtype of the type displayed above, and to provide bound-extraction operations that act upon these new phrase types. The concept of array in [28] could be treated similarly.

9. Labels

Since all one can do with a label ι is to jump to it, its meaning can be taken to be the meaning of **goto** ι. Thus labels can be viewed as identifiers of phrase type **comm**, and **goto** ι can simply be written as ι.

However, as suggested in ALGOL 68, labels denote a special kind of command, which we will call a *completion,* that has the property that it never returns control to its successor. If completions are not distinguished as a separate phrase type, it becomes difficult for either a human reader or a compiler to analyze control flow, particularly when procedure parameters denoting completions are only specified to be commands. To avoid this, we introduce **compl**(etion) as an additional phrase type that is a subtype of **comm** (so that completions can always be used as commands but not vice-versa).

Thus labels are identifiers of phrase type **compl**. Moreover, the production schemas for conditional and case constructions, procedure application, and recursion provide a variety of compound phrases of type **compl**. This variety can be enriched by the following syntax, in which various ways of forming commands are used to form completions:

$$\langle \text{compl}, \pi \rangle ::= \langle \text{comm}, \pi \rangle ; \langle \text{compl}, \pi \rangle$$

$$\mid \textbf{new } \tau \textbf{ var } \iota \textbf{ in } \langle \text{compl}, [\pi \mid \iota : \tau\, \tau \text{var}] \rangle$$

$$\parallel \textbf{newvar } (\tau)\ \langle \tau\ \tau\ \text{var} \rightarrow \text{compl}, \pi \rangle$$

$$\langle \text{compl}, [\pi \mid \iota : \tau\ \text{exp}] \rangle ::= \tau \textbf{ value } \iota \textbf{ in } \langle \text{compl}, [\pi \mid \iota : \tau\ \tau\ \text{var}] \rangle .$$

Two more schemas suffice to describe commands and completions in which labels are declared in an ALGOL-like notation:

$$\langle \text{comm}, \pi \rangle ::= \langle \text{comm}, \pi' \rangle ; \iota_1 : \langle \text{comm}, \pi' \rangle ; \cdots ; \iota_n : \langle \text{comm}, \pi' \rangle$$

where $\iota_1, \ldots, \iota_n$ are distinct and $\pi' = [\pi \mid \iota_1 : \text{compl} \mid \cdots \mid \iota_n : \text{compl}]$;

$$\langle \text{compl}, \pi \rangle ::= \langle \text{comm}, \pi' \rangle ; \iota_1 : \langle \text{compl}, \pi' \rangle ; \cdots ; \iota_n : \langle \text{compl}, \pi' \rangle$$

where $\iota_1, \ldots, \iota_n$ are distinct and $\pi' = [\pi \mid \iota_1 : \text{compl} \mid \cdots \mid \iota_n : \text{compl}]$.

Since these declarative constructions involve binding, we must desugar them into more basic forms. For this purpose, we introduce an escape operation that is a paremeterless variant of Landin's J-operator [3].

$$\langle \text{comm}, \pi \rangle ::= \textbf{escape } \langle \text{compl} \rightarrow \text{comm}, \pi \rangle .$$

This operation can be described in terms of a conventional label declaration: If $P \in \langle \text{compl} \rightarrow \text{comm}, \pi \rangle$ and $\iota \notin \text{dom}(\pi)$, then

$$\textbf{escape } P \equiv_{\text{comm}, \pi} (P(\iota); \iota : \textbf{skip}).$$

Our present goal, however, is the reverse. To describe label declarations in terms of escapes, we proceed in two steps. First, we describe a label-declaring command in terms of a label-declaring completion by adding a final jump to an enclosing escape: If $\iota_1, \ldots, \iota_n, \iota$ are distinct identifiers, $\pi' = [\pi \mid \iota_1 : \text{compl} \mid \cdots \mid \iota_n : \text{compl}]$, $C_0, \ldots, C_n \in \langle \text{comm}, \pi' \rangle$, and $\iota \notin \text{dom}(\pi)$, then

$$C_0 ; \iota_1 : C_1 ; \ldots ; \iota_n : C_n \equiv_{\text{comm}, \pi}$$
$$\textbf{escape } \lambda \iota : \textbf{compl}. \ (C_0 ; \iota_1 : C_1 ; \ldots ; \iota_n : (C_n : \iota)).$$

Then we describe a label-declaring completion in terms of recursive definitions: If $\iota_1, \ldots, \iota_n$ are distinct identifiers, $\pi' = [\pi \mid \iota_1 : \text{compl} \mid \cdots \mid \iota_n : \text{compl}]$, $C_0, \ldots, C_{n-1} \in \langle \text{comm}, \pi' \rangle$, and $K \in \langle \text{compl}, \pi' \rangle$, then

$$C_0 ; \iota_1 : C_1 ; \ldots ; \iota_n : K \equiv_{\text{compl}, \pi}$$
$$\textbf{letrec } \iota_1 : \textbf{compl be } (C_1 ; \iota_2) \ \& \cdots \& \ \iota_{n-1} : \textbf{compl be } (C_{n-1} ; \iota_n)$$
$$\& \ \iota_n : \textbf{compl be } K$$
$$\textbf{in } (C_0 ; \iota_1).$$

We have chosen to desugar the ALGOL notation for declaring labels

because of its familiarity. Other, possibly preferable notations can be treated similarly; for example, Zahn's event facility [29] can be described by escapes without recursion. Actually, the wisest approach might be to avoid all syntactic sugar and simply provide escapes.

Semantically, the introduction of labels requires a change from direct to continuation semantics, which will not be discussed here. In [20] it is shown that hidden abstraction on state sets can be extended to continuation semantics, though with a different notion of expansion.

10. Products and Sums

Although procedures and arrays are the only ways of building compound phrase types in ALGOL, most newer languages provide some kind of product of types, such as records in ALGOL W or class members in Simula 67 [26], and often some kind of sum of types, such as unions in ALGOL 68 or variant records in PASCAL. In this section we will explore the addition of such mechanisms to our illustrative language.

Since we distinguish two kinds of type, we must decide whether to have products of data types or phrase types (or both). Products of data types would be record-like entities, except that one would always assign to entire records rather than their components. (Complex numbers are a good example of a simple product of data type.) On the other hand, products of phrase types are more like members of SIMULA classes than like records; one can never assign to the entire object, but only to components that are variables; other types of components, such as procedures, are also possible. In this paper, we will only consider products (and sums) of phrase types, thereby retaining the ALGOL characteristic that data types are never compound.

We must also decide between numbered and named products, i.e. between selecting components by an ordinal or by an identifier (i.e. field name). In this paper we will explore named products, since they are more commonly used than numbered products, and also since they are amenable to a richer subtype relationship.

To introduct named products of phrase types, we expand the set of phrase types to include

prod π,

where π is a type assignment. Usually we will write products in the form **prod** $[\iota_1 : \theta_1 | \cdots | \iota_n : \theta_n]$, where $\iota_1, \ldots, \iota_n$ are distinct identifiers. However, it should be understood that the phrase type denoted by this expression is independent of the ordering of the pairs $\iota_k : \theta_k$.

For a subtype ordering, one at least wants a component-wise ordering. But a more interesting and useful structure arises if we permit implicit conversions that drop components, e.g.

prod [age: **integer exp** | sex: **Boolean exp** | salary: **integer var**]
$\leq$ **prod** [age: **integer exp** | salary: **integer var**].

In general, we have

prod $\pi \leq$ **prod** π' if and only if

$$\mathrm{dom}(\pi') \subseteq \mathrm{dom}(\pi) \text{ and } (\forall \iota \in \mathrm{dom}(\pi'))\ \pi(\iota) \leq \pi'(\iota).$$

Next we introduce the syntax of phrases for constructing products and selecting their components:

$$\langle \text{prod } [\iota_1 : \theta_1 | \cdots | \iota_n : \theta_n],\ \pi \rangle ::= \langle \iota_1 : \langle \theta_1, \pi \rangle, \ldots, \iota_n : \langle \theta_n, \pi \rangle \rangle$$

where $\iota_1, \ldots, \iota_n$ are distinct identifiers

$$\langle \theta, \pi \rangle ::= \langle \text{prod}[\iota : \theta], \pi \rangle \cdot \iota.$$

In the second production, notice that our subtype ordering permits us to write $[\iota : \theta]$ instead of $[\cdots | \iota : \theta | \cdots]$.

The semantics of products is explicated by the following equivalences: When $P_1 \in \langle \theta_1, \pi \rangle, \ldots, P_n \in \langle \theta_n, \pi \rangle$, $\iota_1, \ldots, \iota_n$ are distinct, and $1 \leq k \leq n$,

$$\langle \iota_1 : P_1, \ldots, \iota_n : P_n \rangle \cdot \iota_k \equiv_{\theta_k, \pi} P_k$$

When $P \in \langle \text{prod } \pi', \pi \rangle$ and $\mathrm{dom}(\pi') = \{\iota_1, \ldots, \iota_n\}$,

$$\langle \iota_1 : (P \cdot \iota_1), \ldots, \iota_n : (P \cdot \iota_n)) \equiv_{\text{prod } \pi', \pi} P$$

We have mentioned that this kind of product is closely related to the class concept of SIMULA 67. In [25] it is shown that class declarations (in the reference-free sense of Hoare [27] rather than of SIMULA itself) can be desugared into constructions using such products.

Finally, we introduce named sums of phrase types. (Roughly speaking, type sums are disjoint unions.) We expand the set of phrase types to include

sum π,

where π is a type assignment. The subtype relation is

sum $\pi \leq$ **sum** π' if and only if

$\mathrm{dom}(\pi) \subseteq \mathrm{dom}(\pi')$ and $(\forall \iota \in \mathrm{dom}(\pi))\ \pi(\iota) \leq \pi'(\iota)$.

In contrast to the situation with products, a subtype of a sum can contain fewer (rather than more) alternatives.

To construct sums and to do case analysis, we introduce the syntax

$\langle \mathrm{sum}[\iota : \theta], \pi \rangle ::=$ **tag** $\iota : \langle \theta, \pi \rangle$,

$\langle \theta, \pi \rangle ::=$ **sumcase** ι **is** $\langle \mathrm{sum}[\iota_1 : \theta_1 \mid \cdots \mid \iota_n : \theta_n], \pi \rangle$

in $(\iota_1 : \langle \theta, [\pi \mid \iota : \theta_1] \rangle, \ldots, \iota_n : \langle \theta, [\pi \mid \iota : \theta_n] \rangle)$

where $\iota_1, \ldots, \iota_n$ are distinct identifiers.

Again, the semantics can be explicated by equivalences. When $\iota_1, \ldots, \iota_n$ are distinct, $1 \leq k \leq n$, $P_1 \in \langle \theta, [\pi \mid \iota : \theta_1] \rangle, \ldots, P_n \in \langle \theta, [\pi \mid \iota : \theta_n] \rangle$, and $A \in \langle \theta_k, \pi \rangle$,

sumcase ι **is tag** $\iota_k : A$ **in** $(\iota_1 : P_1, \ldots, \iota_n : P_n) \equiv_{\theta, \pi}$

let ι **be** A **in** P_k.

When $S \in \langle \mathrm{sum}\ \pi', \pi \rangle$ and $\mathrm{dom}(\pi') = \{\iota_1, \ldots, \iota_n\}$,

sumcase ι **is** S **in** $(\iota_1 :$ **tag** $\iota_1 : \iota, \ldots, \iota_n :$ **tag** $\iota_n : \iota) \equiv_{\mathrm{sum}\ \pi', \pi} S$.

Since **sumcase** is a binding operation, Principle 1 requires us to express it in terms of a construction in which the binding is done by lambda expressions. For this purpose, we introduce the idea of 'source-tupling'. Suppose $P_1, \ldots, P_n$ are procedures of phrase types $\theta_1 \to \theta, \ldots, \theta_n \to \theta$ respectively. Then **sourcetuple**$(\iota_1 : P_1, \ldots, \iota_n : P_n)$ is a procedure of type **sum**$[\iota_1 : \theta_1 \mid \cdots \mid \iota_n : \theta_n] \to \theta$ that will behave like P_k when applied to a parameter tagged with ι_k.

To make this precise we use the syntax

$\langle \mathrm{sum}[\iota_1 : \theta_1 \mid \cdots \mid \iota_n : \theta_n] \to \theta, \pi \rangle ::=$

sourcetuple$(\iota_1 : \langle \theta_1 \to \theta, \pi \rangle, \ldots, \iota_n : \langle \theta_n \to \theta, \pi \rangle)$

where $\iota_1, \ldots, \iota_n$ are distinct identifiers.

Then **sumcase** is desugared by the following equivalence: If $\iota_1, \ldots, \iota_n$ are distinct, $S \in \langle \mathrm{sum}[\iota_1 : \theta_1 \mid \cdots \mid \iota_n : \theta_n], \pi \rangle$, $P_1 \in \langle \theta, [\pi \mid \iota : \theta_1] \rangle, \ldots,$

$P_n \in \langle \theta, [\pi \mid \iota : \theta_n] \rangle$, then

$$\textbf{sumcase } \iota \textbf{ is } S \textbf{ in } (\iota_1 : P_1, \ldots, \iota_n : P_n) \equiv_{\theta, \pi}$$

$$\textbf{sourcetuple}(\iota_1 : \lambda \iota : \theta_1 \cdot P_1, \ldots, \iota_n : \lambda \iota : \theta_n \cdot P_n)(S).$$

It should be noted that sums of phrase types do not introduce any failure of typing such as the 'mutant variable record problem' of PASCAL, since one cannot change the tag of a sum by assignment. On the other hand, sums of data types would also avoid these problems since a branch on the tag of a value would not imply any assumption that a variable with that value would continue to possess the same tag. This suggests that the type-safety problem with sum-like constructions is due to a failure to distinguish data and phrase types.

11. Final Remarks

I have neglected the topic of program proving since I have discussed it elsewhere at length. Although Hoare's work on proving procedures is incompatible with call by name and procedure parameters, an alternative approach called specification logic appears promising. In [23] this logic is formulated for a subset of ALGOL W; in [24] it is given for a language closer to that described here.

Like ALGOL itself, our illustrative language raises problems of interference, i.e. variable aliasing and interfering side effects of statements and proper procedures. The language is rich enough that an assertion that two phrases do not interfere must be proved (as in specification logic) rather than derived syntactically. Several years ago in [25], I attempted to restrict a language like that described here to permit interference to be detected syntactically. Unfortunately, this work led to some nasty syntactic complications (described at the end of [25]) that have yet to be resolved. Still, I have hopes for the future of this approach.

Although this paper has dealt with nearly all the significant aspects of ALGOL 60, it has not gone much beyond the scope of that language. More for lack of understanding than space, I have avoided block expressions, user-defined types, polymorphic procedures, recursively defined types, indeterminate and concurrent computation, references, and compound data types.

It remains to be seen whether our model can be extended to cover these topics. Of course, some of them could reasonably be labelled unALGOL-like. But the essence of ALGOL is not a straightjacket. It is a conceptual universe for language design that one hopes will encompass languages far more general than its progenitor.

References

[1] P. Naur et al., Revised report on the algorithmic language ALGOL 60, Comm. ACM 6 (1) (January 1963) 1–17.

[2] P.J. Landin, A λ-calculus approach, in: L. Fox (Ed.), Advances in Programming and Non-Numerical Computation (Pergamon Press, Oxford, 1966) pp. 97–141.

[3] P.J. Landin, A correspondence between ALGOL 60 and Church's lambda-notation, Comm. ACM 8 (2,3) (February–March 1965) 89–101 and 158–165.

[4] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I, Comm. ACM 3 (4) (April 1960) 184–195.

[5] P. Lucas, P. Lauer and H. Stigleitner, Method and notation for the formal definition of programming languages, TR 25.087, IBM Laboratory Vienna. (June 1968).

[6] D. Scott and C. Strachey, Toward a mathematical semantics for computer languages, Proc. Symposium on Computers and Automata, Polytechnic Inst. of Brooklyn, Vol. 21, pp. 19–46. (Also: Technical Monograph PRG-6, Oxford Univ. Computing Lab., Programming Research Group (1971).)

[7] P.J. Landin, The next 700 programming languages, Comm. ACM 9 (3) (March 1966) 157–166.

[8] A. Evans, PAL – A language designed for teaching programming linguistics, Proc. ACM 23rd Natl. Conf., 1968 (Brandin Systems Press, Princeton, NJ) pp. 395–403.

[9] J.C. Reynolds, GEDANKEN – A simple typeless language based on the principle of completeness and the reference concept, Comm. ACM 13 (5) (May 1970) 308–319.

[10] A. van Wijngaarden (Ed.), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Report on the algorithmic language ALGOL 68, MR 101, Mathematisch Centrum, Amsterdam (February 1969), also Numer. Math. 14 (1969) 79–218.

[11] C.A.R. Hoare, Recursive data structures, Int. J. Comput. Information Sci. 4 (2) (June 1975) 105–132.

[12] N. Wirth, The programming language PASCAL, Acta Inform. 1 (1) (1971) 35–63.

