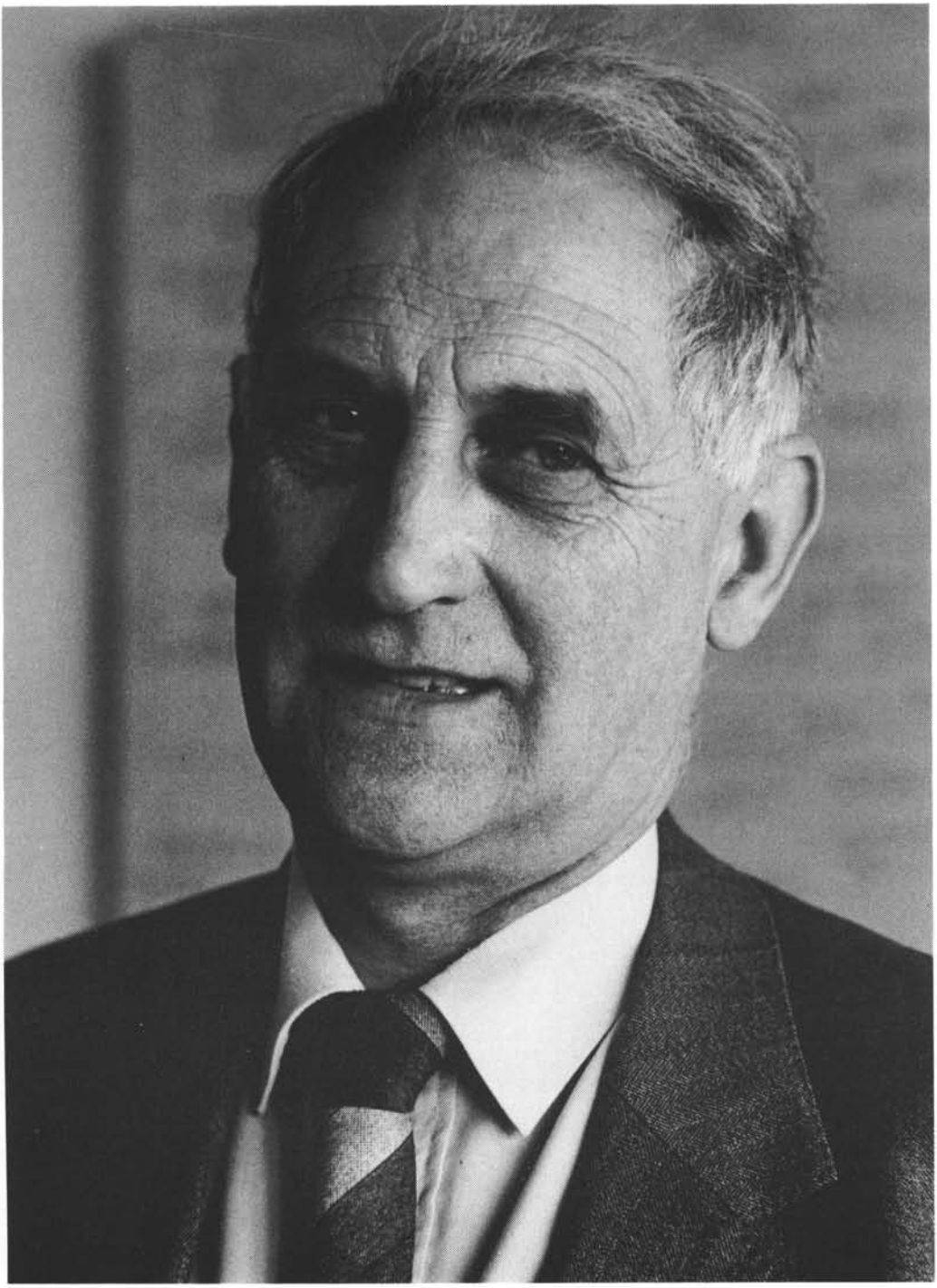
Liber Amicorum ac Collegarum

bij het afscheid van

Prof.dr.ir. W.L. van der Poel

26 oktober 1988

Vooruitgang bit voor bit



Vooruitgang
bit voor bit

Liber Amicorum ac Collegarum
bij het afscheid van
Prof.dr.ir. W.L. van der Poel
26 oktober 1988

Delftse Universitaire Pers / 1988

Uitgegeven door:

Delftse Universitaire Pers
Stevinweg 1
2628 CN Delft
Telefoon (015) 78 32 54

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Vooruitgang, bit voor bit : symposium ter gelegenheid van het afscheid van prof.dr.ir. W.L. van der Poel, 26 oktober 1988 / (samenst. C. Pronk ... et al.) - Delft : Met lit. opg.
ISBN 90-6275-493-7
SISO 520 UDC 001:681.3 NUGI 852
Trefw.: computerwetenschap.

Inhoudsopgave

Prelude

Beste Wim,

Dit boekje is samengesteld ter gelegenheid van je afscheid als bezoldigd hoogleraar aan de Technische Universiteit Delft. Met het bereiken van de VUT-gerechtigde leeftijd heb je je teruggetrokken uit de verplichtingen die een volledige werkkring met zich mee brengt.

Echter, het onderwijs geven zit je in het bloed en daarom is het voorbehoud in de eerste zin gemaakt; je hebt ons in mei jongstleden verlaten als bezoldigd hoogleraar, maar je bent direkt weer aangesteld als onbezoldigd hoogleraar met als leeropdracht het geven van onderwijs in je specialiteit: de theorie der programmeertalen, hetgeen zoals wij weten overeenkomt met Lambda calculus, combinatoren en Lisp.

Alle auteurs hebben het genoegen gehad tijdens hun carrière op de één of andere wijze tot je kennissenkring te hebben behoord; als collega-hoogleraar, als staflid, als student of op een andere wijze. Allen hebben de uitstraling van je enthousiasme ondervonden en hebben iets van je enthousiasme en kennis overgenomen. Helaas waren niet allen die waren uitgenodigd een bijdrage aan dit boekje te leveren daartoe in staat. Hopelijk is iedereen in de gelegenheid op de 26e oktober het afscheidssymposium en je officiële afscheidscollege mee te beleven.

De samenstellers van dit boekje willen hun dank uitspreken aan de auteurs van de bijdragen, T. Bosman voor het maken van de foto, de Delftse Universitaire Pers in de persoon van mevr. L.M.ter Horst-ten Wolde voor de verzorging van het drukwerk, S. Kalkman voor de hulp bij de realisatie van de omslag en, last but not least, mevr. G. M. A. Stoute-van de Garde voor het vele organisatorische en type-werk.

De samenstellers,
C. Pronk
W.J. Toetenel

CURRICULUM VITEA

Willem Louis van der Poel

Willem Louis van der Poel werd geboren te 's-Gravenhage op 2 december 1926. Hij behaalde het diploma HBS-B te 's-Gravenhage in 1944. In verband met de oorlogsomstandigheden ving hij zijn studie in de Afdeling der Natuurkunde van de Technische Hogeschool Delft aan in september 1945. In 1947 behaalde hij het Candidaats- en in 1950 het Doctoraal examen voor natuurkundig ingenieur. Reeds gedurende zijn studie gaf hij blijk van zijn grote research-kwaliteiten. Van 1947 tot 1949 was hij onder leiding van Prof. Dr. N.G. de Bruyn werkzaam als speurwerkassistent voor het Delfts Hogeschoolfonds met als onderwerp "Moderne automatische rekenmachines". In dezelfde tijd begon hij onder leiding van Prof. Dr. A.C.S. van Heel met het ontwerp en de constructie van een relais-rekenmachine. In 1948 nam hij deel aan een door de TH uitgeschreven prijsvraag over een onderzoek op het gebied van moderne rekenmachines. Voor dit werk ontving hij in 1949 een eervolle vermelding.

Direct na zijn studie aan de TH trad hij in dienst van de PTT bij het Centraal Laboratorium (later Dr. Neherlab.) om daar bij de Mathematische Afdeling onder leiding van Dr. Ir. L. Kosten (later Prof. Kosten) aan de ontwikkeling van moderne elektronische automatische rekenmachines te werken. De eerste elektronische rekenmachine van de PTT, de PTERA, kwam onder leiding van prof. van der Poel tot stand, later gevolgd door de ontwikkeling van de ZEBRA. Op de concepten en principes van deze, voor die tijd geavanceerde machines promoveerde hij op 1 februari 1956 cum laude aan de Gemeentelijke Universiteit van Amsterdam op een proefschrift getiteld: "The Logical Design of Some Simple Computers" tot doctor in de Wis- en Natuurkunde. Promotor was Prof. Dr. Ir. A. van Wijngaarden.