[13] R.L. London, J.V. Guttag, J.J. Horning, B.W. Lampson, J.G. Mitchell and G.J. Popek, Proof rules for the programming language EUCLID, Acta Inform. 10 (1) (1978) 1–26.

[14] C.M. Geschke, J.H. Morris, and E.H. Satterthwaite, Early experience with MESA, Comm. ACM 20 (8) (August 1977) 540–553.

[15] J.D. Ichbiah, J.C. Heliard, O. Roubine, J.G.P. Barnes, B. Krieg-Brueckner and B.A. Wichmann, Preliminary ADA reference manual and Rationale for the design of the ADA programming language, SIGPLAN Notices 14 (6) (June 1979).

[16] C.A.R. Hoare, An axiomatic basis for computer programming, Comm. ACM 12 (10) (October 1969) 576–580 and 583.

[17] C.A.R. Hoare and N. Wirth, An axiomatic definition of the programming language PASCAL, Acta Inform. 2 (4) (1973) 335–355.

[18] C.A.R. Hoare, Procedures and parameters: An axiomatic approach, in E. Engeler (Ed.), Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics Vol. 188 (Springer-Verlag, Berlin, 1971) pp. 102–116.

[19] J.C. Reynolds, Using category theory to design implicit conversions and generic operators, in: N.D. Jones (Ed.), Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark, January 14–18, 1980, Lecture Notes in Computer Science, Vol. 94 (Springer-Verlag, Berlin, 1980) pp. 211–258.

[20] F.J. Oles, A category-theoretic approach to the semantics of programming languages, Ph.D. Dissertation (in progress), Syracuse University.

[21] R.M. Burstall, J.S. Collins and R.J. Popplestone, Programming in POP-2 (Edinburgh University Press, 1971).

[22] N. Wirth and C.A.R. Hoare, A contribution to the development of ALGOL, Comm. ACM 9 (6) (June 1966) 413–432.

[23] J.C. Reynolds, The Craft of Programming (Prentice-Hall, Englewood Cliffs, NJ, 1981).

[24] J.C. Reynolds, Idealized ALGOL and its specification logic, Computer and information Science, Syracuse University, Report 1–81 (July 1981).

[25] J.C. Reynolds, Syntactic control of interference, Conference Record of the Fifth Annual Symposium on Principles of Programming Languages, Tucson, January 1978 (Association for Computing Machinery, New York, 1978) pp. 39–46.

[26] O.-J. Dahl, B. Myhrhaug and K. Nygaard, Simula 67 Common Base Language, Norwegian Computing Centre, Oslo (1968).

[27] C.A.R. Hoare, Proof of correctness of data representations, Acta Inform. 1 (4) (1972) 271–281.

[28] E.W. Dijkstra, A Discipline of Programming (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[29] C.T. Zahn, A control statement for natural top-down structured programming, Proceedings, Programming Symposium, Paris, April 9–11, 1974, Lecture Notes in Computer Science, Vol. 19 (Springer-Verlag, Berlin, 1974) pp. 170–180.

Algorithmic Languages, de Bakker/van Vliet (eds.)

An Operational Semantics for Bounded Nondeterminism Equivalent to a Denotational One

R. Kuiper

Mathematical Centre, Kruislaan 413, 1098 *SJ Amsterdam, The Netherlands*

Dyadic nondeterministic choice is added to the programming language with recursive procedures as used in de Bakker's monograph on program correctness [5]. This leads to considerable changes in the operational semantics. The possible result of the execution of a program is no more given as a single state, but as a set of possible states. Furthermore, the execution of a program is no more given as a computation sequence but as a set of possible computation sequences with tree-like properties.

We present a 'natural' operational semantics $\mathcal{O}$ defined by means of a function $\mathcal{COMP}$, where $\mathcal{COMP}$ yields for each program $\mathcal{R}$ and each state σ a set of computation sequences, characterized by equations in the style of Cook [7]. For this set of equations we prove, in a topological setting, the existence of a unique solution and the equivalence of the operational semantics to the usual denotational one, defined by fixed point techniques.

0. Introduction

The subject of this paper is to investigate the effects of adding bounded nondeterministic choice to a simple language with recursive procedures on the definition and properties of the operational semantics.

The motivation to introduce an operational semantics is the following usual one. A method for proving program correctness is to abstract to a more mathematical level by defining a denotational semantics and to give a proof system on that level. A way to justify this abstraction is to define an operational semantics such that on the one hand it is intuitively close to the actual program execution and on the other hand can be proved to be

equivalent to the denotational semantics. We provide a 'natural' operational semantics; its justification and the proof of its equivalence to a denotational one are the main aims of this paper.

The reasons to add dyadic nondeterministic choice – as will be seen later, extension to finite choice introduces no extra problems – are twofold. Firstly, in practice nondeterministic choice enters the scene directly, cf. Dijkstra's guarded command [9], as well as indirectly, cf. parallellism and concurrency [12], where one process is selected to proceed, or one communication is selected to be executed. Secondly, in theory nondeterministic choice is a fairly easy setting in which tree-like structures appear instead of computation sequences as when dealing with deterministic sequential programs. This phenomenon also occurs as soon as parallel programming and concurrent processes are concerned and introduces considerable changes in the theoretical treatment. Contrary to the deterministic case, justification of the defined operational semantics in view of existence and uniqueness of the described function is not a clear case, and thus grew into a next-important aim in itself.

The framework we use is that developed in De Bakker's monograph on program correctness [5, especially Chapters 5 and 7]. The (ultra)metric distances defined between sets, and convergence with respect to such metrics we use, are also extensively employed by Nivat and Arnold ([13] and [2]) considering, among other subjects, infinite trees and nondeterminism. In their approach, trees are essentially programs, whereas we use trees of states, i.e. traces of program executions. Furthermore, Arnold and Nivat obtain the set of all trees by completion of the set of all finite trees. We describe a tree by the set of all paths in the tree; the set of all trees is the set of all paths restricted in a suitable way (cf. Definition 9).

It appears that at three stages of the development we are forced to make the same restriction on the set of sequences used. This restriction amounts to require a tree-like property with respect to the occurrence of infinite branches.

This central tree-like property already was observed by Back in [3] treating unbounded nondeterminism.

The setup of the paper is as follows. After this introduction, in Section 1 the syntax and some preliminary information are given. Section 2 starts with the definition of an operational semantics by means of the function $\mathcal{COMP}$, which in turn is defined by a set of equations. The main result here is the existence proof of a unique solution $\mathcal{COMP}$ of this set of equations. In

Section 3 a denotational semantics is described concisely. Finally, in Section 4 we prove the equivalence of the operational semantics to the denotational one.

1. Syntax and Preliminaries

Recursive procedures and finite nondeterministic choice are the key characteristics of the chosen language. Note, that subscripted variables are not treated (i.e., no arrays are present). Including these would necessitate a more complicated framework and only obscure our intentions. A straightforward extension is possible. The phrase "Let $(\alpha\in)C$ be specified by $\alpha ::= qr|x|\alpha_1\alpha_2$ is to be understood as: All α or α_i, $i\in I$ in the sequel are assumed to be elements of the set C; α is of the form qr or x or $\alpha_1\alpha_2$, where α_1, α_2 are elements of C already.

We now define the sets of the syntactic entities we use.

Definition 1 (Syntax). Let $(x\in)$ $\mathcal{VAR}$ be the set of integer variables. Let $(m\in)$ $\mathcal{CON}$ be the set of constants. Let $(P\in)$ $\mathcal{PVAR}$ be the set of procedure variables.

Let $(t\in)$ $\mathcal{IEXP}$ be the set of integer expressions specified by

$$t ::= x\,|\,m\,|\,t_1+t_2\,|\,\cdots\,|\text{ if } b \text{ then } t_1 \text{ else } t_2 \text{ fi}.$$

Let $(b\in)$ $\mathcal{BEXP}$ be the set of boolean expressions specified by

$$b ::= \text{true}\,|\,\text{false}\,|\,t_1=t_2\,|\,\cdots\,|\,\neg b\,|\,b_1\supset b_2.$$

Let $(S\in)$ $\mathcal{STAT}$ be the set of statements, specified by

$$S ::= x:=t\,|\,S_1;S_2\,|\,S_1\vee S_2|\text{ if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}\,|\,P.$$

Let $(E\in)$ $\mathcal{DECL}$ be the set of declarations, specified by

$$E ::= \langle P_i \Leftarrow S_i\rangle_{i=1}^{n},\ n\geq 0,\ P_i\not\equiv P_j,\ 1\leq i<j\leq n.$$

Let $(R\in)$ $\mathcal{PROG}$ be the set of programs, specified by

$$R ::= \langle E\,|\,S\rangle, \text{ for all } P \text{ in } S \text{ or } S_i,\ i=1,\ldots,n, \text{ there exists } j,\ 1\leq j\leq n \text{ such that } P\equiv P_j.$$

Note, that bounded choice now can be obtained by applying $(S_1\vee S_2)\vee S_3$.

The instances left open in $\mathscr{IEXP}$ and $\mathscr{BEXP}$ can be filled in with analogous expressions. Note that $\mathscr{PROG}$ is defined such, that all programs are closed, i.e. only these procedure variables occur in a program, for which the procedure body is given in the declaration E.

The following definitions concern assigning meaning to syntactic objects, i.e. semantics. At this stage, there is no distinction between operational and denotational semantics. Meaning is assigned by way of functions, defined by cases, from a syntactic domain to a domain of interpretation. To enable us later to define the rest of the denotational semantics we design the domains of interpretation as complete partial orders (cpo's).

Definition 2. $(C, \sqsubseteq)$ is a cpo iff

(i) $\sqsubseteq$ is a partial order on C,

(ii) there is an element $\perp \in C$ such that, for all $c \in C$, $\perp \sqsubseteq c$,

(iii) each chain $\langle c_i \rangle_{i=1}^{\infty}$ has a least upper bound $\bigsqcup_{i=1}^{\infty} c_i \in C$.

Definition 3 (Domains of interpretation). $V_0 = \mathbb{N}$, natural numbers; $W_0 = \{tt, ff\}$, truth values; $\Sigma_0 = \mathscr{IVAR} \to V_0$, functions assigning meaning to variables.

Let $(\alpha \in) V = V_0 \cup \{\perp_v\}$, cpo by $\alpha_1 \sqsubseteq \alpha_2$ iff $\alpha_1 = \perp_v$ or $\alpha_1 = \alpha_2$.

Let $(\beta \in) W = W_0 \cup \{\perp_w\}$, cpo analogously.

Let $(\sigma \in) \Sigma = \Sigma_0 \cup \{\perp\}$, cpo analogously.

Definition 4. For C cpo, c_1, c_2, $\perp_c \in C$

$$\text{if } \beta \text{ then } c_1 \text{ else } c_2 \text{ fi} = \begin{cases} c_1 & \text{if } \beta = tt, \\ c_2 & \text{if } \beta = ff, \\ \perp_c & \text{if } \beta = \perp_w. \end{cases}$$

We now define the meaning functions for integer and boolean expressions which yield, by cases, for each of the expressions and a state σ a value in one of the domains of interpretation.

Definition 5.

(a) $\mathscr{V} : \mathscr{IEXP} \to (\Sigma \to V)$,

$\mathscr{V}(t)(\perp) = \perp_v$.

For $\sigma \in \Sigma_0$

$$\mathscr{V}(x)(\sigma) = \sigma(x)$$

$$\mathscr{V}(m)(\sigma) = \alpha \quad \text{where } \alpha \text{ is the integer denoted by } m,$$

$$\mathscr{V}(t_1 + t_2)(\sigma) = \mathscr{V}(t_1)(\sigma) + \mathscr{V}(t_2)(\sigma),$$

$$\ldots$$

$$\mathscr{V}\ (\text{if } b \text{ then } t_1 \text{ else } t_2 \text{ fi})(\sigma) = \text{if } W(b)(\sigma) \text{ then } V(t_1)(\sigma) \text{ else } V(t_2)(\sigma) \text{ fi}.$$

(b)

$$\mathscr{W} : \mathscr{BEXP} \to (\Sigma \to W),$$

$$\mathscr{W}(b)(\perp) = \perp_w.$$

For $\sigma \in \Sigma_0$

$$\mathscr{W}(\text{true})(\sigma) = tt,$$

$$\mathscr{W}(\text{false})(\sigma) = ff,$$

$$\mathscr{W}(t_1 = t_2)(\sigma) = (\mathscr{V}(t_1)(\sigma) = \mathscr{V}(t_2)(\sigma)),$$

$$\ldots$$

$$\mathscr{W}(\neg b)(\sigma) = \neg\, \mathscr{W}(b)(\sigma),$$

$$\mathscr{W}(b_1 \supset b_2)(\sigma) = (\mathscr{W}(b_1)(\sigma) \Rightarrow \mathscr{W}(b_2)(\sigma)).$$

We end this chapter by introducing the notion variant of a state. The purpose of this is to be able to indicate the effect of executing a statement, for instance an assignment statement $x := t$ by a change in the state. The following definition enables us to change in a state σ the value σ assigns to a particular x.

Definition 6.

$$\perp\{\alpha/x\} = \perp,$$

$$\sigma\{\alpha/x_1\}(x_2) = \begin{cases} \alpha & \text{if } x_1 \equiv x_2, \\ \sigma(x_2) & \text{if } x_1 \not\equiv x_2. \end{cases}$$

2. The Operational Semantics

The aim here is to define an operational semantics which is intuitively close to the actual program execution.

In the deterministic case a well-known way to achieve this is by way of a Cook semantics [7]. A function $\mathcal{COMP}$ yields for each program R and each state σ a, possibly infinite, computation sequence of states, $\mathcal{COMP}(R)(\sigma) = \langle \sigma_1, \sigma_2, \dots \rangle$. Intuitively, these states correspond to the states a computer goes through when executing R, starting in σ. The operational semantics then is a function $\mathcal{O}$ which yields for each program R and each state σ the state $\kappa(\mathcal{COMP}(R)(\sigma))$, this being the last element of $\mathcal{COMP}(R)(\sigma)$ if this sequence is finite and the special state $\perp$ otherwise.

Now intuitively $\mathcal{COMP}$ should be defined by rules, stepwise generating the computation sequences; a Cook semantics does so by cases, the cases being possible program forms. For example,

$$\mathcal{COMP}(\langle E \mid S_1; S_2 \rangle)(\sigma) = \langle \sigma \rangle \hat{} \, \mathcal{COMP}(\langle E \mid S_1 \rangle)(\sigma) \hat{} \, \mathcal{COMP}(\langle E \mid S_2 \rangle)(\kappa(\mathcal{COMP}(\langle E \mid S_1 \rangle)(\sigma))).$$

The $\langle \sigma \rangle$ is motivated as to indicate the operation of splitting up $S_1; S_2$, or as a means to make induction arguments later on go through.

Adding nondeterminism necessitates $\mathcal{COMP}$ to yield for each R and σ not the corresponding computation sequence, but the set of computation sequences covering all possible alternatives depending on the different possible choices. We now define computation sequences and a set of rules to describe $\mathcal{COMP}$.

Definition 7 (Computation sequences). (a)

$$\Sigma^+ = \{\langle \sigma_1, \dots, \sigma_i, \dots, \sigma_n \rangle \mid \sigma_i \in \Sigma, 1 \leq i \leq n, n \in \mathbb{N}\},$$

$$\Sigma^\omega = \{\langle \sigma_1, \dots, \sigma_i, \dots \rangle \mid \sigma_i \in \Sigma, i \in \mathbb{N}\},$$

$$\Sigma^\infty = \{\varrho, \dots\} = \Sigma^+ \cup \Sigma^\infty.$$

Note, that the empty sequence is excluded.

(b) $\hat{}: \Sigma^\infty \times \Sigma^\infty \to \Sigma^\infty$, concatenation, is defined by

$$\langle \sigma_1, \dots, \sigma_m \rangle \hat{} \langle \sigma_{m+1}, \dots, \sigma_n \rangle = \langle \sigma_1, \dots, \sigma_m, \sigma_{m+1}, \dots, \sigma_n \rangle,$$

$$\langle \sigma_1, \dots, \sigma_m \rangle \hat{} \langle \sigma_{m+1}, \dots \rangle = \langle \sigma_1, \dots, \sigma_m, \sigma_{m+1}, \dots \rangle,$$

$$\langle \sigma_1, \dots \rangle \hat{} \varrho = \langle \sigma_1, \dots \rangle$$

with the extension $\varrho\hat{}\{\varrho_i \mid i\in I\} = \{\varrho\hat{}\varrho_i \mid i\in I\}$.

(c) $\kappa : \Sigma^\infty \to \Sigma$ is defined by

$$\kappa(\varrho) = \begin{cases} \text{last element of } \varrho & \text{if } \varrho\in\Sigma^+, \\ \bot & \text{otherwise} \end{cases}$$

with the extension: $\kappa(\{\varrho_i \mid i\in I\}) = \{\kappa(\varrho_i) \mid i\in I\}$.

(d)

$$\text{length}(R) = \begin{cases} n & \text{if } \varrho = \langle\sigma_1, \ldots, \sigma_n\rangle, \\ \infty & \text{otherwise.} \end{cases}$$

(e) ϱ' is initial segment of ϱ (i.s.o.) iff $\varrho = \varrho'\hat{}\varrho''$ or $\varrho = \varrho'$.

In the sequel, $P(\Sigma^\infty) = \{A \mid A \subset \Sigma^\infty\}$, the powerset of Σ^∞.

Definition 8 (Rules for generating computation sequences). $\mathcal{COMP} : \mathcal{PROG} \to (\Sigma \to \mathcal{P}(\Sigma^\infty))$ by: For all $R\in\mathcal{PROG}$, for $\sigma = \bot$, $\mathcal{COMP}(R)(\bot) = \{\langle\bot\rangle\}$, for $\sigma\in\Sigma_0$:

(i) $$\mathcal{COMP}(\langle E \mid x := t\rangle)(\sigma) = \{\langle\sigma\{\mathcal{V}(t)(\sigma)/x\}\rangle\},$$

(ii) $$\mathcal{COMP}(\langle E \mid S_1; S_2\rangle)(\sigma)$$
$$= \bigcup\{\langle\sigma\rangle\hat{}\varrho\hat{}\mathcal{COMP}(\langle E \mid S_2\rangle)(\kappa(\varrho)) \mid \varrho\in\mathcal{COMP}(\langle E \mid S_1\rangle)(\sigma)\},$$

(iii) $$\mathcal{COMP}(\langle E \mid S_1 \vee S_2\rangle)(\sigma)$$
$$= \langle\sigma\rangle\hat{}\mathcal{COMP}(\langle E \mid S_1\rangle)(\sigma) \cup \langle\sigma\rangle\hat{}\mathcal{COMP}(\langle E \mid S_2\rangle)(\sigma),$$

(iv) $$\mathcal{COMP}(\langle E \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}\rangle)(\sigma) = \text{if } \mathcal{W}(b)(\sigma)$$
$$\text{then } \langle\sigma\rangle\hat{}\mathcal{COMP}(\langle E \mid S_1\rangle)(\sigma) \text{ else } \langle\sigma\rangle\hat{}\mathcal{COMP}(\langle E \mid S_2\rangle)(\sigma) \text{ fi},$$

(v) $$\mathcal{COMP}(\langle E \mid P\rangle)(\sigma) = \langle\sigma\rangle\hat{}\mathcal{COMP}(\langle E \mid S_j\rangle)(\sigma),$$
$$\text{where } P \equiv P_j,\ P_j \Leftarrow S_j \text{ in } E.$$

Intuitively, these rules are sufficient to describe generating the set of computation sequences for given R and σ. However the concept 'generating' is too fuzzy to be mathematically satisfying. A well-known way out of this problem is to regard this set of rules as a set of equations, for which $\mathcal{COMP}$ should be a solution. From now on we take this approach: Definition 8 is regarded as a set of equations. Now it is clear that then a proof is required that a solution $\mathcal{COMP}$ exists, and moreover that it is

unique. For the deterministic case this id done in various ways by De Bruin in [6]. For the nondeterministic case we now show that an extra equation is needed to ensure uniqueness. We then prove the existence of a unique solution $\mathscr{COMP}$ by extending the techniques of [6]. Then finally we define the operation semantics.

The following examples show, that in general, Definition 8 regarded as a set of equations does not ensure a solution to be unique and provide intuition as to which kind of extra equation might solve this deficiency.

Example 1. $\mathscr{COMP}(\langle P \Leftarrow P \mid P\rangle)(\sigma)$. Intuitively, this should generate $\{\langle \sigma, \sigma, \ldots\rangle\}$. However, regarded as an equation, this program gives rise to

$$\mathscr{COMP}(\langle E \mid P\rangle)(\sigma) = \langle\sigma\rangle \hat{} \mathscr{COMP}(\langle E \mid P\rangle)(\sigma).$$

Now both $\{\langle \sigma, \sigma, \ldots\rangle\}$ and $\emptyset$ satisfy this equation, as Definition 7b implies $\varrho\hat{}\emptyset = \emptyset$, so uniqueness is violated. Both practice (the program will loop) and theory (the rules generate $\langle \sigma, \sigma, \ldots\rangle$) suggest preference for the first alternative.

The above example suggests the extra equation to be of the form $\mathscr{COMP}(R)(\sigma) \neq \emptyset$. However, the following example shows that a stronger requirement is needed.

Example 2.

$$\mathscr{COMP}(\langle P \Leftarrow x := x \vee P \mid P\rangle)(\sigma).$$

Intuitively, this should generate $\{\langle \sigma, \sigma, \sigma\rangle, \langle \sigma, \sigma, \sigma, \sigma, \sigma\rangle, \ldots, \langle \sigma, \sigma, \ldots\rangle\} = C$ respectively for $x := x$ chosen the first time possible, the second time, ..., never. However, regarded as an equation this program gives rise to

$$\begin{aligned}\mathscr{COMP}(\langle E \mid P\rangle)(\sigma) &= \langle\sigma\rangle \hat{} \mathscr{COMP}(\langle E \mid x := x \vee P\rangle)(\sigma)\\ &= \langle\sigma\rangle \hat{} \langle\sigma\rangle \hat{} \mathscr{COMP}(\langle E \mid x := x\rangle)(\sigma)\\ &\quad \cup \langle\sigma\rangle \hat{} \langle\sigma\rangle \hat{} \mathscr{COMP}(\langle E \mid P\rangle)(\sigma)\\ &= \{\langle \sigma, \sigma, \sigma\rangle\} \cup \langle \sigma, \sigma\rangle \hat{} \mathscr{COMP}(\langle E \mid P\rangle)(\sigma).\end{aligned}$$

Now both C and $C \setminus \{\langle \sigma, \sigma, \ldots\rangle\}$ satisfy this equation, so uniqueness of the solution is violated. Both practice and theory indicate which one should be preferred. Considering practice, a cycle that halts or is repeated according

to nondeterministic choice potentially can be repeated any finite amount of time, and also can be repeated forever. So this suggests preference for the first alternative. Considering theory, for the obvious representation of the set of computation sequences by trees, finite nondeterministic choice gives rise to finitely branching trees. By König's lemma then follows that a tree containing infinitely many finite branches, i.e. finite computation sequences, also contains an infinite one, so this also suggests preference for the first alternative.

The above examples suggest the entire equation to be of the form: $\mathscr{COMP}(R)(\sigma) \in \mathscr{G}$, where $\mathscr{G} = \{G \in \mathscr{P}(\Sigma^\infty) \mid G \neq \emptyset$, if $\langle \varrho_i \rangle_{i=1}^\infty$ such that

(i) for all i, $\varrho_i \in G$,
(ii) for all i, ϱ_i i.s.o. ϱ_{i+1},
(iii) $\sup_i\{\text{length}(\varrho_i)\} = \infty$,

then $\exists \varrho \in G$ such that for all i, ϱ_i, i.s.o. $\varrho\}$.

However, the following example shows that an even more subtle requirement is needed.

Example 3.

$$\mathscr{COMP}(\langle P \Leftarrow x := 1 \vee P \mid P \rangle)(\sigma).$$

Intuitively, this should generate $\{\langle \sigma, \sigma, \sigma\{1/x\}\rangle, \langle \sigma, \sigma, \sigma, \sigma, \sigma\{1/x\}\rangle, \ldots, \langle \sigma, \sigma, \ldots\rangle\} = C'$. However, regarded as an equation, like in Example 2, $C' \setminus \{\langle \sigma, \sigma, \ldots\rangle\}$ is also possible. Again, the first alternative is to be preferred.

This example suggests the following strengthening of the above chosen requirement described by $\mathscr{G}$.

Definition 9. $\mathscr{H} = \{H \in \mathscr{P}(\Sigma^\infty) \mid H \neq \emptyset$, if $\langle \varrho_i \rangle_{i=1}^\infty$, $\varrho_i \in \Sigma^\infty$, such that

(i) for all i, $\exists \varrho_i' \in H$ such that ϱ_i i.s.o. ϱ_i',
(ii) for all i, ϱ_i i.s.o. ϱ_{i+1},
(iii) $\sup\{\text{length}(\varrho_i)\} = \infty$,

then $\exists \varrho \in H$ such that for all i, ϱ_i i.s.o. $\varrho\}$.

Remark. In the different setting of unbounded nondeterminism, this is the closedness property to be found in [3].

We claim that the following extension of Definition 8 ensures the existence of a unique solution.

Definition 10. $\mathscr{COMP}: \mathscr{PROG} \to (\Sigma \to \mathscr{H})$ is defined by the following set of equations:

(a) The equations of Definition 8.

(b) For all $R \in \mathscr{PROG}$, $\sigma \in \Sigma$, $\mathscr{COMP}(R)(\sigma) \in \mathscr{H}$.

In De Bruin [6] for the deterministic case four methods to prove the existence of a unique solution are presented. We have chosen to adapt to our case the most topological one, as this seems the best one to extend to sets of sequences. The idea is the following. Consider the set of functions $\mathscr{PROG} \to (\Sigma \to \mathscr{H})$; the solution $\mathscr{COMP}$ we seek to find is, if it exists, an element of this set. Now as the left parts of the equations in part (a) of Definition 10 all contain only $\mathscr{COMP}(R)(\sigma)$, a solution of this set of equations can be interpreted as a fixed point of an endomorphic operator on $\mathscr{PROG} \to (\Sigma \to \mathscr{H})$ defined directly by these equations (cf. Definition 20). To ensure existence of a unique fixed point, from topology it is known that it is sufficient that firstly the space is complete metric, i.e. a space with a metric distance function defined on it such that every Cauchy sequence converges, and secondly, that the operator is a contraction mapping, i.e. the distance between the image of any two points is less than or equal to the distance between the original points multiplied by a fixed constant smaller than 1.

The operator as well as the elements of the domain are given: respectively by the equations and by the type of $\mathscr{COMP}$. Left to choose is the metric. As usual, we choose the distance between two functions to be the supremum over the elements in the domain of the distance between the two images of such an element. To do so, we first define a distance between computation sequences, next between sets of them and finally between the functions. All of these will have to be complete metrics.

We start by considering computation sequences, i.e. Σ^∞. A natural distance is the following.

Definition 11.

$$\varrho[j] = \begin{cases} \varrho & \text{if } \varrho = \langle \sigma_1, \ldots, \sigma_n \rangle \text{ and } j \geq n, \\ \langle \sigma_1, \ldots, \sigma_j \rangle & \text{otherwise.} \end{cases}$$

Definition 12. Distance on Σ^∞

$$\bar{\bar{d}}(\varrho_1, \varrho_2) = \begin{cases} 2^{-k} & \text{if } k = \sup\{j \mid \varrho_1[j] = \varrho_2[j]\} < \infty, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 13. For a Cauchy sequence $\langle \varrho_i \rangle_{i=1}^\infty$ define the limit $\lim_{i\to\infty} \varrho_i$ as follows. As $\langle \varrho_i \rangle_{i=1}^\infty$ has the Cauchy property, $\forall \varepsilon > 0\, \exists N_\varepsilon\, \forall l, m \geq N_\varepsilon\, d(\varrho_l, \varrho_m) < \varepsilon$, or, equivalently, $\forall k \in \mathbb{N}\, \exists N_k\, \forall l, m \geq N_k\, \bar{\bar{d}}(\varrho_l, \varrho_m) < 2^{-k}$.

By Definition 12 this implies $\forall k \in \mathbb{N}\, \exists N_k\, \forall l, m \geq N_k\, \varrho_{N_k}[k] = \varrho_l[k] = \varrho_m[k]$. Now define $\lim_{i\to\infty} \varrho_i$ by $(\lim_{i\to\infty} \varrho_i)[k] = \varrho_{N_k}[k]$.

Lemma 1. *$(\Sigma^\infty, \bar{\bar{d}})$ is a complete metric space.*

Proof. $\bar{\bar{d}}$ evidently is a metric. $\bar{\bar{d}}$ is complete iff every Cauchy sequence converges. Clearly, every Cauchy sequence $\langle \varrho_i \rangle_{i=1}^\infty$ converges to $\lim_{i\to\infty} \varrho_i$.

Next, we proceed to sets of computation sequences. Note, that defining the distance $\bar{\bar{d}}$ enables us to give a much easier definition of $\mathscr{H}$.

Lemma 2. *$\mathscr{H} = \{H \in \mathscr{P}(\Sigma^\infty) \mid H \neq \emptyset$, for each Cauchy sequence $\langle \varrho_i \rangle_{i=1}^\infty$ in H, $\lim_{i\to\infty} \varrho_i \in H\}$.*

Proof. Evident by Definitions 9, 12 and 13.

Remark. (1) Here the topological approach allows an easier solution of the problem than the cpo approach, where it is more difficult to handle cases like Example 3 as may be seen by the difference between the two definitions of $\mathscr{H}$.

(2) For Σ^∞ with the topology $\mathscr{J}(\bar{\bar{d}})$ induced by $\bar{\bar{d}}$, $\mathscr{H}$ can be defined by

$$\mathscr{H} = \{H \in \mathscr{P}(\Sigma^\infty) \mid H \neq \emptyset, H \text{ closed in } \mathscr{J}(\bar{\bar{d}})\}.$$

A natural distance on $\mathscr{H}$ is defined as follows.

Definition 14.

$$H[j] = \{\varrho[j] \mid \varrho \in H\}.$$

Definition 15. Distance on $\mathscr{H}$.

$$\bar{d}(H_1, H_2) = \begin{cases} 2^{-k} & \text{if } k = \sup\{j \mid H_1[j] = H_2[j]\} < \infty, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 16. For a Cauchy sequence $\langle H_i \rangle_{i=1}^{\infty}$ define the limit $\lim_{i\to\infty} H_i$ as follows. As $\langle H_i \rangle_{i=1}^{\infty}$ has the Cauchy property, $\forall \varepsilon > 0\ \exists N_\varepsilon\ \forall l, m \geq N_\varepsilon\ \bar{d}(H_l, H_m) < \varepsilon$, or, equivalently, $\forall k \in \mathbb{N}\ \exists N_k\ \forall l, m \geq N_k\ \bar{d}(H_l, H_m) < 2^{-k}$. By definition 15 this implies $\forall k \in \mathbb{N}\ \exists N_k\ \forall l,\ m \geq N_k\ H_{N_k}[k] = H_k[k] = H_m[k]$. Now define $\lim_{i\to\infty} H_i$ as follows (using Lemma 2).

$$\lim_{i\to\infty} H_i = \bigcup_{k=1}^{\infty} \{\varrho \in H_{N_k}[k] \mid \varrho = \langle \sigma_1, \ldots, \sigma_n \rangle, n < k\}$$

$$\cup \left\{ \lim_{k\to\infty} \varrho_k \mid \langle \varrho_k \rangle_{k=1}^{\infty} \text{ Cauchy sequence, } \varrho_k \in H_{N_k}[k] \right\}.$$

Lemma 3. *$(\mathscr{H}, \bar{d})$ is a complete metric space.*

Proof. The first requirement to be a metric space is $\bar{d}(H_1, H_2) = 0 \Leftrightarrow H_1 = H_2$. Let $\bar{d}(H_1, H_2) = 0$, $\varrho \in H_1$. If $\varrho \in \Sigma^+$, then $\exists j\, \varrho = \varrho[j] = \varrho[j+1]$. As $\bar{d}(H_1, H_2) = 0$, $\varrho = \varrho[j] = \varrho[j+1] \in H_2[j+1]$. Consequently, $\varrho \in H_2$. If $\varrho \in \Sigma^\omega$, then either $\varrho \in H_2$ or $\varrho[i] \in H_2[i]$, $i = 1, 2, \ldots$. In the latter case, there exist $\varrho_i' \in H_2$, $i = 1, 2, \ldots$, such that $\varrho[i] = \varrho'[i]$, $i = 1, 2, \ldots$. Now clearly $(\varrho_i')_{i=1}^{\infty}$ is a Cauchy sequence in H_2, and by Definition 13 $\lim_{i\to\infty} \varrho_i' = \varrho$. Consequently, (by Lemma 2) $\varrho \in H_2$. Conversely let $H_1 = H_2$. Then $\forall j\, H_1[j] = H_2[j]$, so $d(H_1, H_2) = 0$. The other requirements of being a metric space are evidently fulfilled. $\bar{d}$ is complete iff every Cauchy sequence converges. Let $\langle H_i \rangle_{i=1}^{\infty}$ be a Cauchy sequence. By Definition 15, $\langle H_i \rangle_{i=1}^{\infty}$ clearly converges to $\lim_{i\to\infty} H_i$, by Definition 16, clearly $\lim_{i\to\infty} H_i \in \mathscr{H}$.

Remark. The reasons to restrict $\mathscr{P}(\Sigma^\infty)$ to $\mathscr{H}$ in the beginning of this section that did arise when regarding Definition 8 as a set of equations here have their topological counterpart: Should distance $\bar{d}$ be defined on $\mathscr{P}(\Sigma^\infty)$ instead of $\mathscr{H}$, then the sets C and $C \setminus \{\langle \sigma, \sigma, \ldots \rangle\}$ of Example 2 (and likewise C' and $C' \setminus \{\langle \sigma, \sigma, \ldots \rangle\}$ of Example 3) would have distance 0 but not be equal.