In 1956 volgde hij Prof. Kosten op als chef van de Mathematische Afdeling en in 1961 werd hij chef van het overkoepelende Schakeltechnisch Laboratorium. Prof. van der Poel publiceerde in die tijd en ook later veel over programma-ontwikkelingen, in het bijzonder op het gebied van talen en vertalerbouw.

In internationaal verband werden zijn grote verdiensten gewaardeerd, waardoor hij jarenlang voorzitter is geweest van de IFIP-werkgroep (International Federation for Information Processing) voor de ontwikkeling van ALGOL; tevens was hij voorzitter van de Technische Commissie voor Informatieverwerkende Machines van het Nederlands Normalisatie Instituut.

De contacten van Prof. van der Poel met onze Universiteit werden in 1962 opnieuw verstevigd door zijn benoeming tot bijzonder hoogleraar vanwege het Delfts Hogeschoolfonds met als leeropdracht "Toegepaste logica met betrekking tot constructie en gebruik van rekenautomaten". In verband met de snel toenemende onderwijs- en onderzoekaktiviteiten van onze TH werd die benoeming in 1966 omgezet in een buitengewoon hoogleraarschap, waarna Prof. van der Poel in 1967 een volledige betrekking als gewoon hoogleraar aan onze Universiteit aanvaardde.

Prof. van der Poel heeft binnen onze Technische Universiteit een zeer grote bijdrage geleverd aan het tot stand komen en de ontwikkeling van de studierichting Informatica. Hij begeleidde een groot aantal afstudeerders (meer dan 100) en was daarenboven promotor van acht promovendi.

Prof. van der Poel ondervond veel internationale waardering en erkenning. Hij werd o.a. in Engeland, de Verenigde Staten en Japan gevraagd voor een groot aantal gastcolleges. In 1972 werd hem een eredoctoraat aan de University of Bradford (UK) verleend. Tevens mocht hij in februari 1985 één der hoogste onderscheidingen op computergebied in the USA, namelijk de Computer Pioneer Award, ontvangen. In Nederland werden zijn wetenschappelijke prestaties ook erkend door zijn benoeming tot lid van de Koninklijke Nederlandse Akademie van Wetenschappen.

Naast zijn aktiviteiten op wetenschappelijk terrein ontplooide prof. van der Poel ook vele aktiviteiten op bestuurlijk gebied. Binnen de Universiteit is hij van vele commissies en raden lid, en in vele gevallen ook voorzitter geweest. Hiervan zijn te noemen: Bestuur Rekencentrum, Commissie Rekenmachines, Werkgroep Studieprogramma Informatica, Bibliotheekcommissie, vertegenwoordiger namens de TU bij de Sectie Informatica van de Academische Raad, Voorzitter van de vakgroep en Bestuurslid Afdeling Wiskunde. Daarnaast werd nooit tevergeefs een beroep op zijn medewerking gedaan voor talloze werkzaamheden die in het kader van een groot aantal ad-hoc commissies moest worden uitgevoerd. Op nationaal niveau is hij jarenlang bestuurslid en voorzitter geweest van het Nederlands Rekenmachine Genootschap. Internationaal heeft hij ons land ongeveer zeven jaar in de General Assembly van de eerder genoemde IFIP-organisatie vertegenwoordigd.

De aktiviteiten van prof. van der Poel op het gebied van communicatiemiddelen voor doofblinden verdienen bijzondere vermelding. Reeds in zijn PTT-tijd was hij aktief in het ontwikkelen van speciale apparatuur om met doofblinden te kunnen communiceren. Samen met Prof. Dr. Ir. H. Mol ontving hij daarvoor in 1959 de Visser-Neerlandiaprijs. Na zijn overgang naar onze Universiteit heeft hij zich veelvuldig met deze speciale problematiek beziggehouden. Veel vindingen, die met opzet niet geoctrooieerd zijn, waardoor doofblinden weer aan normale maatschappelijke processen kunnen deelnemen, zijn van hem afkomstig. Correspondentie tussen doofblinden onderling, zowel direkt als via telefoonlijnen ging tot de mogelijkheden behoren. Ook de communicatie tussen doofblinden en computers werd gerealiseerd. Mede door deze inspanning heeft ook jarenlang de doofblinde Dr. G. van der Mey een gewaardeerd medewerker van deze universiteit kunnen zijn.

Publicaties van W.L. van der Poel

1. Benadering van experimenteel gegeven functies door middel van veeltermen met de methode van Tchebycheff. (Approximation of experimentally given functions with polynomials of Chebysheff).
   Simon Stevin 4 (1948) 189
2. Schakelingen van automatische cijfermachines. (Circuits for automatic digital computer).
   Ned. Tijdschrift voor Natuurkunde 15 (1949) 255 -264.
3. A Simple Electronic Digital Computer.
   Appl Sci. Res. B2 (1952) 367 -400
4. Dead Programs for a Magnetic Drum Automatic Computer.
   Appl;. Sci. Res. B3 (1953) 190 -198
5. De werking van PTERA. (principles of PTERA).
   Het PTT-Bedrijf 5 (1953) 124 - 134
6. Het programmeren voor PTERA.
   Het PTT-Bedrijf 5 (1953) 135 - 147
7. Enige bijzondere onderwerpen uit de schakelalgebra. (some special topics in switching algebra).
   De Ingenieur (1955) no 1
8. De functie en werking van automatische rekenmachines. (purpose and function of automatic computers).
   De Ingenieur (1956) no 40, 96 -100
9. The Essential Types of Operations in an Automatic Computer.
   Nachrichten Technische Fachberichte 4 (1956) 144 - 145
10. The Logical Principles of Some Simple Computers.
    Thesis, Amsterdam (1956)
11. Ideas and Trends of Computers in Holland.
    Proc. Int. Automation Congress, Madrid (1958) 107 -110
12. The Simple Code for ZEBRA.
    Het PTT-Bedrijf 9 (1959) 31 - 66
13. ZEBRA, a Simple Binary Computer.
    Information Processing. Unesco, Paris. Oldenbourg (1959) 361 - 365
14. Symposium on the Logical Organization of very Small Computers.
    Ibid. 427 - 428
15. Micro-Programming and Trickology.
    In: Digitale Informations-wandler,

Ed. W. Hoffmann. Vieweg, Braunschweig (1961) 269 - 311

16. The Construction of an ALGOL Translator for a small Computer.
In: Symbolic Languages in Data Processing. Gordon and Breach (1962) 229 - 237

17. The Construction of an ALGOL Translator for a Small Machine. Contribution to a panel discussion on Techniques for Processor Construction.
In: Popplewell (Ed.), Information Processing 1962.
North-Holland Publ. Co., Amsterdam

18. Process for an ALGOL Translator. (with G. van der Mey, P.A. Witmans & G.G.M. Mulders).
Report 164 MA, 5 parts.
Dr. Neher Laboratory, Staatsbedrijf der Posterijen, Telegrafie en Telefonie, 1962

19. Talen en Kunsttalen (languages and artificial languages).
Inaugural Oration (1962) Waltman, Delft

20. SERA, een machine voor studiedoeleinden.
Informatie 27 (1963) 41, Stichting Studiecentrum, Amsterdam

21. A Procedure for String Handling. ALGOL-Bulletin 18 (1964) 47

22. SERA, a Simulated Machine for Purposes of Explaining Compiler Construction.
In: Proc. Symp. Inf. Processing Machines, 7-9 Sep 1964.
Czechoslovak Academy of Sci., Prague (1965 117 - 131

23. Recent Developments in the Construction of a New ALGOL.
Proc. of IFIP Congress '65. W.A. Kalenich (Ed.) 314.
Spartan Books, Washington D.C. 1965

24. Enkele theoretische aspecten van door processoren bestuurde telefooncentrales. (Some theoretical aspects of processor- driven telephone exchanges).
Het PTT-Bedrijf XV nr 3 (1967) 231 - 235. Also in De Ingenieur (1967) E141

25. Introduction to Panel on List and String Processing in General Programming Languages.
In: Symbol Manipulation Languages and Techniques.
D.G. Bobrow (Ed.) 340. North-Holland Publ. Co. Amsterdam 1968

26. Automatisch Programmeren.
Tijdschrift voor Efficientie en Documentatie 33 (1963) 314 - 317

27. The Sofware Crisis, some Thoughts and Outlooks.
In: Proc. IFIP Congress '68. North-Holland Publ. Co., Amsterdam (1968) 201 - 205

28. De nieuwe wereld en het rekenen. (The new world and computing).
De Ingenieur, 26 jan 1968, O22 - O24

29. The following articles are reprinted in: Colloquium Moderne Rekenmachine, Nederlands Rekenmachine Genootschap in Samenwerking met het Mathematisch

Centrum, 1969. (Issue on the occasion of the 10th anniversary of the Dutch Computer Soc.)