This violates the metric requirement $\bar{d}(C_1, C_2) = 0 \Leftrightarrow C_1 = C_2$. Now disregard knowledge of the previously defined restriction of $\mathscr{P}(\Sigma^\infty)$ to $\mathscr{H}$

caused by ambiguities with regard to solutions of the equations in Definition 1 and indicated by the Examples 1–3. (Note, that at that stage no distance between sets was even defined.) A natural solution of the present problem then, is the following.

Restrict $\mathscr{P}(\Sigma^\infty)$ to only those subsets of Σ^∞, that contain their limit points in the topology induced by $\bar{d}$. Lemma 3 states that this solves the problem. Not surprisingly, the tree-likeness requirement stated in $\mathscr{H}$ is equivalent to this restriction, as stated in Lemma 2.

By now, we arrive at our first aim, turning $\mathscr{PROG}\to(\Sigma\to\mathscr{H})$ into a complete metric space by defining the following natural distance.

Definition 17.

$$(\phi\in)C=\mathscr{PROG}\to(\Sigma\to\mathscr{H}).$$

Definition 18. Distance on C.

$$d(\phi_1,\phi_2)=\sup_{R,\sigma}\{\bar{d}(\phi_1(R)(\sigma),\phi_2(R)(\sigma))\}.$$

Definition 19. For a Cauchy sequence $\langle\phi_i\rangle_{i=1}^{\infty}$ define the limit $\lim_{i\to\infty}\phi_i$ as follows. As $\langle\phi_i\rangle_{i=1}^{\infty}$ has the Cauchy property, $\forall\varepsilon>0\,\exists N_\varepsilon\,\forall l,\ m\geq N_\varepsilon\, d(\phi_l,\phi_m)<\varepsilon$. By Definition 18 holds

$$\forall\varepsilon>0\,\exists N_\varepsilon\,\forall l,\ m\geq N_\varepsilon\,\forall R\,\forall\sigma\, d(\phi_l(R)(\sigma),\phi_m(R)(\sigma))<\varepsilon.$$

Then $\forall R\,\forall\sigma$, $\langle\phi_i(R)(\sigma)\rangle_{i=1}^{\infty}$ is a Cauchy sequence. By Lemma 3, $\forall R\,\forall\sigma$ $\langle\phi_i(R)(\sigma)\rangle_{i=1}^{\infty}$ converges to $\lim_{i\to\infty}\phi_i(R)(\sigma)$. Now define $\lim_{i\to\infty}\phi_i$ as follows.

$$\forall R\,\forall\sigma\quad\left(\lim_{i\to\infty}\phi_i\right)(R)(\sigma)=\lim_{i\to\infty}(\phi_i(R)(\sigma)).$$

Lemma 4. *(C,d) is a complete metric space.*

Proof. By standard techniques, e.g. see [8, Chapter 14, Theorem 2.6].

Definition 20. $\Phi: C\to C$ is defined by

$$\Phi=\lambda\phi\cdot\lambda R\cdot\lambda\sigma\cdot\qquad\sigma=\bot\to\bot,$$

$\sigma \in \Sigma_0$,

$$R \equiv \langle E \mid x := t \rangle \to \{\langle \{\mathscr{V}(t)(\sigma)/x\}\langle\},$$

$$R \equiv \langle E \mid S_1; S_2 \rangle$$

$$\cup \{\langle \sigma \rangle^{\smile}.\varrho^{\smile}.\phi(\langle E \mid S_2\rangle)(\kappa(\varrho)) \mid \varrho \in \phi(\langle E \mid S_1\rangle)(\sigma)\}$$

$$R \equiv \langle E \mid S_1 \vee S_2\rangle \to \langle \sigma \rangle^{\frown}\phi(\langle E \mid S_1\rangle)(\sigma) \cup \langle \sigma \rangle^{\frown}\phi(\langle E \mid S_2\rangle)(\sigma),$$

$$R \equiv \langle E \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}\rangle$$

$$\to \text{if } W(b)(\sigma) \text{ then } \langle \sigma \rangle^{\frown}\phi(\langle E \mid S_1\rangle)(\sigma)$$

$$\text{else } \langle \sigma \rangle^{\frown}\phi(\langle E \mid S_2\rangle)(\sigma),$$

$$R \equiv \langle E \mid P\rangle \to \langle \sigma \rangle^{\frown}\phi(\langle E \mid S_j\rangle), \text{ where } P \equiv P_j,\ P_j \Leftarrow S_j \text{ in } E.$$

Note, that Φ is well defined, i.e. $\forall \phi\ \Phi(\phi) \in C$, as can be easily seen from the definition.

Lemma 5. *Φ is a contraction mapping, namely* $\forall \phi_1, \phi_2 d(\Phi(\phi_1), \Phi(\phi_2)) \leq \frac{1}{2} d(\phi_1, \phi_2)$.

Proof. Each of the following cases is trivial for $\sigma = \perp$, so from here on $\sigma \in \Sigma_0$.

Case 1: $R \equiv \langle E \mid x := t\rangle$. By Definition 20

$$\forall \phi_1, \phi_2 \forall \sigma \quad \Phi(\phi_1)(R)(\sigma) = \langle \sigma\{\mathscr{V}(t)(\sigma)/x\}\rangle\} = \Phi(\phi_2)(R)(\sigma).$$

So by Definitions 15 and 18, $\bar{d}(\Phi(\phi_1), \Phi(\phi_2)) = 0 \leq \frac{1}{2}\bar{d}(\phi_1, \phi_2)$.

Case 2: $R \not\equiv \langle E \mid x := t\rangle$, $R \not\equiv \langle E \mid S_1; S_2\rangle$. By Definition 20, $\forall \phi\ \forall \sigma$ $\Phi(\phi)(R)(\sigma) = \langle \sigma \rangle^{\frown}\phi(R')(\sigma)$, R' as given by the right hand part of Definition 20.

So

$$d(\Phi(\phi_1), \Phi(\phi_2)) = \sup_{R,\sigma}\{\bar{d}(\Phi(\phi_1)(R)(\sigma), \Phi(\phi_2)(R)(\sigma))\}$$

by Definition 18

$$= \sup_{R,\sigma}\{\bar{d}(\langle \sigma \rangle^{\frown}\phi_1(R')(\sigma), \langle \sigma \rangle^{\frown}\phi_2(R')(\sigma))\}$$

$$= \tfrac{1}{2} \sup_{R,\sigma}\{\bar{d}(\phi_1(R')(\sigma), \phi_2(R')(\sigma))\}$$

by Definition 15

$$= \tfrac{1}{2} d(\phi_1, \phi_2) \quad \text{by Definition 18.}$$

Case 3. $R \equiv \langle E \mid S_1; S_2 \rangle$, analogously to Case 2.

By now we can, by using well-known topology, justify our claim made above.

Lemma 6. *Φ has a unique fixpoint.*

Proof. By Lemmas 4 and 5, using standard techniques from topology. Cf. [8, p. 305] and [6, p. 27].

Theorem 1. *The set of equations of Definition* 10 *has a unique solution* $\mathscr{COMP}$.

Proof. Directly from Definitions 10 and 20 and Lemma 6.

Finally, having justified the definition of $\mathscr{COMP}$, we define the operational semantics.

Definition 21. Operational semantics. $\mathscr{O} : \mathscr{PROG} \to (\Sigma \to \Sigma)$ is defined by: For all R, for all σ, $\mathscr{O}(R)(\sigma) = \kappa(\mathscr{COMP}(R)(\sigma))$.

For later use we here state the following lemma.

Lemma 7.

(i) $\mathscr{O}(\langle E \mid S_1; S_2 \rangle)(\sigma) = \mathscr{O}(\langle E \mid S_2 \rangle) \circ \mathscr{O}(\langle E \mid S_1 \rangle)(\sigma)$,

(ii) $\mathscr{O}(\langle E \mid S_1 \vee S_2 \rangle)(\sigma) = \mathscr{O}(\langle E \mid S_1 \rangle)(\sigma) \cup \mathscr{O}(\langle E \mid S_2 \rangle)(\sigma)$,

(iii) $\mathscr{O}(\langle E \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \rangle)(\sigma) = \text{if } \mathscr{W}(b)(\sigma)$
then $\mathscr{O}(\langle E \mid S_1 \rangle)(\sigma)$ else $\mathscr{O}(\langle E \mid S_2 \rangle)(\sigma)$ fi,

(iv) $\mathscr{O}(\langle E \mid P \rangle)(\sigma) = \mathscr{O}(\langle E \mid S_j \rangle)(\sigma)$, where $P \equiv P_j$, $P_j \Leftarrow S_j$ in E.

Proof. Evident from Definitions 10 and 21.

3. The Denotational Semantics

We here present the denotational semantics for which the operational one, treated in Section 2, was designed to serve as an intuitive counterpart.

The method used is the fixed point approach in a cpo setting, as can be found in [15]. The denotational semantics we use greatly resembles the one in [5, Chapters 5 and 7], so only a very concise treatment is given, just defining the notions and stating the results we need for the equivalence proof in Section 4.

We start by defining a domain, consisting of a selection of subsets of $\mathscr{P}(\Sigma)$ with the Egli-Milner ordering, cf. [10]. Note the resemblance to the domain of results in Section 2, with regard to $\mathscr{H}$ being the outcome domain of $\mathscr{COMP}$.

Definition 22.

$$(\tau\in)T=\{\tau\in\mathscr{P}(\Sigma)\mid \tau \text{ finite or } \bot\in\tau\}.$$

Definition 23 (Egli-Milner ordering).

$$\tau_1\sqsubseteq\tau_2 \quad \text{iff } \bot\in\tau_1 \text{ and } \tau_1\setminus\{\bot\}\subseteq\tau_2$$
$$\text{or } \bot\notin\tau_1 \text{ and } \tau_1=\tau_2.$$

Lemma 8. *$(T,\sqsubseteq)$ is a cpo.*

We now give the domain of strict functions $\Sigma\to_s T$.

Definition 24. $\psi:\Sigma\to_s T$, i.e. ψ is strict iff $\psi(\bot)=\{\bot\}$.

Definition 25.

$$(\psi\in)M=\Sigma\to_s T$$

with the extension: For each $\psi:\Sigma\to_s T$, $\hat{\psi}:T\to_s T$ by $\hat{\psi}=\lambda\tau\cdot\bigcup_{\sigma\in\tau}\psi(\sigma)$ and

$$\psi_1\circ\psi_2=\lambda\sigma\cdot\hat{\psi}_1(\psi_2(\sigma)),$$
$$\psi_1\cup\psi_2=\lambda\sigma\cdot(\psi_1(\sigma)\cup\psi_2(\sigma)).$$

Definition 26. $\psi_1\sqsubseteq\psi_2$ iff for all $\sigma\in\Sigma$, $\psi_1(\sigma)\sqsubseteq\psi_2(\sigma)$.

Lemma 9. *$(M,\sqsubseteq)$ is a cpo.*

We now introduce $\gamma\in\Gamma$, where γ gives meaning to procedure variables; furthermore we define variant of γ.

Definition 27. $(\gamma \in)\Gamma = \mathscr{P}_{\mathscr{VAR}} \to M$.

Definition 28.

$$\gamma\{\psi/P\}(P') = \begin{cases} \psi & \text{if } P' \equiv P, \\ \gamma(P') & \text{otherwise.} \end{cases}$$

The following is needed from the theory of cpo's.

Definition 29. $(C, \sqsubseteq)$ cpo, $f: C \to C$.
(a) x is a fixed point of f iff $f(x) = x$.
(b) x is the least fixed point μf of f iff:
(i) x is a fixed point of f;
(ii) for all y, y fixed point of f, $x \sqsubseteq y$.

Definition 30. $(C, \sqsubseteq)$, $(C', \sqsubseteq)$ cpo; $f: C \to C'$ is continuous iff:
(i) $x_1 \sqsubseteq x_2 \Rightarrow f(x_1) \sqsubseteq' f(x_2)$ (monotonicity);
(ii) for each chain $\langle x_i \rangle_{i=0}^{\infty}$ in C,

$$f\left(\bigsqcup_{i=0}^{\infty} x_i\right) = \bigsqcup_{i=0}^{\infty} f(x_i).$$

Notation: $f \in [C \to C']$.

Lemma 10. *C_i cpo, $f_i \in [C^n \to C]$, $i = 1, \ldots, n$*

$$\langle f_1, \ldots, f_n \rangle \in [C^n \to C^n]$$

by

$$\langle f_1, \ldots, f_n \rangle(\langle c_1, \ldots, c_n \rangle) = \langle f_1(\langle c_1, \ldots, c_n \rangle), \ldots, f_n(\langle c_1, \ldots, c_n \rangle) \rangle.$$

Then

$$\mu\langle f_1, \ldots, f_n \rangle = \bigsqcup_{k=1}^{\infty} \langle f_1, \ldots, f_n \rangle^k (\langle \bot, \ldots, \bot \rangle).$$

After these preparations we define the denotational semantics as follows.

Definition 31 (Denotational semantics). (a) $\mathscr{N} : \mathscr{STAT} \to (\Gamma \to M)$ is defined by

(i) $$\mathscr{N}(x := t)(\gamma) = \lambda\sigma \cdot \{\sigma\{\mathscr{V}(t)(\sigma)/x\}\},$$

(ii) $\mathcal{N}(S_1; S_2)(\gamma) = \mathcal{N}(S_2)(\gamma) \circ \mathcal{N}(S_1)(\gamma)$,

(iii) $\mathcal{N}(S_1 \vee S_2)(\gamma) = \mathcal{N}(S_1)(\gamma) \cup \mathcal{N}(S_2)(\gamma)$,

(iv) $\mathcal{N}$ (if b then S_1 else S_2 fi)$(\gamma) = \lambda\sigma \cdot$ if $\mathcal{W}(b)(\sigma)$ then $\mathcal{N}(S_1)(\gamma)(\sigma)$ else $\mathcal{N}(S_2)(\gamma)(\sigma)$ fi.

(b) $\mathcal{M} : \mathcal{PROG} \to (\Gamma \to M)$ is defined by

$$\mathcal{M}(\langle E \mid S\rangle)(\gamma) = \mathcal{N}(S)(\gamma\{\psi_i/P_i\}_{i=1}^n),$$

where $\langle \psi_1, \ldots, \psi_n\rangle = \mu\langle \Psi_1, \ldots, \Psi_n\rangle$ and

$$\Psi_j = \lambda\psi_1', \ldots, \lambda\psi_n' \mathcal{N}(S_j)(\gamma\{\psi_i'/P_i\}_{i=1}^n),$$

$j = 1, \ldots, n$.

Lemma 11.

$$\lambda\psi_1', \ldots, \lambda\psi_n' \cdot \mathcal{N}(S_j)(\gamma\{\psi_i'/P_i\}_{i=1}^n) \in [M^n \to M], \quad j = 1, \ldots, n.$$

Theorem 2. *$\mathcal{M}$ is well defined.*

Proof. Essentially from Lemmas 10 and 11.

For later use, in Section 4, we here state the following lemmas.

Lemma 12. *$\lambda\sigma \cdot \mathcal{N}(S)(\gamma)(\sigma)$ is monotone.*

Lemma 13. *$\mathcal{M}(\langle E \mid P\rangle) = \mathcal{M}(\langle E \mid S_j\rangle)$, where $P \equiv P_j$, $P_j \Leftarrow S_j$ in E.*

4. The Equivalence of the Operational and the Denotational Semantics

The set-up of the equivalence proof for the two semantics defined in the foregoing sections, i.e. $\mathcal{O}(R) = \mathcal{M}(R)(\gamma)$, is as follows.

A natural way to proceed might seem to apply induction on the length of individual computation sequences in $\mathcal{COMP}(R)(\sigma)$ proving $\sigma' \in \mathcal{O}(R)(\sigma) \Rightarrow \sigma' \in \mathcal{M}(R)(\gamma)(\sigma)$. However, it is only possible to prove this for σ' such that $\mathcal{COMP}(R)(\sigma) \in \mathcal{P}(\Sigma^+)$. Namely, if there is an infinite computation sequence in $\mathcal{COMP}(R)(\sigma)$, then $\perp \in \mathcal{O}(R)(\sigma)$, as can be directly inferred from Definitions 7 and 21. It is by no means clear, that in this case also

$\perp \in \mathcal{M}(R)(\gamma)(\sigma)$, as the concept of computation sequence belongs to the realm of operational semantics. So using set inclusion $\mathcal{O}(R)(\sigma) \subseteq \mathcal{M}(R)(\gamma)(\sigma)$ is not feasible. Choosing the Egli-Milner ordering $\mathcal{O}(R)(\sigma) \sqsubseteq \mathcal{M}(R)(\gamma)(\sigma)$ with this induction is also impossible, as for this ordering it is required to prove $\mathcal{O}(R)(\sigma) = \mathcal{M}(R)(\gamma)(\sigma)$ if $\perp \notin \mathcal{O}(R)(\sigma)$.

The way out we have chosen is to apply, in case $\mathcal{COMP}(R)(\sigma) \in \mathcal{P}(\Sigma^+)$, induction on the sum of the lengths of the computation sequences in $\mathcal{COMP}(R)(\sigma)$, thus proving $\mathcal{O}(R)(\sigma) = \mathcal{M}(R)(\gamma)(\sigma)$ in this case. In case there is an infinite computation sequence in $\mathcal{COMP}(R)(\sigma)$, and so, by Definition 21, $\perp \in \mathcal{O}(R)(\sigma)$, we prove $\mathcal{O}(R)(\sigma) \setminus \{\perp\} \subseteq \mathcal{M}(R)(\gamma)(\sigma)$ elementwise by the above mentioned induction on the length of individual computation sequences. Thus we yield $\mathcal{O}(R) \sqsubseteq \mathcal{M}(R)(\gamma)$. Proving $\mathcal{M}(R)(\gamma) \sqsubseteq \mathcal{O}(R)$ by standard techniques then completes the proof.

In order to apply induction to the sum of the lengths of the computation sequences in case $\mathcal{COMP}(R)(\sigma) \in \mathcal{P}(\Sigma^+)$ we have to prove that this sum is finite. This is made explicit by a careful application of an analogue of König's lemma. One of the well-known formulations of König's lemma is the following:

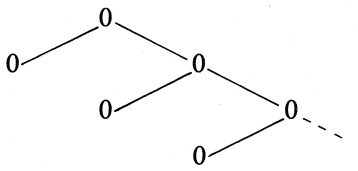
Lemma (König's). *A finitely branching tree where all branches are of finite length contains only finitely many nodes.*

As we work in the realm of sets of (computation) sequences instead of trees, we want to restate this lemma using these notions. Restate "finitely branching" by "there are only finitely many different initial segments of length n, for all $n \in \mathbb{N}$", "all branches are of finite length" by "a set of finite sequences", and finally "finitely many nodes" by "finitely many different sequences".

So the analogue to König's lemma seems to be

If in a set of finite sequences there are only finitely many different initial segments of length n, for all $n \in \mathbb{N}$, then there are only finitely many different sequences.

Now this is not true! Counter example: $\{\langle 0\rangle, \langle 0,0\rangle, \ldots\}$. The reason for this is, that the tree structure does not allow $\{\langle 0\rangle, \langle 0,0\rangle, \ldots\}$ as a set of branches in a finitely branching tree but forces to add $\langle 0,0,\ldots\rangle$:



For a set of finite sequences this is not the case. So an extra requirement of such a set is to be added. Not surprisingly, taking the set to have a property analogeous to $\mathscr{H}$ for computation sequences is sufficient, as this reflected the tree-like way in which $\mathscr{COMP}$ generated a 'set of computation sequences'.

We now give the analogue of König's lemma.

Lemma 14. *If in a set C of finite sequences $\{r, \ldots\}$, $r = \langle s_1, s_2, \ldots, s_n \rangle$, $n \in \mathbb{N}$, there are only finitely many different initial segments of length n for all $n \in \mathbb{N}$, and C has the following property: C is tree-like i.e. if there is a row of sequences $\langle r_i' \rangle_{i=1}^{\infty}$, not necessarily $r_i' \in C$, such that*

(i) *for all i, $\exists\, r_i \in C$: r_i' i.s.o. r_i,*

(ii) *for all i, r_i' i.s.o. r_{i+1}',*

(iii) $\sup_i \{\text{length}(r_i')\} = \infty$,

then $\lim_{i \to \infty} r_i' \in C$, then there are only finitely many sequences in C.

Proof. By contradiction. Suppose there are infinitely many different sequences. Let $G(n) = \{r \mid \text{length}(r) = n,$ i.s.o. infinitely many different sequences$\}$. We show by induction that $G(n) \neq \emptyset$ for all $n \in \mathbb{N}$. Induction basis: To prove $G(1) \neq \emptyset$. As there are only finitely many different initial segments of length 1 and infinitely many different sequences, $G(1) \neq \emptyset$. Induction step: To prove $G(k+1) \neq \emptyset$. As there are infinitely many different sequences but only finitely many different initial segments of length k or $k+1$, and $G(k) \neq \emptyset$ (Ind. hyp.), $G(k+1) \neq \emptyset$. So $G(n) \neq \emptyset$ for all $n \in \mathbb{N}$. Now clearly, for all $n \in \mathbb{N}$ every element of $G(n)$ is initial segment of at least one of the elements of $G(n+1)$. So by the axiom of choice there are $\bar{r}_i \in G(i)$ such that $\bar{r}_i$ i.s.o. $\bar{r}_{i+1}$, $i = 1, 2, \ldots$. As C is tree-like, this implies that there is an infinite sequence in C. Contradiction.

Remark. Note that the property 'tree-like' had to be brought to the surface on three fully independent occasions where it was more or less hidden in the structure of the concepts under consideration:

(1) In Definition 10 to select tree-like solutions of the equations.

(2) In Definition 15, restricting the distance $\bar{d}$ to a space consisting of only tree-like sets.

(3) In Lemma 14 to select sets sufficiently tree-like to prove an analogue of König's lemma for them.

We now show, that for all R and all σ, $\mathscr{COMP}(R)(\sigma)$ satisfies the requirements of Lemma 14. The only requirement left to prove is, that $\mathscr{COMP}(R)(\sigma)$ gives only rise to finitely many different initial segments. This is done in the following lemma.

Lemma 15. *For all R and all σ the following holds for $\mathscr{COMP}(R)(\sigma)$: There are only finitely many different initial segments of length n, for all $n \in \mathbb{N}$, in $\mathscr{COMP}(R)(\sigma)$.*

Proof. Let $R \equiv \langle E \mid S \rangle$. Proof by cases, applying induction on the length of the initial segment. Let

$$I(n)(\mathscr{COMP}(R)(\sigma)) = \{\varrho' \mid \varrho' \text{ i.s.o. } \varrho \in \mathscr{COMP}(R)(\sigma), \text{length}(\varrho') = n\}.$$

Induction basis: To prove $\#(I(1)(\mathscr{COMP}(R)(\sigma))) < \infty$.

By cases:

(i) $S \equiv x := t$. Then $\mathscr{COMP}(R)(\sigma) = \{\langle \sigma\{\mathscr{V}(t)(\sigma)/x\}\rangle\}$. Consequently, $\#(I(1)(\mathscr{COMP}(R)(\sigma))) = 1 < \infty$.

(ii) $S \equiv S_1; S_2$. Then

$$\mathscr{COMP}(R)(\sigma) = \bigcup\{\langle\sigma\rangle \hat{} \varrho \hat{} \mathscr{COMP}(\langle E \mid S_2\rangle)(\kappa(\varrho)) \mid \varrho \in \mathscr{COMP}(\langle E \mid S_1\rangle)(\sigma)\},$$

so $I(1)(\mathscr{COMP}(R)(\sigma)) = \{\langle\sigma\rangle\}$. Consequently, $\#(I(1)(\mathscr{COMP}(R)(\sigma))) = 1 < \infty$.

Cases (iii), (iv) and (v) of Definition 10 analogeously to (ii) lead to $\#(I(1)(\mathscr{COMP}(R)(\sigma)) = 1 < \infty$. Induction hypothesis: Assume

$$\#(I(l)(\mathscr{COMP}(R)(\sigma))) < \infty,$$

for $1 \leq l \leq k$. Induction step: To prove $\#(I(k+1)(\mathscr{COMP}(R)(\sigma))) < \infty$.

By cases:

(i) $S \equiv x := t$. Then $I(k+1)(\mathscr{COMP}(R)(\sigma)) = I(k+1)(\{\langle\sigma\{\mathscr{V}(t)(\sigma)/x\}\rangle\}) = \emptyset$. Consequently, $\#(I(k+1)(\mathscr{COMP}(R)(\sigma))) = 0 < \infty$.

(ii) $S \equiv S_1; S_2$. Then

$$\mathcal{COMP}(R)(\sigma) = \{\langle\sigma\rangle \hat{} \varrho \hat{} \mathcal{COMP}(\langle E \mid S_2\rangle)(\kappa(\varrho)) \mid \varrho \in \mathcal{COMP}(\langle E \mid S_1\rangle)(\sigma)\}.$$

Consequently,

$$\#(I(k+1)(\mathcal{COMP}(R)(\sigma))) = \#(I(k)(\mathcal{COMP}(\langle E \mid S_1\rangle)(\sigma))) + \Sigma\{\#(I(k+1-(1+\text{length}(\varrho)))(\mathcal{COMP}(\langle E \mid S_2\rangle)(\kappa(\varrho)))) \mid \varrho \in \mathcal{COMP}(\langle E \mid S_1\rangle)(\sigma), \text{length}(\varrho) < k\} < \infty \quad \text{(Ind. hyp.)}$$

Cases (iii), (iv) and (v) of Definition 10 analogously to (ii) lead to $\#(I(k+1)(\mathcal{COMP}(R)(\sigma))) < \infty$. So $\#(I(n)(\mathcal{COMP}(R)(\sigma))) < \infty$ for all R, all σ, all $n \in \mathbb{N}$.

After these preparations, we can state Lemma 16, which enables us to apply induction on the sum of the lengths of the computation sequences in $\mathcal{COMP}(R)(\sigma)$ in case $\mathcal{COMP}(R)(\sigma) \in \mathcal{P}(\Sigma^+)$.

Lemma 16. *For all R and all σ for which $\mathcal{COMP}(R)(\sigma) \in \mathcal{P}(\Sigma^+)$, $\mathcal{COMP}(R)(\sigma)$ is a finite set.*

Proof. It is given that all computation sequences in $\mathcal{COMP}(R)(\sigma)$ are finite. By Lemma 15 there are only finitely many different initial segments of length n, for all $n \in \mathbb{N}$. By Definition 10(b), and Definition 9, $\mathcal{COMP}(R)(\sigma)$ has the tree-like property as required in Lemma 14. Consequently, by Lemma 14, $\mathcal{COMP}(R)(\sigma)$ in this case is finite.

Finally, we arrive at the main theorem of this chapter, stating equivalence of $\mathcal{O}(R)$ and $\mathcal{M}(R)(\gamma)$.

Definition 32. For

$$A \in \mathcal{P}(\Sigma^\infty), \text{length}(A) = \begin{cases} \Sigma\{\text{length}(\varrho) \mid \varrho \in A\} & \text{if } \#(A) < \infty \text{ and } \forall \varrho \in A, \varrho \in \Sigma^+, \\ \infty & \text{otherwise.} \end{cases}$$

Theorem 3. *For all R and all γ, $\mathcal{O}(R) = \mathcal{M}(R)(\gamma)$.*

Proof. Let $R \equiv \langle E \mid S \rangle$. We prove $\forall R\, \forall \gamma\, \forall \sigma\, \mathcal{O}(R)(\sigma) = \mathcal{M}(R)(\gamma)(\sigma)$. As this holds trivially for $\sigma = \perp$, in the sequel assume $\sigma \in \Sigma_0$. We prove Egli-Milner inclusion in both directions.

(1) $\mathcal{O}(R)(\sigma) \sqsubseteq \mathcal{M}(R)(\gamma)(\sigma)$ as follows.

Case A: If R and σ are such that $\mathcal{COMP}(R)(\sigma) \in \mathcal{P}(\Sigma^+)$, then $\mathcal{O}(R)(\sigma) = \mathcal{M}(R)(\gamma)(\sigma)$ proof by cases, applying induction on the sum of the lengths of the computation sequences. (Justified by Lemma 16.)

(i) $S \equiv x := t$.

$$\begin{aligned}\mathcal{O}(\langle E \mid x := t \rangle)(\sigma) &= \kappa(\mathcal{COMP}(\langle E \mid x := t \rangle)(\sigma)) \\ &= \kappa(\{\langle \sigma\{\mathcal{V}(t)(\sigma)/x\} \rangle\}) \\ &= \sigma\{\mathcal{V}(t)(\sigma)/x\} \\ &= \mathcal{M}(\langle E \mid x := t \rangle)(\gamma)(\sigma).\end{aligned}$$

N.B. This result holds for all σ, as $\mathcal{COMP}(\langle E \mid x := t \rangle)(\sigma) \in \mathcal{P}(\Sigma^+)$. By Definition 10 only $\sigma = \perp$ or $S \equiv x := t$ lead to length $(\mathcal{COMP}(R)(\sigma)) = 1$, so the induction basis is provided.

(ii) $S \equiv S_1; S_2$. By Definition 10 and Lemma 16,

$$\text{length}(\mathcal{COMP}(\langle E \mid S_1 \rangle)(\sigma)) < \text{length}(\mathcal{COMP}(\langle E \mid S_1; S_2 \rangle)(\sigma)) < \infty$$

and

$$\text{length}(\mathcal{COMP}(\langle E \mid S_2 \rangle)(\kappa(\mathcal{COMP}(\langle E \mid S_1 \rangle)(\sigma)))) < \text{length}(\mathcal{COMP}(\langle E \mid S_1; S_2 \rangle)(\sigma)) < \infty.$$

So by induction

$$\mathcal{O}(\langle E \mid S_1 \rangle)(\sigma) = \mathcal{M}(\langle E \mid S_1 \rangle)(\gamma)(\sigma)$$

and

$$\mathcal{O}(\langle E \mid S_2 \rangle)(\kappa(\mathcal{COMP}(\langle E \mid S_1 \rangle)(\sigma))) = \mathcal{M}(\langle E \mid S_2 \rangle)(\gamma)(\kappa(\mathcal{COMP}(\langle E \mid S_1 \rangle)(\sigma))).$$

Consequently,

$$\begin{aligned}\mathcal{O}(\langle E \mid S_1; S_2 \rangle)(\sigma) &= \mathcal{O}(\langle E \mid S_2 \rangle) \circ \mathcal{O}(\langle E \mid S_1 \rangle)(\sigma) \quad \text{by Lemma 7} \\ &= \mathcal{O}(\langle E \mid S_2 \rangle)(\kappa(\mathcal{COMP}(\langle E \mid S_1 \rangle)(\sigma))) \\ &\qquad \text{by Definition 21}\end{aligned}$$

$$= \mathscr{M}(\langle E \mid S_2 \rangle)(\gamma)(\kappa(\mathscr{COMP}(\langle E \mid S_1 \rangle)(\sigma)))$$

$$= \mathscr{M}(\langle E \mid S_2 \rangle)(\gamma)(\mathscr{O}(\langle E \mid S_1 \rangle)(\sigma))$$

by Definition 21

$$= \mathscr{M}(\langle E \mid S_2 \rangle)(\gamma) \circ \mathscr{M}(\langle E \mid S_1 \rangle)(\gamma)(\sigma)$$

$$= \mathscr{M}(\langle E \mid S_1; S_2 \rangle)(\gamma)(\sigma) \quad \text{by Definition 31.}$$

Cases (iii), (iv) and (v) of Definition 10 can be treated analogously to (ii), applying Lemma 13 when treating Case 5.

Case B: If R and σ are such that $\perp \in \mathscr{O}(R)(\sigma)$ then $\sigma' \neq \perp$, $\sigma' \in \mathscr{O}(\langle E \mid S \rangle)(\sigma)$ implies $\sigma' \in \mathscr{M}(\langle E \mid S \rangle)(\gamma)(\sigma)$, proof by cases, applying induction on the length of the computation sequence corresponding to that outcome. There may be more than one sequence satisfying this requirement; in that case choose one arbitrary. We again distinguish the following cases.

(i) $S \equiv x := t$. Immediately by the above proved equivalence

$$\mathscr{O}(\langle E \mid x := t \rangle)(\sigma) = \mathscr{M}(\langle E \mid x := t \rangle)(\gamma)(\sigma).$$

By Definition 10 this is the only case pertaining to length $(\varrho) = 1$, $\varrho \in \mathscr{COMP}(\langle E \mid S \rangle)(\sigma)$ so the induction basis is provided.

(ii) $S \equiv S_1; S_2$. Consider a computation sequence

$$\langle \sigma_1 (= \sigma), \sigma_2, \ldots, \sigma_n (= \sigma') \rangle \in \mathscr{COMP}(\langle E \mid S_1; S_2 \rangle)(\sigma).$$

By Definition 10 there is an intermediate state $\sigma_j \neq \perp$ in this sequence such that $\langle \sigma_2, \ldots, \sigma_j \rangle \in \mathscr{COMP}(\langle E \mid S_1 \rangle)(\sigma)$ and $\langle \sigma_j, \ldots, \sigma_n \rangle \in \mathscr{COMP}(\langle E \mid S_2 \rangle)(\sigma_j)$. As length $(\langle \sigma_2, \ldots, \sigma_j \rangle) <$ length$(\langle \sigma_1, \ldots, \sigma_n \rangle)$ and length$(\langle \sigma_j, \ldots, \sigma_n \rangle) <$ length$(\langle \sigma_1, \ldots, \sigma_n \rangle)$, by induction $\sigma_j \in \mathscr{M}(\langle E \mid S_1 \rangle)(\gamma)(\sigma)$ and

$$\sigma' \in \mathscr{M}(\langle E \mid S_2 \rangle)(\gamma)(\sigma_j).$$

Consequently, by Definition 31 $\sigma' \in \mathscr{M}(\langle E \mid S_1; S_2 \rangle)(\gamma)(\sigma)$.