Welke opdrachten zijn essentieel in een rekenmachine?
(Which instruction codes are essential in a computer?
(from 1952) part1, 27 - 33

De PTERA. (from 1953) part 1, 88 - 94

Floating Addresses in an Automatic Digital Computer.
(from 1955) part 2, 19 - 27

Logical Structure of the ZEBRA. (from 1956) part 2, 48
-53

De invoerorganisatie van de ZEBRA. (Input organization
in ZEBRA) (from 1956) part 2, 75 - 81

30. SERA 69, definierend rapport samengesteld in opdracht van de SERA Commissie. Editor W.L. van der Poel.
Stichting Nederlands Studiecentrum voor Informatica, 1979. 74 pages.

31. SERA 69, a New Hypothetical Machine for Educational Purposes. Proc. IFIP World Conf. on Computer Education. Amsterdam 1970. Par III, p 209 - 214

32. Programming. Syllabus of a course on Automation for TELEAC, the Television Academy.
In: Automatisering, TELEAC, Delft (1967) 85 - 114

33. FORTRAN "Call" statement for the PDP-8 Disk System (4k).
Decuscope 9 (1970) no. 3, p6. Digital Equipment Computer Users Society

34. De programmeertaal TRAC.
Informatie, 13 (1971) 51 - 58

35. The Programming Language TRAC and its Implementation.
Proc. of a Computer Science Seminar, Sep. 20 - 21,
Stuttgart, Germany. IBM Stuttgart.

36. A Comparative Study of Some Higher Programming Languages. Advanced Course on Programming Languages and Data Structures,
June 1972, Amsterdam. 90 pages

37. Some notes on the history of ALGOL.
In: MC-25 Informatica Symposium, Mathematical Centre Tracts 37, Mathematical Centre, Amsterdam

38. Comment on the Composition of Semantics in ALGOL 68.
Comm. ACM 15 (1972) 772

39. Gegeneraliseerde macroprocessoren.
Verslag van de gewone vergadering der Afd. Natuurkunde,
Kon. Ned. Akad. v. Wetenschappen, 82 (1973) 54

40. Combinatoren en Lambdavormen.
Vakantiecursus Abstracte Informatica.
Mathematisch Centrum, Amsterdam 1973, Publ. VC 27/73

41. The Programming Language HILT.
Proc. of IFIP Working Conf. on Programming Teaching Techniques.
North-Holl. Publ. Co. 1973

42. Combinatorische Logica.
Informatie, 15 (1973) 667 - 676

43. Operating systems.
Inl. voordracht in symposium over Operating Systems door het NRMG.
Informatie, 16 (1974) 638 - 642

44. Tekenrijverwerking op computers.
Jaarboek Kon. Mij. Diligentia 1973

45. Machine Oriented Higher Level Languages
(Editor with L.A. Maarssen). North-Holl. Publ. Co. Amsterdam, 1974

46. Inleiding in de techniek van elektronische rekenautomaten.
Statistica Neerlandica 19 (1965) nr 4, p 167 -172

47. Combinatoren en Lambdavormen.
Verslagen Kon. Ned Akad. v. Wetenschappen, Afd. Natuurkunde, 84 (1975) no10,
172 - 179

48. MULTI8, a Real-time Multitasking Foreground/Timesharing Background
operating System for a DEC PDP/8.
In: J. Bell and G. Bell (Eds.),
Proc. of IFIP Working Conf. on Software for Minicomputers,
North-Holland Publ. Co. 1976

49. Minialgol.
Machine Oriented Languages Bulletin No.5 (IFIP) 1976, p 173 - 176

50. Parameterized batch files in OS/8.
DECUS 12 Bit Special Interest Group Newsletter, No. 31, Nov 1978, p 40 - 42

51. Reference Manual of the Language MIDIAL.
Report Info 1978-1, Dept of Math.,
Univ. of Technology, Delft, Nov. 1978. 72 pages

52. Software voor Microprocessoren.
Informatie 12 (1979) p 367 - 371

53. Transporteerbare Programmatuur.
Verslagen Kon. Ned. Akad. v. Wetenschappen, Deel 88 (1979) p 109 - 113

54. TI59 op tilt.
HCC Nieuwsbrief 23 (1980) p28

55. With C.E. Schaap and G. van der Mey,
New Arithmetical Operators in the Theory of Combinators.
Proc. Kon. Ned. Akad. v. Wet. a83(3), Sept (1980) p 271 - 325

56. Recursive procedures in Basic.
Computer Bulletin of the British Comp. Soc. II/42, Dec 1984, p22

57. One recursion operator is sufficient
To appear in Proceedings Royal Acad. Sci Oct. 1989

58. The mechanization of Lamda Calculus
36 pages, to be published

59. De ZEBRA Club, Causerie ter gelegenheid van het afscheid van Prof. dr. A. van der Sluis, 15 nov. 1988.
To appear in Liber Amicorum

Promovendi van W. L. van der Poel

1. J.A.C. Derksen
   QA4, A language for artificial intelligence.
   25 april 1973
2. J. F. Anthoni
   Multi-8, a real-time multitasking foreground/timesharing background operating system for a mini computer.
   1 mei 1975
3. K.A.G. Müller
   On the Feasibility of Concurrent Garbage Collection.
   3 maart 1976
4. M.J. Vliegenthart
   Het omzetten van teksten in Braille kortschrift met behulp van een rekenautomaat.
   24 maart 1976
5. A.J. Bequé
   LPR, An Integrated Program and Data Management System for Engineering applications.
   23 maart 1977
6. H. Oemrawsingh
   Studies on abstract machines and some related decision problems.
   1 juni 1977, Paramaribo
7. G.J. Holzmann
   Coordination problems in multiprocessing systems.
   14 juni 1979
8. J. van Katwijk
   The Ada - Compiler. On the design and implementation of an Ada compiler.
   15 september 1987

Afstudeerders van W. L. van der Poel

Nr	Naam	Jr	Titel afstudeerwerk
1	Okker DW	64	Een analyse van het opzetspel met behulp van een elektronische rekenmachine
2	Jager JJ	65	Toepassing van rekenmachines op het vierkleuren probleem
3	Roets PP	67	Rekursieve Macro-Samenstellers
4	Vlasblom G	67	Een assembler voor de PDP8
5	Brinkhuizen R	67	Formula manipulation with rational functions
6	Leentfaar R	68	A LISP compiler for LALGOL
7	Wilgenburg	68	Een LISP interpretator geschreven in ALGOL
8	Scheer LB van der	68	Een toepassing van ketenverwerkingstechnieken bij het beheer v. grote bestanden
9	Zijl RF v	68	De getallen van Van der Poel
10	Boer W de	69	Het simuleren van een analoog rekenproces op een digitaal computer
11	Barreveld HE	69	Een SNOBOL3 interpretator voor de X8
12	Taal AW	69	Een machine onafhankelijke SERA interpretator
13	Deckers F	69	De implementatie van Focal op een PDP9
14	Knoet CF	69	Symbolic Analysing Program for the PDP9
15	Brouwer AA	70	Een Elan Interpretator voor de assemblertaal Elan van de X8
16	Eijken GGJA v d	70	Ontwerpaspecten en verwezenlijking van een time sharing systeem
17	Jong P de	70	SAT, a snapshot and trace program for the PDP8
18	Saris WAG	70	Gecompileerde lambda-expressies in programmeercellen van het LISP systeem voor de PDP-9
19	Rijswijk JJ van	70	EULER interpretation voor de PDP9
20	Steen GJ van der	71	Semantische processen in kunstmatig intelligente systemen
21	Ververs F	71	Een SERA 69 assembler
22	Eek W van	71	Data-base management mbv Indexed tables
23	Katwijk J van	71	TRIC, a PDP9 implementatie van gemodificeerd Trac
24	Velsink WA	71	HORA
25	Houting JJ den	71	Een conversationele stuurkaarten generator
26	Janssen ThJBM	71	Compiler voor BCPL op de PDP9
27	Nieuwenhuis CJH	72	A Hybrid Interactive Formula Interpreting Programming System
28	Mevius JW	72	HILT, een programmeertaal gebaseerd op TRAC
29	Touwen L	72	Minicomputers in Nederland
30	Boesenkool R	72	Een disassembler voor de PDP-9

Nr	Naam	Jr	Titel afstudeerwerk
31	Hillegers LThME	72	Het programma Lab8 advanced averager+een 8k PDP8 interpretator voor een 4k
32	Schellart PAJ	72	De ombouw van PRA tot batch handler
33	Sandee R	72	Een BASIC compiler voor de PDP9 met behulp van STAGE2
34	Oostrom JWJ	73	Een BCPL compiler voor de P860
35	Hoenderkamp T	73	L6
36	Capel W	73	Dual monitor voor een PDP9
37	Jansen PL	73	Een inversie reductieproces voor grote matrices
38	Schimmel A	73	OVP, een overlay-package voor de PDP8 en andere systeemprogram
39	Bohm APW	74	Affixgrammatica's
40	Alink WFBF	74	Een Alias compiler in Alias
41	Visser GM	74	Een ALGOL 60 runtime systeem
42	Monteban L	74	Een implementatie van de programmeertaal TRAC voor de P860 minicomputer
43	Luiijf HAM	75	The language MIDIAL and its implementation on 8K PDP8
44	Gerbers J	75	An implementation of HILT with overlays
45	Boiten J	75	A portable MINIAL compiler
46	Verburg J	75	Systeemsoftware voor een PDP8 computer
47	Eijk PJC van der	76	Een Harvalias compiler in Harvalias
48	Hemerik C	76	Een eenvoudige ALGOL 68 compiler
49	Roos G	77	File protection in RT11
50	Goedings H	77	De implementatie van een eenvoudig operating systeem voor een Raytheon 704
51	Leynse Petronella	77	Een open identificatie model
52	Hoed MA den	77	Een LISP interpretator voor de PDP11
53	Hoof RRM van	78	Implementatie van ML/I mbv STAGE2
54	Zuidam H	78	Koppeling van minicomputers aan een DEC systeem10 door synchrone transmissie
55	Wijk OW van	78	Bootstrap of the Pascal P compiler on a Harris/4
56	Schaap CE	79	Unary number systems in combinatory logic
57	Westerman RIAM	79	Een bijdrage aan de computerverwerking van bouwnormen
58	Hedel HWM van	79	Mode handling in ALGOL 68
59	Holtz Edith B v	79	Codegeneratie in de ALGOL 68 D compiler
60	Mesman AJ	79	Het RT11 operating system onder UNIX

Nr	Naam	Jr	Titel afstudeerwerk
61	Zanten A van	79	Report of a minial compiler on UNIX
62	Eersten WJ v d	80	The Portable Language MIDIAL. Implementation on the UNIX time sharing system
63	Cohen PG	80	Priemgetallen onderzoek mbv recurrente rijen
64	Hermsen JF	81	Syntax directed error recovery for LR parsers
65	Langelaan CA	81	TRAC implementatie
66	Bal HE	82	Overloading resolution in DAS
67	Bosman JCP	82	KMC Development system
68	Kamphorst WHJ	82	The portable language MIDIAL, report of a portable MINIAL compiler
69	Loon CA van	82	The Portable Language MIDIAL. Completion of the impl. on the UNIX tss
70	Someren J	82	Computation of adressing information in de DAS code generator
71	Vonk Jeanette C	82	DAS Definition
72	Ostermann Irma MA	82	A CHILL interpreter
73	Hoek Jacoba J vd	82	High level Forth (HILF). Definition and compilation
74	Pauw WS de	83	The first code generator pass of the DAS compiler
75	Niekel RBM	83	The C programming language in host/target software development for microcomputers
76	Kampen R van	83	A PDP10 code generator for CHILL
77	Derksen J	83	Een ontwerp van programmeergereedschap voor dialoogsystemen
78	Barkey JA	83	Using PCC's backend for generating code in the DAS compiler
79	Groothuizen MR	83	OS1, an operating system for Z80 microcomputers
80	Oort GJ	84	Flush
81	Vollebregt FRE	84	Flush
82	Toetenel WJ	84	Code generation for expression in the DAS compiler. Some special topics
83	Bertz PR	84	McG, Its users guide and the description of its implementation
84	Dijkstra K	84	Een symbolische DAS Debugger
85	Hoekstra F	84	PDP11/60 user microprogramming under UNIX
86	Konijnenburg E	84	Retargeting the DAS compiler with the EM machine model
87	Niet M de	84	Een nieuw front-end voor de DAS compiler
88	Oorschot J v	84	Quest, een uitbreiding op een terminal emulatie pakket
89	Schelvis MAJ	84	Wsh, a multi-stream multi-window textual user-interface
90	Schiet JJC	84	The portable language MIDIAL

Nr	Naam	Jr	Titel afstudeerwerk
91	Druzdzel MJ	85	Implementation of the memory management module of the UNIX 5 on a MC68010
92	Werry PJM	85	Een MIDIAL naar IC compiler geschreven mbv META4
93	Dekkers R	86	Een implementatie van de kernel van MS-DOS 2.0
94	Moermans WJA	86	CHIPO, A table driven machine independent peephole optimiser for CHILL
95	Schurink MR	86	UDPX. Inleiding en documentatie van de initiatie programmatuur
96	Groeneveld M	86	VEYACC, de Very Extended YACC
97	Hekken MC van	86	Boomcompiler. Een compiler voor specificaties van intermediaire boo
98	Spee P	86	RISC, Reduced instruction set computer architecture
99	Verseveld O	86	IDEFINE, a graphical editor for making IDEFO diagrams
100	Ysebrand H	86	Een GANDALF editor voor EDISON
101	Karaiskos Z	86	Implementation of the MP/M on the TRS-80 Model II
102	Oosterhof R	86	Het ontwerp en implementatie van de simulatietaal THORG
103	Varkevisser P R	87	A work station manager
104	Grivel E	87	Code Generators, an investigation of different techniques
105	Eggink F	87	Een real time multitasing host target debugger
106	Moermans AM	88	ST Mini-UNIX. Impl. of MiniUNIX on the ATARI ST
107	Diertens B	88	An MC68020 Code Genrator with the use of CGSS
108	Berg T vden	88	Using the Amsterdam Compiler KIT
109	Wesselius JH	88	De implementatie van Smalltalk 80 en Methods
110	Leeflang WJ	88	Terminatieproblemen in lambda calculus en combinator theory
111	Bekkers WB	88	Uitbreidingen aan de combinatoren en lambda machine
112	Steenweg WL	88	An implementation of LISP in C
113	Ooyen WO van	88	WATS Een Ada Tasking Supervisor en een model voor Ada Tasking op een multiprocessor
114	Landzaat, M.J.	88	Implementation and use of the Code Generator Generator CGYACC
115	Ganswijk WJ van	88	A register usage delimiter for the intermediate code of a compiler

The Evolution of System Implementation Languages

R. D. Huijsman
C. Pronk

Faculty of Mathematics and Computer Science
Delft University of Technology
Delft
The Netherlands

1. Introduction

System Implementation Languages (or SILs) have been a research subject at the Computer Science Department for a long period of time. The purpose of this contribution to the valedictory symposium of prof. dr. ir. W. L. van der Poel is to give an overview of the evolution of these languages. Where appropriate, references to the work done at our faculty will be made in this paper.

The structure of the remaining part of the paper is as follows: we first give a global overview of SIL history; next, we separately discuss, in relation to SILs, the subjects (i) control abstraction, (ii) data abstraction, (iii) concurrency, (iv) program structuring/modularisation, and (v) object-oriented languages; finally, we end with an attempt to formulate some conclusions.

2. History of System Implementation Languages

System Implementation Languages form a section of the more general area of programming languages. Their development started in the late sixties and the name used in those days: "Machine Oriented Higher Level Languages" (or MOHL's), clearly indicates that the visibility of the underlying machine was a major difference between MOHL's and the more abstract High Level Languages developed then.

A landmark in the development of MOHL's was the IFIP conference on Machine Oriented Higher Level Languages held in 1973 in Trondheim (Norway). The proceedings of that conference, edited by W. L. van der Poel and L. A. Maarssen [Poel-73], gave a critical examination of about 20 languages being developed or used at that moment.

It is not easy to give a precise definition of what exactly constitutes a SIL or MOHL. In [Wulf-73] Wulf tries to give some definitions of which the preferred one (according to the insights of the authors of the present paper) is:

> to replace almost all uses of assembly language not covered by
> the 'general purpose languages', e.g. Algol.

SILs are typically used for writing systems programs such as operating systems, compilers and the like. Systems built this way are characterized by Wulf [Wulf-73] as follows:

— They involve the interactions of several algorithms; the efficacy of the system depends on the interactions as well as on the individual algorithms.

— Many of the problems ... relate to the simple fact that these systems are not used under the paternal supervision of their authors.