Cases (iii), (iv) and (v) of Definition 10 can be treated analogeously to (ii), applying Lemma 13 when treating Case 5.

Now combining A and B yields $\forall R \, \forall \gamma \, \forall \sigma \, \mathscr{O}(R)(\sigma) \sqsubseteq \mathscr{M}(R)(\gamma)(\sigma)$.

(2) Conversely, we prove $\mathscr{M}(R)(\gamma) \sqsubseteq \mathscr{O}(R)$ as follows:
By Definition 31, it is equivalent to show $\mathscr{N}(S)(\gamma\{\psi_i/P_i\}_{i=1}^n) \sqsubseteq \mathscr{O}(\langle E \mid S \rangle)$. By Definition 31 and Lemma 10, ψ_i can be defined as follows. Let $\langle \psi_1^0, \ldots, \psi_n^0 \rangle = \langle \lambda\sigma \cdot \perp, \ldots, \lambda\sigma \cdot \perp \rangle$

$$\langle \psi_1^{k+1}, \ldots, \psi_n^{k+1} \rangle = \langle \Psi_1(\langle \psi_1^k, \ldots, \psi_n^k \rangle), \ldots, \Psi_n(\langle \psi_1^k, \ldots, \psi_n^k \rangle) \rangle,$$

$$k = 0, 1, \ldots$$

then $\psi_i = \bigsqcup_{k=0}^{\infty} \psi_i^k$, $i = 1, \ldots, n$.

By Lemma 11, $\mathcal{N}(S)(\gamma\{\psi_i^k/P_i\}_{i=1}^n) = \bigsqcup_{k=0}^{\infty} \mathcal{N}(S)(\gamma\{\psi_i^k/P_i\}_{i=1}^n)$. Therefore it is sufficient to show that for all k, $\mathcal{N}(S)(\gamma\{\psi_i^k/P_i\}_{i=1}^n) \sqsubseteq \mathcal{O}(\langle E \mid S\rangle)$. We apply induction on $\langle k, l(S)\rangle$, where $l(S)$ is the length of S, i.e. the number of symbols of S with ordering $\langle k_1, l_1\rangle < \langle k_2, l_2\rangle$ iff $k_1 < k_2$ or $k_1 = k_2$ and $l_1 < l_2$.

$$\mathcal{N}(S)(\gamma\{\psi_i^0/P_i\}_{i=1}^n) = \mathcal{N}(S)(\gamma\{\lambda\sigma \cdot \bot/P\}_{i=1}^n) \sqsubseteq \mathcal{O}(\langle E \mid S\rangle),$$

so the induction basis is satisfied.

We again distinguish the following cases:

(i) $S \equiv x := t$.

$$\mathcal{N}(x := t)(\gamma\{\psi_i^k/P_i\}_{i=1}^n) = \lambda\sigma \cdot \sigma\{\mathcal{V}(t)(\sigma)/x\} = \mathcal{O}(\langle E \mid x := t\rangle).$$

(ii) $S \equiv S_1; S_2$. $l(S_j) < l(S_1; S_2)$, so $\langle k, l(S_j)\rangle < \langle k, l(S_1; S_2)\rangle$, $j = 1, 2$. So by induction $\mathcal{N}(S_j)(\gamma\{\psi_i^k/P_i^k\}_{i=1}^n) \sqsubseteq \mathcal{O}(\langle E \mid S_j\rangle)$, $j = 1, 2$. Consequently, by Lemma 9, 12 and Definition 31,

$$\mathcal{N}(S_1; S_2)(\gamma\{\psi_i^k/P_i\}_{i=1}^n) \sqsubseteq \mathcal{O}(\langle E \mid S_1; S_2\rangle).$$

Cases (iii) and (iv) of Definition 10 can be treated analogeously to (ii).

(v) $S \equiv P$.

By Definition 10, $P \equiv P_j$, $P_j \Leftarrow S_j$ in E. By Lemma 10, $\mathcal{O}(\langle E \mid P\rangle) = \mathcal{O}(\langle E \mid S_j\rangle)$. If $k = 0$ there is nothing to prove. Otherwise

$$\begin{aligned}
\mathcal{N}(S_j)(\gamma\{\psi_i^k/P_i^k\}_{i=1}^n = \psi_j^k &= \Psi_j(\psi_1^{k-1}, \ldots, \psi_n^{k-1}) \\
&= \mathcal{N}(S_j)(\gamma\{\psi_i^{k-1}/P_i\}_{i=1}^n) \quad \text{by Definition 31} \\
&\sqsubseteq \mathcal{O}(\langle E \mid S_j\rangle) \\
&= \mathcal{O}(\langle E \mid P\rangle).
\end{aligned}$$

Combining these results yields $\mathcal{M}(R)(\gamma) \sqsubseteq \mathcal{O}(R)$, i.e.

$$\forall k\, \forall \gamma\, \forall \sigma\, \mathcal{M}(R)(\gamma)(\sigma) \sqsubseteq \mathcal{O}(R)(\gamma)(\sigma).$$

Acknowledgements

I wish to thank Jaco de Bakker for his stimulating remarks and Arie de Bruin for his fruitful cooperation during the process of writing this paper. To Linda Brown and Susan Carolan I am grateful for the efficient typing of the manuscript. The referees I thank for their constructive remarks.

References

[1] K.R. Apt and G. Plotkin, A Cook's tour of countable nondeterminism, in: Proc. 8th Int. Colloq. on Automata, Languages and Programming (Springer, Berlin, 1981) to appear.
[2] A. Arnold and M. Nivat, Metric interpretations of infinite trees and semantics of nondeterministic recursive programs, Theoret. Comput. Sci. 11 (1980) 108–205.
[3] R.-J. Back, Semantics of unbounded nondeterminism, in: J.W. de Bakker and J. van Leeuwen, (Eds.), Proc. 7th Int. Colloq. on Automata, Languages and Programming, Lecture Notes in Computer Science, Vol. 85 (Springer, Berlin, 1980) 51–63.
[4] J.W. de Bakker, Semantics of infinite processes using generalized trees, in: J. Grusk (Ed.), Mathematical Foundations of Computer Science 1977, Lecture Notes in Computer Science, Vol. 53 (Springer, Berlin, 1977) 240–252.
[5] J.W. de Bakker, Mathematical Theory of Program Correctness (Prentice Hall, Englewood Cliffs, NJ, 1980).
[6] A. de Bruin, On the existence of Cook semantics, Mathematical Centre Report, IW 163/81 (1981).
[7] S.A. Cook, Soundness and completeness of an axiom system for program verification, SIAM J. Comput. 7 (1978) 70–90.
[8] J. Dugundji, Topology (Allyn and Bacon, Boston, IL, 1966).
[9] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivations of programs, Comm. ACM 18 (1975) 453–457.
[10] H. Egli, A mathematical model for nondeterministic computations, ETH, Zürich (1975).
[11] E.A. Emerson and E.M. Clarke, Characterizing correctness properties of parallel programs using fixed points, in: J.W. de Bakker and J. van Leeuwen (Eds.), Proc. 7th Int. Colloq. on Automata, Languages and Programming, Lecture Notes in Computer Science Vol. 85 (Springer, Berlin, 1980) 169–181.
[12] C.A.R. Hoare, Communicating sequential processes, Comm. ACM 21 (1978) 666–677.
[13] M. Nivat, Infinite words, infinite trees, infinite computations, in: J.W. de Bakker and J. van Leeuwen (Eds.), Foundations of Computer Science III, Part 2, Mathematical Centre Tract 109 (1979) 1–52.
[14] D. Park, On the semantics of fair parallelism, in: D. Bjørner (Ed.), Proc. 1979 Copenhagen Winter School, Lecture Notes in Computer Science Vol. 86 (Springer, Berlin, 1979) 504–526.
[15] J. Stoy, Denotational Semantics, The Scott-Strachey Approach to Programming Language Theory (MIT Press, Cambridge, MA, 1977).

Algorithmic Languages, de Bakker/van Vliet (eds.)

A Proof Rule for Fair Termination of Guarded Commands*

Orna Grümberg, Nissim Francez, Johann A. Makowsky
and Willem P. de Roever†

Department of Computer Science, Technion-Israel Institute of Technology, Haifa, Israel
† *Vakgroep Informatika, Rijksuniversiteit Utrecht, Utrecht, The Netherlands*

We present a proof rule for fairly terminating guarded commands based on a well-foundedness argument. The rule is applied to several examples, and proved to be sound and complete w.r.t. an operational semantics of computation trees. The rule is related to another rule suggested by Pnueli, Stavi and Lehmann by showing that the (semantic) completeness of the PSL-rule follows from the completeness of ours'.

1. Introduction

The use of well-ordered sets to prove termination of programs originates from Floyd [3] and remained prominent ever since. After the appearance of non-deterministic and concurrent programming language constructs, the notion of termination was generalized to the notion of *liveness* [10], which also covers properties such as *eventual* occurrence of events during program execution. One way of specifying and proving such properties is by applying *temporal reasoning* [4]. This may be formalized by using Temporal Logic [12], a tool suitable for expressing such eventualities.

* Preliminary work regarding this problem was carried out while the 2nd author visited the University of Utrecht, sponsored by a grant from The Netherlands Organization for the Advancement of Pure Research (Z.W.O.); the work was completed while the 4th author visited the Technion sponsored by the Technion; the 2nd author was partly sponsored by an IBM-Israel Research grant. The third author was supported by Swiss National Science Foundation grant No. 82.820.0.80.

Within this framework, one of the more interesting concepts that can be studied is the concept of *fairness* [6]. However, application of temporal reasoning does not appeal to a direct use of well-foundedness arguments (see e.g. [11]). Recently, there is a revival of the interest in such direct appeals (see e.g. [1]), generalizing arguments hitherto involving finite non-determinism to a context of infinite non-determinism, and [13], generalizing sequential well-foundedness arguments to the context of concurrency (using a shared variable model).

A common property of well-foundedness arguments for more complicated types of termination is the use of higher countably infinite ordinals, which can be traced back to [8], this in contrast to the fact that for deterministic programs (or programs displaying finite non-determinism) natural numbers suffice.

In this paper, we propose a rule for proving *fair termination* of guarded loops using well-foundedness arguments.

We chose guarded commands [2] since it is relatively well known and simple, has as a natural extension to the language Communicating Sequential Processes (CSP) [9] and the proof rule proposed in this paper extends equally naturally to CSP. This extension is the subject of a companion paper.

The ideas in this paper were developed mostly independent of [13], in which a similar situation is dealt with. We shall describe the influence of [13] on our work in the last section.

In Section 2, we introduce the proof rule for termination and apply it to several examples. In Section 3 we present soundness and semantic completeness proofs of the suggested rule w.r.t. an operational semantics using computation trees. Section 4 ends with a reduction of the semantic completeness of the rule of [13] to the present one.*

2. A Proof Rule for Fair Termination

Basic notions and definitions

We consider the language GC, with the following syntax:

* *Note added in proof*: Conversely, Daniel Lehmann recently reduced the completeness proof of our rule to that of [13]. Consequently, the two rules are equivalent.

⟨statement⟩ ::= ⟨assignment statement⟩ | ⟨skip⟩ | ⟨selection⟩
| ⟨repetition⟩ | ⟨composition⟩
⟨assignment statement⟩ ::= ⟨variable⟩ := ⟨expression⟩
⟨skip⟩ ::= *skip*
⟨selection⟩ ::= [⟨boolean-expression⟩ → ⟨statement⟩
{□
⟨boolean-expression⟩ → ⟨statement⟩}*
]
⟨repetition⟩ ::= *⟨selection⟩
⟨composition⟩ ::= ⟨statement⟩; ⟨statement⟩.

Boolean expressions are also called *guards.*

Its semantics follows from the usual definition of computation sequence π: $\xi_0\xi_1 \ldots$, where all ξ_i's denote states (mappings from variables to values).

In the sequel we consider programs of the form of repetitions

$$C :: *[B_1 \to C_1 \,\square\, \cdots \,\square\, B_n \to C_n],$$

also abbreviated to $*[\square_{i \in \{1,\ldots,n\}} B_i \to C_i]$.

C_i is *enabled* in ξ iff $B_i(\xi)$ holds.

Definition. (1) An execution sequence π of C is *fair* iff it is finite, or it is infinite and for every $1 \leq i \leq n$, if C_i is infinitely often enabled along π, it is also infinitely often chosen along π.

(2) C is *fairly terminating* iff all its infinite execution sequences are not fair, i.e., *unfair.*

Thus, a fairly terminating program has finite computation sequences (terminating computations), and unfair infinite computation sequences, but may *not* have infinite fair computation sequences.

For a given initial state ξ, we consider the tree of all possible computation sequences, T_ξ. In case of a selection, $[B_1 \to C_1 \,\square\, \cdots \,\square\, B_n \to C_n]$, a state (node) η in T_ξ has subtrees for every i, $1 \leq i \leq n$ s.t. $B_i(\eta)$ holds. Observe that in case of fair termination, T_ξ contains finite and unfair infinite computation paths.

Example. Consider Dijkstra's example for a random generator of natural numbers [2]; this is a possibly non-terminating program, its only infinite computation sequence being unfair. Hence this program fairly terminates:

$$C :: x := 0;\ b := \mathit{true};$$
$$\quad *[b \to x := x + 1$$
$$\quad \Box b \to b := \mathit{false}].$$

Notice that by restricting the underlying semantics of the language of guarded commands to fair computation sequences only, a fairly terminating repetition as defined above becomes a terminating one, possibly displaying countably infinite nondeterminism.

Our goal is to characterize deductively the class of all fairly terminating GC programs. The characterization suggested does carry over directly to concurrent programs with shared variables; a companion paper extends it to CSP.

We use the notation $\langle\langle r \rangle\rangle C \langle\langle q \rangle\rangle$ to express that C fairly terminates in all initial states satisfying r, and that q holds upon termination.

The intuition behind the suggested proof rule is as follows: For an always terminating nondeterministic program, there exists a well-founded quantity which decreases along every computation sequence, i.e., along every direction in the computation tree.

Now, let us *choose* the directions along which a certain well-founded quantity decreases, taking care that these directions (certain moves C_i) are always eventually enabled, until they are taken. Let the other directions be non-increasing. Then by the fairness assumption eventually a decreasing move has to occur. Thus all fair computation sequences are guaranteed to be finite.

The proof rule

Choose a well-ordered set $(W, \leq)$ (without loss of generality we can assume that W is an initial sequence of the countable ordinals, as shown by the completeness proof). Also choose a predicate

$$p : W \to [\text{States} \to \{\mathit{true}, \mathit{false}\}],$$

assigning a truth value to every pair (w, ξ).

For each $w \in W$, $w > 0$ (or, in general, any non-minimal element in W) choose a partition D_w, S_w of $\{1, \ldots, n\}$, with $D_w \neq \emptyset$. (D stands for decreasing, S for steady.)

Let the following clauses hold:

(1) $\quad \langle\langle p(w) \wedge w > 0 \wedge B_j \rangle\rangle C_j \langle\langle \exists v < w \cdot p(v) \rangle\rangle \quad$ for all $j \in D_w$,

(2) $\langle\langle p(w) \wedge w > 0 \wedge B_i \rangle\rangle C_i \langle\langle \exists v \leq w \cdot p(v) \rangle\rangle$ for all $i \in S_w$,

(3) $\langle\langle p(w) \wedge w > 0 \rangle\rangle * \left[\Box_{i \in S_w} B_i \wedge \neg \bigvee_{j \in D_w} B_j \rightarrow C_i \right] \langle\langle true \rangle\rangle$

(4) $P(0) \supset q \wedge \bigwedge_{i=1}^{n} \neg B_i, \qquad w > 0 \wedge p(w) \supset \bigvee_{i=1}^{n} B_i, \qquad r \supset \exists v \cdot p(v).$

Then, we conclude

$$\langle\langle r \rangle\rangle C \langle\langle q \rangle\rangle,$$

i.e., repetition C fairly terminates.

Explanation

(ad 1) This clause guarantees that along every direction in D_w, if it is enabled and taken, then there is a decrease in the well-ordering. (Note again that we use a unique minimal element, denoted by 0, to keep the notation simple.) Note also that at least one decreasing direction is required.

(ad 2) This clause guarantees that along every direction in S_w, if enabled and taken, there is no increase in the well-ordering. Thus, an infinite computation proceeding along S_w direction only, and not decreasing, is possible. We have to assure that such a sequence is unfair. Whence clause (3).

(ad 3) This clause imposes a recursive application of the rule to an auxiliary program C_w, and hence requires a subproof. C_w terminates because of one of two reasons:

(a) $\wedge_{i \in S_w} \neg B_i$ is true, hence no S_w-move is possible and only D_w-moves are left.

(b) For some $j \in D_w$, B_j is true, i.e., a D_w-move is enabled. Hence, this clause guarantees that along infinite S_w-computations, D_w-moves are infinitely often enabled, that is, such computations are unfair. By convention, $C_w = skip$ if $S_w = \emptyset$.