— These systems must operate on a variety of configurations and load conditions.

Summarizing, the application area of SILs can be best described by what is currently called "embedded systems".

A successful MOHL design should satisfy the following requirements (partly according to Wulf):

— The quality of the produced code should approximate the quality of the code produced by a capable programmer using assembly code.

— The level of the language should be high, resulting in reduced complexity of programs and simplified communication between programmers working on the same job.

— The portability of applications to other configurations should be acceptable.

— A (restricted) access to the underlying hardware should be possible.

— Only a small run-time system should be required.

— A graceful way of handling exceptional situations should be provided.

As some of these requirements are contradictory, a delicate balance needs to be achieved.

In the book referred to above, many languages have been discussed. Only a small number of these languages have stood the tooth of time, some have been the basis for further developments and only a few of those presented in 1973 are still in use. The accompanying graph (Fig. 1) gives an impression of the evolution of some SILs.

Such a graph inevitably gives an incomplete picture of reality:

— To show the evolution of some SILs, a few languages whose usefulness as a SIL could be questioned, were added to this graph.

— A link between languages is meant to indicate that the newer language was based on the older one; the absence of a link may not be interpreted as the negation of this relation.

— The presence of a language in this graph means that the language is considered important in the light of this symposium; some languages were added because some students worked on them under the supervision of our young emeritus (Alias, Harvalias, Midial, Minial).

— The languages are placed in a time perspective as far as possible according to the year of appearance of their defining report. Not always could such a report be traced.

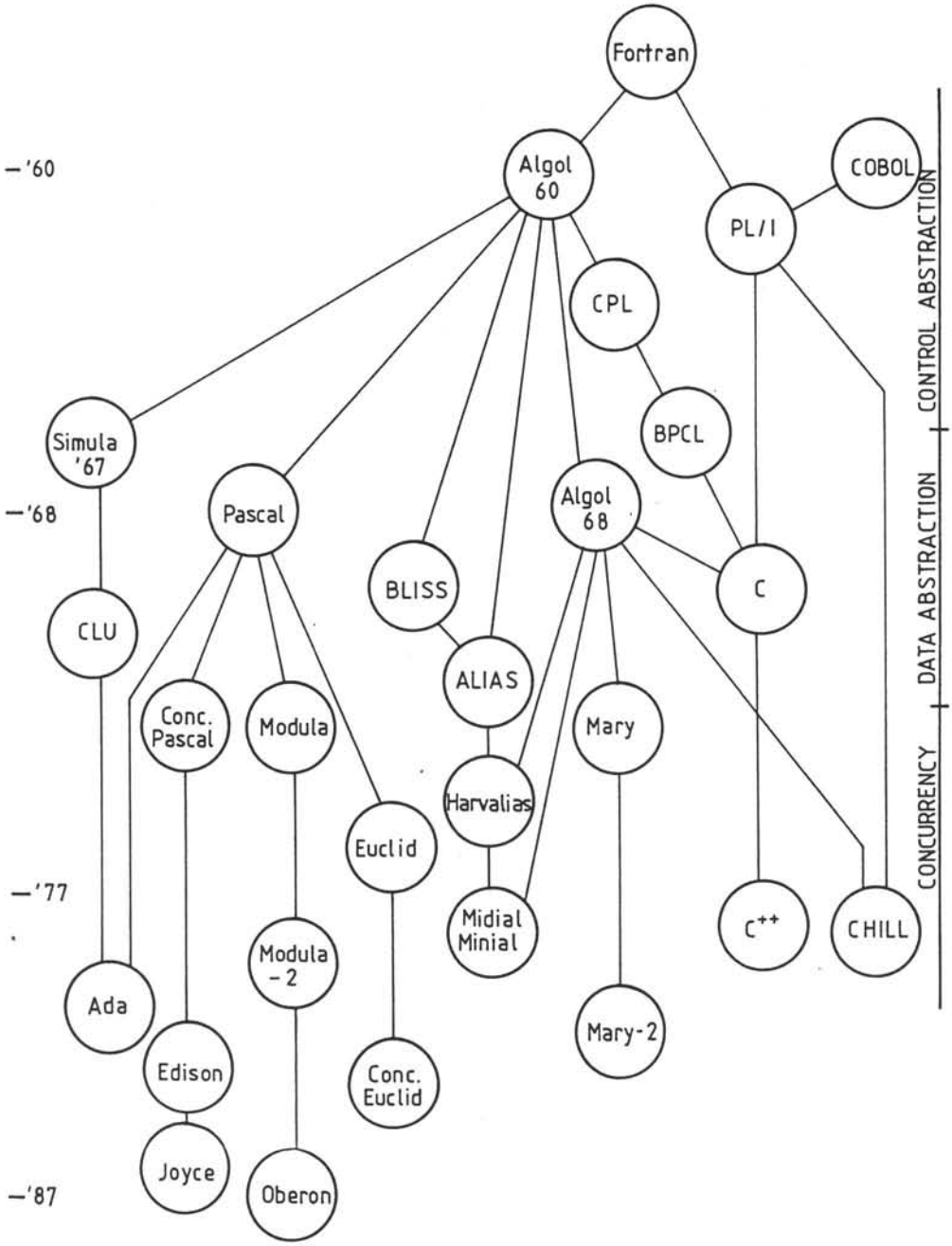

Fortran
Algol 60
COBOL
PL/I
CPL
BPCL
Simula '67
Pascal
Algol 68
BLISS
C
CLU
ALIAS
Conc. Pascal
Modula
Mary
Harvalias
Euclid
Midial Minial
C++
CHILL
Modula -2
Ada
Mary-2
Edison
Conc. Euclid
Joyce
Oberon
—'60
—'68
—'77
—'87
CONTROL ABSTRACTION
DATA ABSTRACTION
CONCURRENCY

Fig. 1

Most languages presented in this graph are well known; those being particular to Delft will be elaborated upon here:

— Alias was an early MOHL being developed by Barreveld e.a. [Barreveld-73]. It was influenced by Algol-60.

— Harvalias [Eijk-73a,b] was a SIL inspired by BLISS [Bliss-70], Alias and Harvey. Control structures were Algol-68-like; the implementation was done on a PDP11/45 and should be easily portable to (byte addressable) machines.

— The design of the languages MIDIAL and MINIAL (Mini-Algol) [Poel-78] was heavily influenced by Algol-68. High expressiveness of the language, conciseness of the programs written in the language and the ease of porting the compiler were major considerations during the design of the language. Several Masters Theses are available [Eersten-80, Loon-82, Schiet-84].

— Ada, being a widely known language, has been the subject of research by staff members of the group of prof. van der Poel during a number of years. This research had led to a doctoral thesis [Katwijk-87] and numerous papers. Research on several aspects of tasking is carried on [Toetenel-88, Ooyen-88].

— Modula-2 [Wirth-83] is currently being standardized by an ISO-committee. One of the authors of this paper is actively participating in this standardization effort.

— Oberon, a new language developed by Wirth [Wirth-88] has introduced a new concept of "type extension". Currently an implementation of Oberon is part of a students assignment.

A striking, general characteristic of SILs is their imperative nature, which, however, can be explained by the fact that all of them, directly or indirectly (by their very "descendance"), are based on conventional machine architectures. These architectures, called "von Neumann" architectures, are an agreed upon obstacle to efficient implementation of languages embodying other, perhaps more appealing computational models, as explained in [Backus-78].

Regarding the evolution of SILs, one can hardly deny the fact that this evolution is to a large degree not specific for this category of languages and rather reflects the general evolution in thinking about programming and programming languages. However, what is definitely specific, is the change in the way the demands of direct access to the hardware machine are satisfied.
What we see here is perhaps characterized best as elevation of the conceptual level at which direct interaction with the underlying hardware takes place and an attempt to unify low-level and high-level features through integration of direct machine access into existing higher forms of abstraction (Ada [LRM]): identification of interrupts with entry calls, representation specification for otherwise typed data, type checking of pointers, etc..

Last, but not least, one cannot, of course, separate the evolution of SILs from hardware developments. In particular the rapid succession of ever faster processors and ever larger memories at continuously falling prices put the demand for efficiency in a

quite different perspective. As a matter of course this affected its weight as a factor in the production of systems software, and as such also, inevitably, left its traces in SIL architecture.

3. Control abstraction

Programming in assembler is characterized by direct use of the hardware machine with hardly any abstraction at all: algorithms are expressed as sequences of machine instructions, chained by jumps. Although this offers the programmer an almost complete freedom in defining the thread of control, it entails at the same time non-neglectable risks: in most cases the —low— level of abstraction bears an unfavourable relation to the complexity of the problem at hand and as such makes it difficult to maintain an overall view.

Since about 1968 there has been a growing awareness of the difficulty to keep the implementation of ever larger and complexer systems software under firm and safe control. Also the rather low productivity of assembler programming in relation to the rising demand for software was becoming more and more a factor of importance. All this fostered the development of languages containing so-called control structures intended to raise the level of abstraction in programming without compromising the efficiency of the resulting programs.

This evolution, which, of course, cannot be considered typical for SILs only, initially comprised no more than codification of a number of existing, in assembler programs frequently occurring, execution patterns. The overall picture is one of containment of the "goto", which was, in its unrestricted form, nothing more than a plain machine jump in disguise, and as such was considered a source of problems [Dijkstra-68]. Dijkstra's stigmatizing of the "goto" was a strong incentive for a development as a result of which the construction either disappears, as in Modula-2 [Wirth-83] and Edison [Brinch Hansen-81/83], or survives in a very weakened form. The frequently occurring applications are now expressed by specialized sequencers like **exit** in Ada [LRM] or "single-entry/single-exit" constructs like **while** in Pascal [Jensen-75], while the most dangerous manifestations have been tamed by constraining their application and turning them into constructs which fit to higher forms of abstraction, as e.g. the exception in Ada and CHILL [CHILL-84].

In spite of all this, the subject of presence and use of "goto" now and then gives rise to fresh debates [Creak-87, Rubin-87], some implementers judging it hardly possible or even impossible to make efficient programs without "goto"'s, referring to the pretended inefficiency of the "single-entry"/"single-exit" constructs: these would entail unnecessary auxiliary variables, code duplication and call overhead [Fairley-85].

A subject deserving special attention in this context is the phenomenon of procedural abstraction. Different from the above described abstraction forms procedural abstraction is expensive because its use doesn't fit well together with the possibilities of conventional and still common hardware. Consequently, this form of abstraction is at odds with the —particularly for SILs— dominating requirement of efficiency. It shouldn't surprise, then, to see various languages exhibit features intended as viable compromises between the

advantages of procedural abstraction and the drawbacks of the procedure call. These features vary from lexical means like macro's in C [Kernighan-76] and PL/1 [PL/1] to implementational shortcuts as in-line expansion in Ada.

As an additional complicating factor one can mention the possibility of recursive interpretation of procedure definitions. On the one hand the possibility of recursion raises the level of abstraction as it enables many algorithms to be expressed in a much more natural manner. On the other hand it affects efficiency in a negative way because of the accompanying call overhead combined with the impossibility to exploit compromises like macro's or in-line expansion.
This becomes most manifest if one considers the phenomenon of tail recursion: here the loss of efficiency is very striking because this form of recursion is semantically equivalent to iteration which can be expressed in imperative languages by very efficiently implementable, specialized, constructs. Accordingly the view taken in various languages towards the phenomenon of recursion is not a very uniform one, varying from absence, as in FORTRAN [FORTRAN] through support when specified by the programmer, as in PL/1, to plain presence, as in Pascal, C, Modula-2, Ada etc..

4. Data abstraction

Developments in the usage of data in programs were to a large extent analogous to those in the area of sequence control. With the advent of Algol 68 [Wijngaarden-76] and especially Pascal, the programmer was able to abstract from the lack of structure and the peculiarities of conventional computer memory. The presence of a number of primitive, scalar "types" (integer, boolean, real, character) and the availability of static composition mechanisms made it possible to use data at a higher conceptual level than that embodied by a sequence of bits, bytes or words. Of course there is a clear and undeniable relation between this form of abstraction and the earlier discussed procedural abstraction, which as a matter of fact is indispensable for the definition of operations on data at the appropriate level of abstraction.

Comparable to the developments in the area of control structuring, data structuring in its infancy wasn't much more than only a reflection of common patterns of use: homogeneous as well as heterogeneous sequences of related data were expressible directly as arrays c.q. records, while, moreover, orthogonal composition of the two made it possible to construct entities of a still higher level of abstraction.

At the same time we see developing the notion of "type", initially to be understood only in a structural sense, i.e. type being a designation of a particular structure, together with the notion of "type checking", the act of checking for domain incompatibilities in expressions, in order to prevent —inadvertent— misuse of data during execution. Initially the latter didn't go hardly beyond counteracting direct access to the representation of scalar types (thereby ruling out a *shift left* as integer multiplication), preventing violation of scalar type abstractions (no addition of Booleans), and enforcing the usage of specialized selection operators to access structure components (e.g. indexing).

Later on, the type concept gradually evolves from a denotation of structure to a denotation of real abstraction: abstract data types in combination with name equivalence, as opposed to structural equivalence, as a criterion for type compatibility. Every type is considered to represent a unique collection of values together with a collection of applicable operations, which makes it possible to protect programmer defined abstractions in the same rigorous way as scalar, predefined types. Type checking secures the representation of defined abstractions against arbitrary access, manipulation being possible only via specifically defined interface operations. Syntactical expression of the concept is achieved by introduction of scope controlling encapsulation constructs which proved to be equally useful for other purposes (see the paragraph on modularisation). An illustrative example is the ubiquitous stack, here presented in Ada style, without implementation:

```
package STACK_ADT is
  type INT_STACK (SIZE: NATURAL) is limited private;
       -- only access via defined, visible operations
  procedure PUSH (S: in out INT_STACK; V: in INTEGER);
  procedure POP  (S: in out INT_STACK; V: out INTEGER);
  function TOP (S: in INT_STACK) return INTEGER;
  function IS_EMPTY (S: in INT_STACK) return BOOLEAN;
  function IS_FULL  (S: in INT_STACK) return BOOLEAN;
  OVERFLOW: exception;
  UNDERFLOW: exception;
private
       -- some otherwise hidden stack representation
end STACK_ADT;
```

As far as the overall picture is concerned, there is a clear parallel with control abstraction, the pointer playing the role of the "goto": we see a development in which the pointer loses much of its significance and thereby its dominating position. Pointer arithmetic, generally recognized as being severely error-prone, becomes unnecessary in many cases and the pointer itself turns from an indispensable instrument for efficient access of static structures to a means for construction of dynamic or recursive data structures which cannot be expressed by existing composition mechanisms.
In this context, making the pointer itself subject of type checking, as is done in Ada, should be considered as an attempt to extend name equivalence and type checking to dynamic structures as well.

It is clear, however, that, although features for data structuring and data abstraction elevate the conceptual level of programs, and thereby their clarity, the degree of incorporation in SILs of the above mentioned developments, as well as the specific nature thereof, is determined largely by their being efficiently implementable. Consequently, advancements in the area of code generation and optimization technology can be expected to have affected SIL architectures in the course of years.

As a matter of fact, then, younger SILs frequently offer specific constructs c.q. definition mechanisms which advance abstraction without loss of efficiency, thereby eliminating the need to use domain incompatible operations. An illustration of this is to

be found in Ada, where the so-called low-level features enable specification of location and layout for packed data structures to the very bit level, without precluding the latter from being manipulated as normal abstractions. Self-evidently, this means that part of the burden resulting from the SIL efficiency requirements shifts from the SIL programmer to the SIL implementer.

It must be clear, that the possibility of —mechanical— optimization can also affect SIL architecture in such a way as to make irrelevant the presence of certain features enabling optimization by hand, during programming. That this wasn't always recognized as such, however, is exemplified by a complaint about Pascal [Conradi-76], saying that the impossibility of passing string constants as **var** parameters leads to inefficiency, and the accompanying suggestion of a *ref val* mode as a means for prevention.

5. Program Structuring/Modularization

Together with the introduction of procedural abstraction we also see scope rules make their appearance, since it was necessary to distinguish local variables from non-local ones. It does hardly surprise to see SILs adhere to the rules of lexical or static scoping, as these rules enable almost all necessary checks to be carried out during program translation, thereby leaving execution time efficiency practically unaffected.

As a logical, next step static scope rules were subsequently extended to the procedural objects themselves, in this way creating the nowadays classical combination of block structure and static scoping in imperative languages. However, despite the clear program structure resulting from these concepts, programming practice in Pascal-like languages revealed some undeniable drawbacks:

— The partitioning of program source into independently compilable units —a very desirable and useful feature when engineering large, complex systems— is impossible without seriously impairing the strong type checking which is possible in the case of a single, monolithic text.
This circumstance not only forces a program to be compiled in its entirety, even when only minor changes have been applied, but it also severely hampers the safe use of libraries and a fortiori the reuse of software in general.