(ad 4) This clause guarantees that the program terminates only when reaching a minimal element of $(W, <)$.

Remarks. (1) If we take $S_w = \emptyset$ (and hence $D_w = \{1, \ldots, n\}$) for all $w \in W$, the rule reduces to the usual termination rule for *GC* (see e.g. [7]).

(2) In proving clauses (1)–(4) of the rule, application of the ordinary rules (for assignments, etc.) is allowed.

Example 1. First, consider again Dijkstra's example for a random generator of natural numbers [2], which is a possibly non-terminating program, its only infinite computation sequence being unfair. Hence, this program terminates fairly.

$$\begin{aligned} C :: & \; x := 0;\ b := \mathit{true};\\ & *[b \to x := x + 1\\ & \Box b \to b := \mathit{false}\\ &]. \end{aligned}$$

We prove $\langle\langle \mathit{true}\rangle\rangle C \langle\langle \mathit{true}\rangle\rangle$. Choose as well-ordering $\{0, 1\}$ with $0 < 1$, as $S_1 = \{1\}$, $D_1 = \{2\}$, and as ranking predicate

$$p(w)(x, b) \overset{\text{def}}{\equiv} (w = 1 \supset b) \wedge (w = 0 \supset \neg b).$$

As to clause (1): b changes from *true* to *false* upon move $b := \mathit{false}$, and hence w drops from 1 to 0.

As to clause (2): b remains *true* under $x := x + 1$, and $p(w)$ is independent of x, so w stays 1.

As to clause (3): $C_1 :: *[b \wedge \neg b \to \cdots]$ which obviously terminates.

Example 2. In Example 1, a D-move is always enabled (in the terminology of [13], that program is *just*). Next, consider a program, in which *D-moves are only eventually enabled,* and clause (3) is less trivially satisfied.

$$\begin{aligned} C :: & \; b := \mathit{true};\ c := \mathit{true};\\ & *[b \to c := \neg c\\ & \Box b \wedge c \to b := \mathit{false}\\ &]. \end{aligned}$$

Again we prove $\langle\langle \mathit{true}\rangle\rangle C \langle\langle \mathit{true}\rangle\rangle$. Choose W, p, S_1, D_1 as above. The difference lies in clause (3), with auxiliary program

$$C_1' :: *[b \wedge \neg (b \wedge c) \to c := \neg c],$$

which terminates after one step at most.

This example is still trivial, but is should give the reader a feeling for the spirit of the rule, which captures eventual enabling of a D-move by means of a proof of termination of the auxiliary program.

Example 3. Next, we show that the natural numbers N are not sufficient for fair termination proofs, since there is no bound on the length of finite computations.

Let x, y, z range over N.

$$
\begin{aligned}
&C :: x := 0;\ y := 0;\\
&\quad *[x = 0 \to y := y + 1\ \Box x = 0 \to x := 1\\
&\quad\ \Box x \neq 0 \wedge y \neq 0 \to y := y - 1\ \Box x \neq 0 \wedge y \neq 0 \to z := z + 1\\
&\quad\ ].
\end{aligned}
$$

To prove $\langle\langle true\rangle\rangle\, C\, \langle\langle true\rangle\rangle$, choose $W = N \cup \{\infty\}$,

$$p(w)(x,y,z) \overset{\text{def}}{\equiv} (w = \infty \supset x = 0) \wedge (w \neq \infty \to x \neq 0 \wedge y = w),$$

$$S_\infty = \{1,3,4\}, \qquad D_\infty = \{2\}, \qquad S_n = \{1,2,4\}, \qquad D_n = \{3\}.$$

For clause (3) we get as auxiliary programs:

$$
\begin{aligned}
&C_\infty :: *[x = 0 \wedge x \neq 0 \to \cdots\\
&\qquad \Box x \neq 0 \wedge y \neq 0 \to y := y - 1\\
&\qquad \Box x \neq 0 \wedge y \neq 0 \to z := z + 1\\
&\qquad].
\end{aligned}
$$

$$
\begin{aligned}
&C_n :: *[x = 0 \to \cdots\\
&\qquad\qquad \Box x = 0 \to \cdots\ \Box x \neq 0 \wedge y \neq 0 \wedge \neg (x \neq 0 \wedge y \neq 0) \to \cdots].
\end{aligned}
$$

To prove $\langle\langle p(n) \wedge n > 0\rangle\rangle\ C_n\ \langle\langle true\rangle\rangle$ is trivial since $p(n) \supset x \neq 0$, and hence C_n terminates immediately.

To prove $\langle\langle x = 0\rangle\rangle\ C_\infty\ \langle\langle true\rangle\rangle$, choose $W' = N$, and let $S'_n = \{1,3\}$, $D'_n = \{2\}$, $n \in N$, and $p(n)(x,y,z) \overset{\text{def}}{\equiv} y = n \wedge x \neq 0$. Note that the alternatives are renumbered.

Clause (1) is satisfied since $y := y - 1$ decreases y, and clause (2) is satisfied since $p(n)$ is independent of z. As to clause (3), we again construct an auxiliary program, $C_{\infty,n}$,

$$C_{\infty,n} :: [x = 0 \wedge x \neq 0 \to \cdots\ \Box x \neq 0 \wedge y \neq 0 \wedge \neg (x \neq 0 \wedge y \neq 0) \to \cdots],$$

which trivially terminates.

Finally, consider the following program:

$$
\begin{aligned}
&C :: y := 1;\ b := \mathit{true};\\
&\quad *[b \to y := y + 1\\
&\quad\ \Box b \wedge \mathrm{prime}(y) \wedge \mathrm{prime}(y + 2) \to b := \mathit{false}\\
&\quad\ ].
\end{aligned}
$$

This program fairly terminates iff the conjecture that there exist infinitely many 'twin' primes is *true*.

3. Soundness and Semantic Completeness

In this section we prove the soundness of the suggested proof rule w.r.t. the semantics of computation trees consisting of fairly terminating sequences, and its semantic completeness. We shall not deal in this paper with the specification language needed to express $p(w)$ and the partitions, an issue dealt with elsewhere [15].

(a) *Soundness.* We have to prove that if all premises of the rule hold, so does its conclusion.

Assume that for program C we found a well-ordered set $(W, \leq)$, a partition S_w, D_w for each $w>0$ s.t. $D_w \neq \emptyset$, and a predicate p, satisfying clauses (1)–(4) of the rule.

Assume by way of contradiction, that for some state ξ_0, T_{ξ_0} contains an infinite fair path $\langle \xi_i \rangle_{i=0}^{\infty}$. Consider the corresponding sequence of moves $\langle d_i \rangle_{i=0}^{\infty}$. It cannot contain an infinite subsequence $\langle d_{i_j} \rangle_{j=0}^{\infty}$ of D-moves, since by clause (1) this would imply the existence of an infinite decreasing sequence of elements in W, contradicting W's well-foundedness. Thus, from some k onwards, $p(w)(\xi_k)$ holds, and all moves d_j for $j>k$ are S_w-moves (by clause (2)). By clause (3) there is some $d \in D_w$ which is infinitely often enabled and not taken, contradicting the assumption that $\langle \xi_i \rangle_{i=0}^{\infty}$ is fair.

(b) *Completeness.* This is the harder part. Assume $\langle\langle r \rangle\rangle C \langle\langle q \rangle\rangle$ holds. Then we have to find a well-ordered set $\langle W, \leq \rangle$, partitions S_w, D_w for each $w>0$ s.t. $D_w \neq \emptyset$, and a predicate p (given by a collection of pairs (w, ξ)) such that clauses (1)–(4) hold.

Since all we 'have at hand' is the computation tree, we have to derive everything needed from that tree (compare also [14] for another well-foundedness argument based on the 'operational' object ~ the computation stack, for nondeterministic recursive procedures).

We are given that the computation tree T_{ξ_0}, for every state ξ_0 satisfying r, is either well-founded, or contains at least one infinite, hence unfair, computation sequence. The basic idea is to construct another (possibly

infinitely wide) tree $T^*_{\xi_0}$, some of whose nodes are obtained by collapsing certain infinite families of nodes in T_{ξ_0}, all lying on unfair sequences originating in nodes $\xi \in T_{\xi_0}$, *such that* $T^*_{\xi_0}$ *is well founded,* i.e., contains finite paths only. Then we use a standard ranking of $T^*_{\xi_0}$ by means of ordinals. A move which leaves ξ and remains in the same infinite family belongs to S_w for the corresponding rank. A move which exits such a family belongs to D_w. Special care must be taken that these partitions do not depend on ξ_0, the root of the computation tree.

We now present the details of the construction. Let T_{ξ_0} be given.

Case (a): T_{ξ_0} is well founded (this means that C always terminates in ξ_0). Choose a ranking of the nodes by means of an initial segment of the ordinals, ranking leaves by 0, and proceeding inductively level by level from leaves till root (a standard set-theoretical construction); furthermore, choose uniformly $S_w = \emptyset$, $D_w = \{1, \ldots, n\}$. It is easy to verify that clauses (1)–(4) of the rule hold.

Case (b): T_{ξ_0} contains at least one unfair, hence infinite, computation path π. This case is dealt with below.

Definition. (1) A computation sequence π is d-unfair ($1 \leq d \leq n$) iff along π C_d was infinitely often enabled, but only finitely often chosen.

(2) Let $\xi \in T_{\xi_0}$. Define ξ's d-cone $\mathrm{CONE}_d(\xi)$ as follows:

$\mathrm{CONE}_d(\xi) =$ the set of all occurrences of states in T_{ξ_0} residing on infinite computation sequences which contain only finitely many d-moves and which start in ξ.

(Obviously, all occurrences of states on d-unfair sequences starting in ξ belong to its d-cone.)

Lemma 1. *Let* $\xi \in T_{\xi_0}$, *and let* $\eta \in \mathrm{CONE}_d(\xi)$, *for some* $1 \leq d \leq n$. *Then every computation sequence leaving* $\mathrm{CONE}_d(\xi)$, *say at node* η, *is either finite or contains a d-move.*

Proof. Suppose not. Then an infinite path π starts in η and does not contain any d-move. Since $\eta \in \mathrm{CONE}_d(\xi)$, there is some finite path π' joining ξ to η, along which a d-move was taken at most a finite number of times. Hence the concatenation $\pi'\pi$ of π' and π is contained in $\mathrm{CONE}_d(\xi)$, contradicting the assumption that π leaves $\mathrm{CONE}_d(\xi)$.

The situation is described in Fig. 1, where a triangle denotes a well-founded tree.

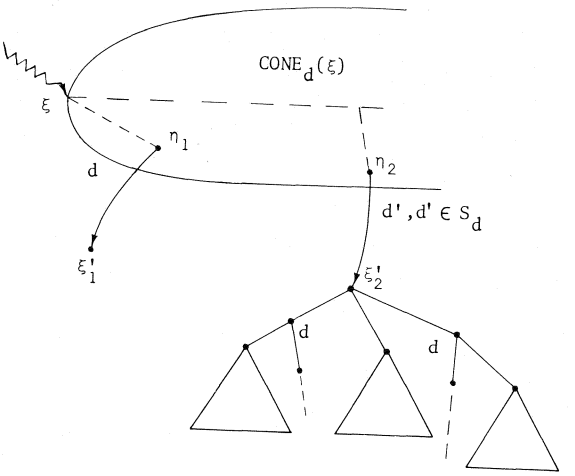


Fig. 1.

Observation. If state ξ resides on a d-unfair sequence, then $\mathrm{CONE}_d(\xi) \neq \emptyset$.

Our candidates for families 'to be collapsed into a node in $T^*_{\xi_0}$' are such d-cones.

Next we define inductively a hierarchy of d-cones.

Base step: Since by assumption T_{ξ_0} contains an unfair sequence, fix some $1 \leq d_0 \leq n$ s.t. there exists a d_0-unfair sequence in ξ_0, and let $\mathrm{CONE}_{d_0}(\xi_0)$ be defined as above. It is not empty by the observation above. We say that $\mathrm{CONE}_{d_0}(\xi_0)$ is at level 0.

Induction step: Suppose at level $i-1$ a d-cone $\mathrm{CONE}_d(\xi_{i-1})$ was defined, and let π be some path leaving $\mathrm{CONE}_d(\xi_{i-1})$. By Lemma 1 either π is finite, or there is a d-move on path π resulting in state ξ_i. If π is finite we finish the construction as far as π is concerned. So assuming state ξ_i as above, construct $\mathrm{CONE}_{d'}(\xi_i)$ at level i, where d' is determined as follows:

If there is a move d' not appearing in $\xi_0 \cdots \xi_1 \cdots \xi_{i-1} \cdots \xi_i$, and there is an infinite sequence with a finite number of occurrences of d' starting in ξ_i, choose move d'. Otherwise, choose the index of the move which did not appear longest in $\xi_0 \cdots \xi_i$, for which there is an infinite sequence containing finitely many occurrence of that move, starting in ξ_i.

Thus, when iterating the cone construction, we *vary* the move-indices of the cones *maximally*.

Lemma 2. *There does not exist an infinite sequence of cones* $\mathrm{CONE}_{d_i}(\xi_i)$ *s.t.* $\langle \xi_i \rangle_{i=0}^{\infty}$ *is an infinite path of* T_{ξ_0}.

Remark. If we describe the construction of cones as in Fig. 2, we have by Lemma 2 only finite chains of cones.

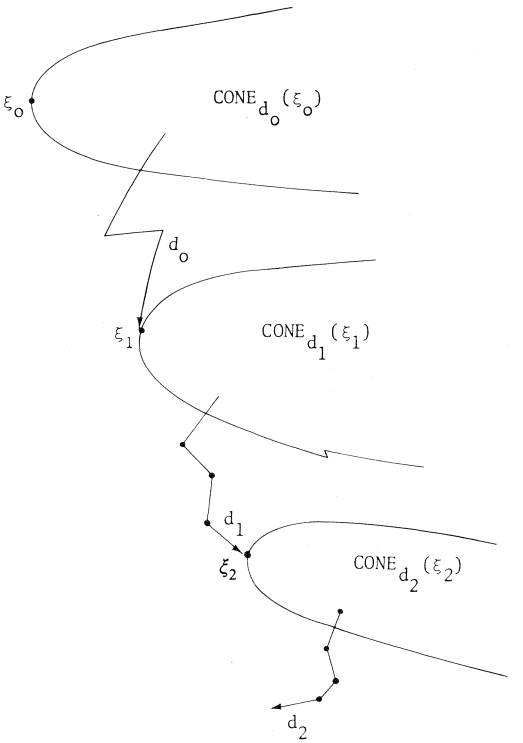
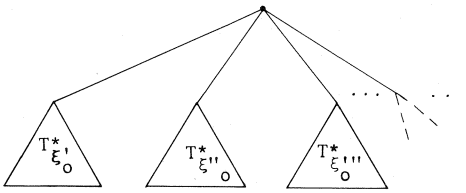


Fig. 2.

Proof. Suppose such an infinite sequence $\langle \xi_i \rangle$ exists. Then it is unfair by definition of T_{ξ_0}. Thus, there is some $1 \leq \bar{d} \leq n$ s.t. $\langle \xi_i \rangle$ is $\bar{d}$-unfair. Then there is an i_0 s.t. at ξ_{i_0} either $\bar{d}$ did not occur on $\xi_0 \cdots \xi_{i_0}$ or it occurred less recently than any other move. Hence $\bar{d} = d_{i_0}$ in the inductive construction of $\mathrm{CONE}_{d_{i_0}}(\xi_{i_0})$, and $\langle \xi_i \rangle$ would have been contained in $\mathrm{CONE}_{d_{i_0}}(\xi_{i_0})$, contrary to assumption.

Now we define $T^*_{\xi_0}$ as suggested above. Its nodes are all the nodes in T_{ξ_0} not belonging to any cone, and the set of all cones. Its edges are either edges entering cones, or edges leaving cones, and, otherwise, edges outside cones. By Lemmas 1 and 2, the tree $T^*_{\xi_0}$ is well founded.

In order to get rid of unwanted ξ_0-dependence of S_w and D_w as suggested above, we do one more construction: Combine all $T^*_{\xi_0}$ s.t. $r(\xi_0)$ holds into one infinitary well-founded tree T^*_C:



Next, rank the nodes of T^*_C. However, we must take care that if ξ occurs in two places in T^*_C with the same rank, it determines some (S, D) partition uniquely.*

In order to achieve this we perform a *rank-shift:* Suppose that at some level of the ranking, say λ, there are equiranked occurrences of a state ξ, say of ordertype α. Then rerank these consecutively by $\lambda+1, \ldots, \lambda+\alpha$, and proceed to the next level $\lambda+\alpha+1$.