— The structure of nested procedures together with the relation between scope and lifetime of objects on the one hand, and the need for statically allocated local procedure variables on the other hand, leads to global declaration of procedures and variables with the unavoidable consequence of unwanted visibility, or gives rise to language features like "static" (C, PL/1) and **own** [Algol60] declaration of variables.

The solution to these problems was supplied by encapsulation constructs of the same kind as applied for the implementation of the meanwhile matured ideas about abstract data types. The essence of these new scope regulating constructs lies in their structural separation of interface and code. The interface part or specification contains a.o. procedure headers and describes the user visible entities or "exports"; the code or implementation part contains the definition of all visible entities as well as necessary, auxiliary operations and non-local variables whose existence must remain hidden. Making

these constructs separately compilable entities within the framework of strong type checking and allowing them to be selectively imported, eliminates the need for unwanted global declarations or **own**-like features. To illustrate the previously introduced concepts the following example of a random number generator is given both in Pascal and Modula-2:

```
{ Random Number Generator, PASCAL Style }
program random(input,output);
        var     seed : integer;         { global, so visible everywhere }
        function random : integer;
        const a = 333; b = 6925; modulus = 32767;
begin
        seed := (a * seed + b) mod modulus; { random, more or less }
        random := seed
end;

begin                                   { main }
        seed := 123;                    { start value }
        while true do                   { continue forever }
                writeln(random : 7)
end.
```

```
(* Random Number Generator, MODULA-2 Style *)
DEFINITION MODULE randomize;
        EXPORT QUALIFIED random;    (* EXPORT only necessary in Wirth-2 *)
        PROCEDURE random () : INTEGER;
END randomize.

IMPLEMENTATION MODULE randomize;
        VAR seed : INTEGER;     (* local to this module, invisible outside module *)
        PROCEDURE random () : INTEGER;
                CONST a = 333; b = 6925; modulus = 32767;
        BEGIN
        seed := (a * seed + b) MOD modulus;
        RETURN seed
        END random;

BEGIN                                   (* initialization code *)
    seed := 123
END randomize.
```

```
MODULE testrandom;
      FROM SysStreams  IMPORT sysOut;
      FROM TextIO      IMPORT WriteINTEGER, NewLine;
      FROM randomize   IMPORT random;
BEGIN
    LOOP
        WriteINTEGER(sysOut, random());
        NewLine(sysOut);
    END
END testrandom.
```

The above described module concept, with its syntactical expression in the form of interface/code pairs, is present a.o. in the SILs Modula-2 (DEFINITION MODULE and IMPLEMENTATION MODULE) and Ada (**package** and **package body**), with otherwise significant differences in scope regulating power.

A complete program can be constructed with the help of a large number of separately compiled interface/code pairs ("modules"), strong type checking of the whole being secured by the enforced respect of the partial ordering as implied by all module imports (Fig. 2).

The fact that module constructs are efficiently implementable will undoubtedly have been a major factor contributing to their incorporation in SILs. All interface checks can be done at compilation time, although repeated opening and closing of files, as mentioned in [Conradi-85], can cause a considerable system load. However, careful implementation can reduce run-time penalties to practically nil, the effect of using the module concept manifesting itself mainly in unnecessary large executables, containing exported routines which are not imported by the exporting modules' clients. But, fortunately, even this can be remedied by the use of special, intelligent linkers [Cough-86].

6. Concurrency

The programming of general purpose operating systems and special purpose real-time kernels (e.g. for process control) constitutes an important application area for SILs. In this area the ability to model concurrency in the higher level language can be advantageous. In many language designs which have appeared since 1970, experiments with the introduction of concurrency as a language concept have been performed. The following table gives a résumé:

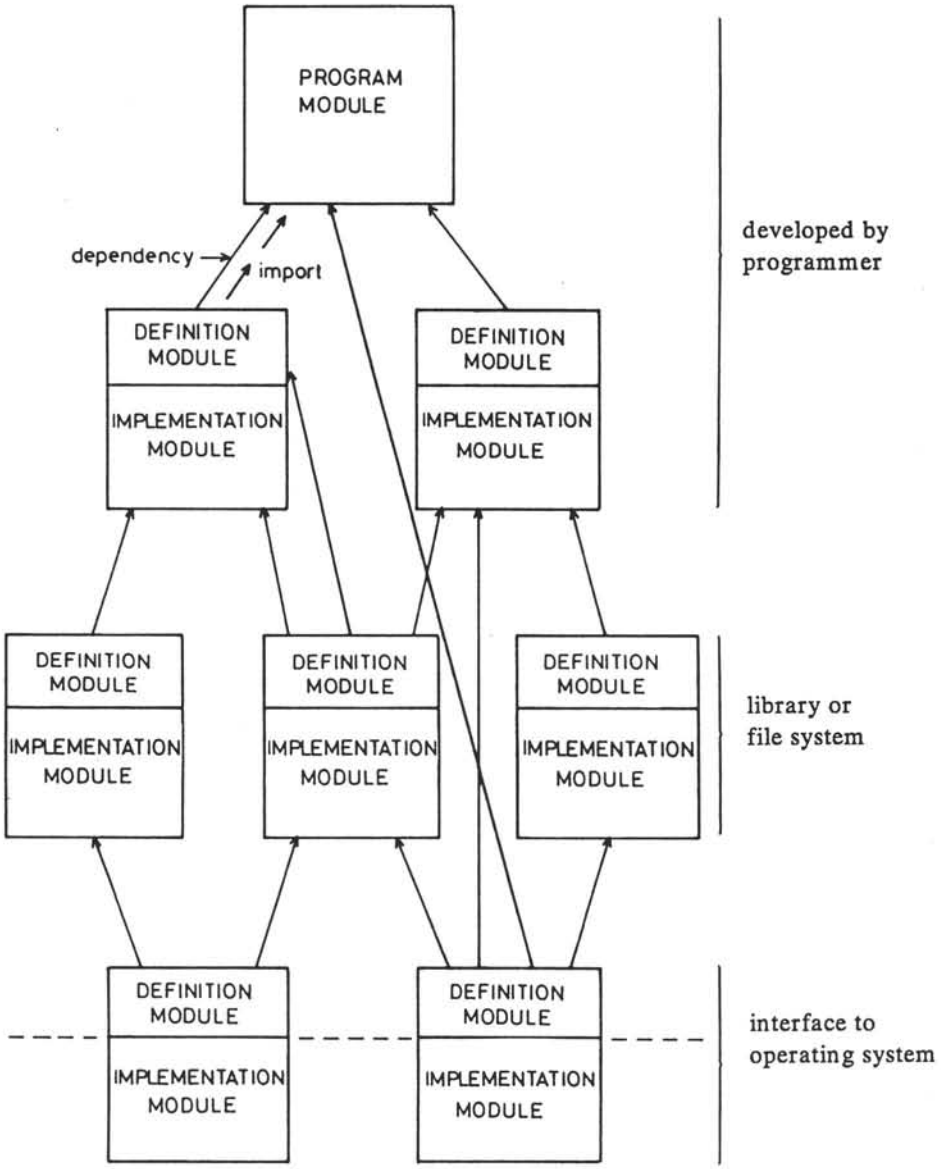

PROGRAM MODULE
dependency
import
DEFINITION MODULE
IMPLEMENTATION MODULE
DEFINITION MODULE
IMPLEMENTATION MODULE
DEFINITION MODULE
IMPLEMENTATION MODULE
DEFINITION MODULE
IMPLEMENTATION MODULE
DEFINITION MODULE
IMPLEMENTATION MODULE
DEFINITION MODULE
IMPLEMENTATION MODULE
DEFINITION MODULE
IMPLEMENTATION MODULE
developed by programmer
library or file system
interface to operating system

Fig. 2

		concurrency mechanism	synchronization mechanism
Algol 68	[Wijngaarden-76]	cobegin/coend, par	semaphore
Conc. Pascal	[Brinch Hansen-75]	process	monitor
Edison	[Brinch Hansen-83]	cobegin/coend	cond.crit.region
Modula	[Wirth-76]	process	signal
Modula-2	[Wirth-83]	coroutine	monitor
Ada	[LRM-83]	task	rendez-vous
Occam (2)	[Occam-84]	process	rendez-vous

There is no genuine need however, to introduce concurrency into a language. The 'C' language e.g. has no concurrent features, although there are many implementations where concurrency, mostly in the form of coroutines, has been added to the language using library calls [Ridder, Binding-85].

Regarding the required efficiency of a SIL, it can be shown that the overhead incurred by task switching or synchronization cannot always be neglected. Huijsman e.a. [Huijsman-87] show that implementing the Ada rendez-vous mechanism requires much care. It can be questioned whether the objective of the Ada language ("suitable for embedded systems") has been met. In fact, there are considerations to look for other concurrency and synchronization mechanisms. A first impulse to this discussion was given by Toetenel [Toetenel-88].