Let ϱ denote the ranking function of T^*_C. Then we define predicate p and partitions (S_w, D_w). As W we chose the ordinals ranking T^*_C, an initial segment of the countable ordinals.

$$p(w)\xi \underset{\text{DEF}}{=} \exists\, \eta, d \cdot \xi \in \text{CONE}_d(\eta) \wedge \varrho(\text{CONE}_d(\eta)) = w$$
$$\vee$$
$$\forall\, \eta, d \cdot \xi \notin \text{CONE}_d(\eta) \wedge \varrho(\xi) = w.$$

For $w>0$:

$$S_w = \begin{cases} S_d & \text{if } \exists\, \eta, d \cdot \varrho(\text{CONE}_d(\xi)) = w, \\ \emptyset & \text{otherwise} \end{cases}$$

where $S_d = \{1, \ldots, n\} - \{d\}$. Hence, $D_w = \{d\}$, a singleton set, or $D_w = \{1, \ldots, n\}$.

* *Note added in proof*: Due to technical considerations, all non-leave nodes should be ranked differently.

Note that the rank-shift of T_C^* assures that S_w is well defined.

Next, we show that clauses (2)–(4) of the rule hold; and thereafter we refine the cone-construction so as to satisfy clause (1), too.

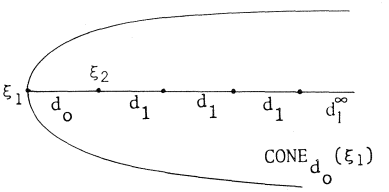
Lemma 3.1. *W, p, (S_w, D_w) satisfy clause* (2)–(4) *of the rule. (As we shall see clause* (1) *need not hold.)*

Proof. *Clause* (2): Assume $p(w) \wedge w > 0 \wedge B_i$ holds in ξ, for $i \in S_w$. Without loss of generality (by the rank-shift), assume $\xi \in T_{\xi_0}$ and $r(\xi_0)$ holds. Then $\xi \in \mathrm{CONE}_d(\eta)$ for some η and d (since, otherwise, $S_w = \emptyset$), and $d \neq i$. If move C_i remains in the cone, by construction the rank remains the same. Otherwise, it leaves the cone, and hence, since T_C^* is ranked from bottom-leaves to top-root, the rank decreases.

Clause (3): Assume again $p(w) \wedge w > 0$ holds in ξ. We have to demonstrate that C_w fairly terminates. Since $S_w \neq \emptyset \supset S_w = S_d$ for some d, the guards of C_w are $B_i \wedge B_d$. Again, assume we are in T_{ξ_0} as above. Let π be a fair computation sequence of C_w starting in ξ. Then π can be extended in front to a fair computation sequence starting in ξ_0, and hence is finite. Thus C_w fairly terminates. (At this point it should be clear to the reader that the whole proof proceeds by induction on the number of alternatives of C.)

Clause (4): By construction, in T_C^* holds $\varrho(\xi) = 0 \leftrightarrow \xi$ is a leaf of T_C^*.

To see that condition (1) does not hold, consider the case:



I.e., $d_0 d_1^\infty$ labels a d_0-unfair computation sequence, contained in $\mathrm{CONE}_{d_0}(\xi_1)$, and let $\varrho(\mathrm{CONE}_{d_0}(\xi_1)) = w$. Then $p(w)\xi_1 \wedge w > 0 \wedge B_0$ holds, and hence, $\langle\langle p(w)\xi_1 \wedge w > 0 \wedge B_0 \rangle\rangle C_0 \langle\langle p(w)\xi_2 \rangle\rangle$, that is, w need not necessarily decrease under the C_0 move as indicated.

Finally, we modify our construction of cones so as to satisfy clause (1) of the rule, too. This modification affects the collapsing of a d-cone; instead of collapsing such a cone to a node of $T_{\xi_0}^*$, we collapse it to a well-founded subtree of $T_{\xi_0}^*$.

Let $\mathrm{CONE}_d(\xi)$ be given. Now repeat the inductive construction, but modified by defining subcones *within* $\mathrm{CONE}_d(\xi)$ which include only infinite computation sequences containing *no* occurrences of *d at all,* and ξ itself (hence never being empty).

Definition. For $\eta \in \mathrm{CONE}_d(\xi)$, let $\text{S-CONE}_d(\eta) = (\text{the set of all occurrences of states along infinite paths in } \mathrm{CONE}_d(\xi) \text{ starting in } \eta \text{ and containing } no \text{ occurrence of a } d\text{-move}) \cup \{\eta\}$.

By an argument similar to the one in the proof of Lemma 1 we establish:

Lemma 4. *Every computation sequence leaving* $\text{S-CONE}_d(\eta)$ *is either finite or contains a d-move.*

The inductive construction of subcones of $\mathrm{CONE}_f(\xi)$ goes as follows: At level 0, define $\text{S-CONE}_d(\eta_0)$ with $\eta_0 = \xi$. Suppose $\text{S-CONE}_d(\eta_{i-1})$ is defined (at level $i-1$).

Case (1): There exists a computation sequence leaving $\text{S-CONE}_d(\eta_{i-1})$ which does not leave $\mathrm{CONE}_d(\xi)$, thereby being infinite. Let C_d denote the first occurrence of a d-move along that computation sequence. Such a d-move exists by Lemma 4, since we exclude finite sequences (as these left the 'big' $\mathrm{CONE}_d(\xi)$ already). In case $\text{S-CONE}_d(\eta_{i-1}) = \{\eta_{i-1}\}$, a computation sequence 'leaving' $\text{S-CONE}_d(\eta_{i-1})$ starts in η_{i-1}. Let η_i denote the resulting occurrence of a state. Then define the descendant $\text{S-CONE}_d(\eta_i)$ at level i.

Case (2): There does not exist a computation sequence leaving $\text{S-CONE}_d(\eta_{i-1})$. Then this S-CONE has no descendant.

Lemma 5. *There does not exist an infinite chain of* $\text{S-CONE}_d(\eta_i)$*'s with* $\eta_0 = \xi$.

Proof. Suppose such a chain exists. Then there exists an infinite computation sequence starting in ξ with an infinite number of occurrences of d-moves, contained in $\mathrm{CONE}_d(\xi)$, contradicting the definition of $\mathrm{CONE}_d(\xi)$.

Thus, we now collapse each $\mathrm{CONE}_d(\xi)$ into a well-founded subtree, with subcones $\text{S-CONE}_d(\eta)$ collapsed to nodes. By Lemma 5 this subtree is well founded, and hence, the whole tree $T^*_{\xi_0}$ is well founded. Now repeat the previous ranking procedure to $T^*_{\xi_0}$ so obtained.

Now, clause (1) holds, too, because every d-move either leads to a lower ranked node corresponding to a subcone, or leaves the whole cone, therefore also leading to a lower ranked node. Satisfaction of the other clauses is not affected by the modification described above. Hence we established:

Theorem. *If C fairly terminates, $\langle W, \leq \rangle$, p, $\langle (S_w, D_w) \rangle_{w \in W, w > 0}$, exist satisfying all the clauses appearing as premises in our rule for proving fair termination of guarded loops.*

Comparing the construction in the completeness proof with the statement of the rule itself, one cannot help noticing that there is a certain discrepancy between the two. In the construction, we always end up with $|D_w = 1|$ for collapsed nodes, whereas the rule itself allows $|D_w| > 1$. We would like to give some semantic significance to the case $|D_w| > 1$ in the light of the previous construction.

Suppose in π there exist infinite computation sequences $\pi_1, \ldots, \pi_k$, not containing, respectively, moves $d_1, \ldots, d_k$ an infinite number of times. Then $\pi_i \in \bigcup_{i=1}^{k} \mathrm{CONE}_{d_i}(\xi)$.

Define $\mathrm{CONE}_{\{d_1, \ldots, d_k\}}(\xi) = \bigcup_{i=1,\ldots,k} \mathrm{CONE}_{d_i}(\xi)$, where $\{d_1, \ldots, d_k\}$ is the *maximal* set of moves s.t. $\mathrm{CONE}_{d_i}(\tau) \neq \emptyset$, $i = 1, \ldots, k$. Next, one verifies:

Lemma 6. *Every infinite sequence leaving* $\mathrm{CONE}_{\{d_1, \ldots, d_k\}}(\xi)$ *contains moves* $d_1, \ldots, d_k$.

Then, one modifies the iterative cone construction in that a new (generalized) cone is constructed after all moves $d_1, \ldots, d_k$ occurred. Observe that the analogue of Lemma 2 holds again.

Now, generalize the construction of subcones to maximal sets of moves. Assume $k = 2$, for simplicity of notation (the construction generalizes to $k \leq n$). In order to satisfy clause (1), we refine our ranking, as in Fig. 3.

Split $\text{S-CONE}_{\{d_1, d_2\}}(\xi)$ into three parts:

$$\text{S-CONE}_{d_1}(\xi) - \text{S-CONE}_{d_2}(\xi),$$

$$\text{S-CONE}_{d_2}(\xi) - \text{S-CONE}_{d_1}(\xi),$$

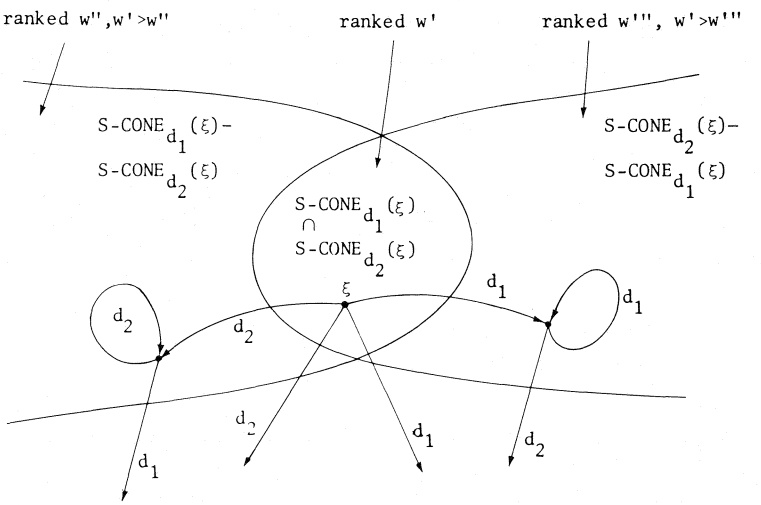


Fig. 3.

and

$$\text{S-CONE}_{d_1}(\xi) \cap \text{S-CONE}_{d_2}(\xi),$$

and rank them, respectively, w'', w''', w' with $w' > w''$, $w' > w'''$. (This can be easily accomplished by superposing a lexicographical order on ϱ.)

Choose $D_{w'} = \{d_1, d_2\}$, $D_{w''} = \{d_1\}$, $D_{w'''} = \{d_2\}$. Now clause (1) is satisfied (as suggested in Fig. 3).

4. Relation to Other Work

As already mentioned in the introduction, our work is closely related to [13]. In [13] three fairness-like notions are introduced:

(1) *Impartial execution:* along infinite computation sequences *all* moves appear infinitely often (no reference to being enabled or not).

(2) *Just execution:* along infinite computation sequences enabled moves, which once enabled remain enabled until taken (i.e., are continuously enabled), are eventually taken.

(3) *Fair execution:* along infinite computations sequences, moves infinitely often enabled are eventually taken.

This distinction influenced clause (3) of our rule. Without clause (3), our

rule is sound and complete for impartial execution.* The difference between just termination and fair termination is reflected in Examples 1 and 2 in Section 2.

A notable difference between our rule and the one in [13], called method *F*, is that we partition the moves in an ordinal-dependent way, whereas in [13] state predicates play a crucial rule in determining decreasing moves.

Now we show that our rule implies method *F*, and hence the semantic completeness of our rule implies the semantic completeness of method *F*.

Assume that for program C we found $(W, \leq)$, p, $\langle (S_w, D_w) \rangle_{w \in W,\ w>0}$ satisfying clauses (1)–(4) of our rule, relative to precondition r, and that $|D_w| = 1$.

In order to apply method *F*, we have to:

(i) Find a partial ranking function $\varrho : \text{States} \to W'$, where W' is ordered by a well-founded ordering, $\geq$.

(ii) Find predicates Q_i, $i = 1, \ldots, n$ over states, where $Q = \bigvee_{i=1}^{n} Q_i$, satisfying:

(0) $Q(\xi)$ implies $\varrho(\xi)$ is defined,

(1) $r(\xi) \supset Q(\xi)$,

(2) $Q(\xi) \wedge \eta \in C_i(\xi) \supset (Q(\eta) \wedge \varrho(\xi) \geq \varrho(\eta))$,

(3) $Q_i(\xi) \wedge \eta \in C_j(\xi) \wedge \varrho(\xi) = \varrho(\eta) \supset Q_i(\eta)$ for $i \neq j$,

(4) $Q_i(\xi) \wedge \eta \in C_i(\xi) \supset (\varrho(\xi) \gneqq \varrho(\eta))$ (thus the Q_i determine the decreasing directions),

(5) Program $C' :: *[\Box_{j=1,\ldots,n} B_j \wedge \neg B_i \to C_j]$ satisfies $\langle\langle Q_i \rangle\rangle C' \langle\langle true \rangle\rangle$.

To satisfy method *F*, take $W' = W$ (using the same ordering), and define $\varrho(\xi) = \min_w p(w)\xi$, $Q_i = i \in D_{\varrho(\xi)}$. Hence $Q(\xi) \equiv \exists w \cdot p(w)\xi$.

Next, we verify conditions (0)–(5) of method *F*.

Condition (0): $\exists w \cdot p(w)\xi \supset \{w \mid p(w)\xi\} \neq \emptyset$, and the minimum of $\{w \mid p(w)\xi\}$ exists by a property of the ordinals.

Condition (1): $r(\xi) \supset \exists w \cdot p(w)\xi$ holds by clause (4).

Condition (2): follows from clauses (1), (2) of our rule, guaranteeing that $p(v)$ holds for $v \leq w$; hence the minimal v s.t. $p(v)$ does not increase, either.

* *Note added in proof*: Daniel Lehmann informed us that a sound and complete version of our rule for just execution is obtained by replacing clause (3) by

$$(3') \qquad p(w) \wedge \neg B_j \supset \neg \bigvee_{i=1}^{n} B_i \quad \text{for } j \in D_w.$$

The resulting rule is complete for programs terminating under the following assumption upon the underlying semantics: each of the computation sequences generated is either finite or every guard is infinitely often tried.

Condition (3): $Q_i(\xi) \wedge \eta \in C_j(\xi)$, $i \neq j$, implies that an S-move is taken, and since $\varrho(\xi)$ is the minimal w s.t. $p(w)\xi$, this S-move does not decrease the ordinal, hence $Q_i(\xi)$ still holds.

Condition (4): follows directly from clause (1), since $\eta \in C_i(\xi)$ and $Q_i(\xi)$ imply a D-move is taken.

Condition (5): reduces to clause (3).

Acknowledgements

Amir Pnueli and Shmuel Katz are thanked for helpful discussions. Daniel Lehmann suggested the reduction of the [PSL]-rule to ours'.

References

[1] K.R. Apt and G. Plotkin, A Cook's tour of countable non-determinism, Proc. ICALP 81, Haifa (July 1981).

[2] E.W. Dijkstra, A Discipline of Programming (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[3] R.W. Floyd, Assigning meaning to programs, in: J.T. Schwartz (Ed.), Math. Aspects of Computer Science, Proc. Symp. in Appl. Math. (AMS, Providence, RI, 1967).

[4] N. Francez and A. Pnueli, A proof method for cyclic programs, Acta Informat. 9 (1978).

[5] N. Francez and W.P. de Roever, Fairness in communicating processes, University of Utrecht (1980).

[6] D. Gabbay, A. Pneuli, S. Shelah and Y. Stavi, On the temporal analysis of fairness, Proc. 7th POPL Conf. (1980).

[7] D. Harel, First Order Dynamic Logic, Lecture Notes in Computer Science, Vol. 68 (Springer, Berlin, 1979).

[8] P. Hitchcock and D. Park, Induction rules and termination proofs, in: M. Nivat (Ed.), Automata, Languages and Programming, IRIA (North-Holland, Amsterdam, 1973).

[9] C.A.R. Hoare, Communicating sequential processes, Comm. ACM 21 (8) (1978).

[10] L. Lamport, Proving the correctness of multiprocess programs, IEEE Trans. Software Engrg. 3 (2) (1977).

[11] L. Lamport and S. Owicki, Proving liveness properties of concurrent programs, SRI-TR (1980).

[12] A. Pnueli, The temporal semantics of concurrent programs, Theoret. Comput. Sci. 13 (1) (1981).

[13] A. Pnueli, Y. Stavi and D. Lehmann, Impartiality, justice and fairness: the ethics of concurrent termination, Proc. ICALP 81, Haifa (July 1981).

[14] W.P. de Roever, Dijkstra's Predicate Transformer, Non-determinism, Recursion and Termination, MFCS, 1976, Lecture Notes in Computer Science, Vol. 45 (Springer, Berlin).

[15] W.P. de Roever, A formalism for reasoning about fair termination, in: D. Kozen (Ed.), Proc. of the Workshop on Programming Logics, Lecture Notes in Computer Science (Springer, Berlin, 1981) to appear.

Algorithmic Languages, de Bakker/van Vliet (eds.)

Invited Address

ALGOL 68 Revisited Twelve Years Later or from AAD to ADA

Władysław M. Turski*

Warsaw University, Warsaw, Poland

* This paper has been omitted from the Participants' Edition at the explicit request of the author.