The foregoing paragraphs considered SILs for single processors; multiprocessors entail many currently unsolved problems. These problems can be solved by either introducing appropriate constructs in otherwise sequential languages (Ada, Chill) or by designing parallel languages from scratch [Occam]. Research on this subject is very active, however, reporting on this research is considered to be beyond the scope of this paper.

7. Object-Oriented Languages

The next step in the development process of general purpose languages is the introduction of "object-oriented programming". Under this name many languages are now being advertized (Obj-C [Cox-86], Eiffel [Meyer-88], C++ [Stroustrup-83]). According to [Stroustrup-88, Pascoe-86], object-oriented languages should possess the following properties:

— Encapsulation constructs to allow information hiding and data abstraction,

— Inheritance to enable programmers to create classes and subclasses allowing declaration of objects which are specializations of other objects,

— Late binding of a message to a class to attain maximum flexibility by providing a kind of polymorphism.

From these criteria it can be seen that true object-orientedness implies garbage collection

which is thought to be incompatible with the requirements for SILs. In our opinion there will be no developments of SILs towards true object-orientedness. However, partial object-orientedness as is already exemplified in languages as Modula-2, Oberon, Ada and C++ will be developed further.

8. Conclusions

Observing the development of recent SILs, one can state that the distinction between SILs and the more general purpose, higher level languages has become blurred. This is caused mainly by the provision of facilities for access to the underlying machine on a high level of abstraction.

Furthermore, adding concepts like concurrency to an already fully developed sequential language will not always give satisfactory results. A development from scratch is found to lead to more coherent results. Particularly, it is believed that the development of SILs for non "von Neumann" architectures should be tackled along these lines.

9. References

Algol-60 — Revised Report on the Algorithmic Programming Language Algol 60
Computer Journal, Jan. 1963

Backus-78 — Backus, J.
Can Programming Be Liberated from the von Neumann Style?
A functional Style and its Algebra of programs.
CACM, Vol 21 No 8 pp 613-641 (1978)

Bailes-85 — Bailes, P.A.
A Low Cost Implementation of Coroutines for C
Software Practice and Experience, Vol 15 pp 379-395 (1985)

Barreveld-73 — Barreveld H.E.
ALIAS
in [Poel-73]

Binding-85 — Binding, C.
Cheap Concurrency in C
Sigplan Notices, Vol 20 no 9 pp 21-26 (1985)

Bliss-70 — Bliss Reference Manual
Computer Science Department
Carnegie-Mellon University, Pittsburg (1970)

Brinch Hansen-75 — Brinch Hansen, P.
The Programming Language Concurrent Pascal
IEEE Trans. on Software Eng. SE-1, pp 199-207 (1975)

Brinch Hansen-81 — Brinch Hansen, P.
Edison - a Multiprocessor Language
Software Practice and Experience, Vol 11 pp 325-361 (1981)

Brinch Hansen-83 — Brinch Hansen P.
Programming a Personal Computer
Prentice Hall 1983

CHILL-84 — CHILL Language Definition
Draft Recommendation Z.200
CCITT, 1984

Conradi-76 — Conradi, R.
Further critical comments on Pascal,
particularly as a Systems Programming Language
Sigplan Notices, Nov 1976, pp 8-25

Conradi-85 — Mechanisms and Tools for Separate Compilation
Presented at the IFIP WG2.4 meeting in Maine 7..11 Oct 1985

Cox-86 — Cox, B.J.
Object-Oriented Programming
Addison Wesley 1986

Creak-87 — Creak, G. A.
When GOTO goes to, how does it get there?
Sigplan Notices, Vol 22, No 2, pp 36-42

Dijkstra-68 — Dijkstra, E.W.
Goto Statement Considered Harmful
CACM, Vol 11 No 3 pp 147-148 (1968)

Eersten-80 — Eersten, W.J. v.d.
The Portable Language Midial,
Implementation on the Unix Time sharing System
Delft University of Technology

Eijk-76a — Eijk, P.J.C. van der
Harvalias Reference Manual (sept 1975)
INFO-1976-1 Delft University of Technology, Comp. Sc. Dept.

Eijk-76-b — Eijk, P.J.C. van der
Een HARVALIAS compiler in HARVALIAS (febr 1976)
INFO-1976-3 Delft University of Technology, Comp. Sc. Dept.

Fairley-85 — Fairley, R.
Software Engineering Concepts
McGraw-Hill 1985

FORTRAN — American National Standard Programming Language Fortran
X3.9
ANSI, New York

Gough-86 — Gough, K.J.
Why the Plain Vanilla Linkers?
The MODUS-Quarterly, Issue # 6, Nov 86

Huijsman-87 — Huijsman, R.D., Katwijk, J. van, Toetenel, W.J.
Performance Aspects of Ada Tasking in Embedded Systems
in: Microprocessing and Microprogramming 21 pp 301-310
North Holland 1987

Jensen-75 — Jensen, K., Wirth, N.
Pascal: User Manual and Report
2nd Edition, Springer Verlag 1975

Katwijk-87 — Katwijk, J.v.
The Ada — compiler,
on the design and implemetation of an Ada compiler
Doctoral Thesis, Delft University of Technology 1987

Kernighan-76 — Kernighan, B.W., Ritchie, D.H.
The C Programming Language
Prentice Hall, 1976

Loon-82 — Loon, C.A.v.
The Portable Language Midial
Completion of the implementation on the Unix tss.
Delft University of Technology 1982

LRM — Reference Manual for the Ada Programming Language
ANSI/MIL-STD-1815A-1983
United States Department of Defense

Meyer-88 — Meyer, B.
Object-Oriented Software Construction
Prentice Hall 1988

Occam-84 — Occam Programming Manual
Inmos Ltd. 1984

Ooyen-88 — Ooyen, W.O.v.
WATS, Een Ada Tasking Supervisor en een model voor Ada Tasking op een multiprocessor
Delft University of Technology, 1988

Pascoe-86 — Pascoe G.A.
Elements of Object Oriented Programming
Byte 1986, pp 139-145

PL/1 — PL/1 Language Specifications
Form Y33-6003
IBM, 1969

Poel-73 — Poel, W.L.van der, Maarssen, L.A.
Machine Oriented Higher Level Languages
North Holland/Elsevier 1973

Poel-78 — Poel, W.L. van der
Reference Manual for the language MIDIAL
Delft University of Technology 1978

Ridder — Ridder, Th.F. de
Yet Another Implementation of Coroutines for C
EUUGN, Vol 5 No 2

Rubin-87 — Rubin, F.
"Goto Considered Harmful" Considered Harmful
CACM, Vol 30 No 3 (March 87) pp 195-196

Schiet-84 — Schiet, J.J.C.
The Portable Language Midial
Delft University of Technology 1984

Stroustrup-83 — Stroustrup, B.
Adding Classes to the C-language:
an Exercise in Language Evolution
Software Practice and Experience, Vol 13 pp 139-161 (1983)

Stroustrup-88 — Stroustrup, B.
What is Object-Oriented Programming?
IEEE-Software, May 1988 pp 10-20

Toetenel-88 — Toetenel, W.J., Katwijk, J. van
Asynchronous transfer of control in Ada
To appear in Ada-Letters, Oct 1988

Wirth-76 Wirth, N.
Modula: a Language for Modular Multiprogramming
Software Practice and Experience, Vol 7, pp 3-35 (1977)

Wirth-83 Wirth, N.
Programming in Modula-2 (second and third printing)
Springer Verlag 1983/1986

Wirth-88 Wirth, N.
Type extensions
Transactions on Programming Languages, Vol 10, No 2 pp 204-214

Wijngaarden-76 Wijngaarden, A. van, et al
Revised Report on the Algorithmic Language Algol 68
Springer Verlag 1976

Wulf-73 Wulf, W.A.
Issues in higher-level machine-oriented languages
in [Poel-73]

The Control Data Cyber ALGOL 68 Garbage Collector

J. Schlichting

Control Data Corporation

1. Introduction

The Control Data Cyber compiler was implemented by Control Data Nederland BV in the years 1972-1975 under a contract with the Dutch universities of Utrecht and Groningen and SARA, a joint venture of the Mathematical Centre and the two Amsterdam Universities with W. van der Poel as external consultant and the author as project leader.

At the start of the project the language definition was still being developed by IFIP WG 2.1 on Algorithmic Languages. The aim of the project was to conform as closely as possible to the final language specification. Eventually, the language definition and the compiler were completed nearly at the same time.

The second section first briefly explains the need for a garbage collector for the programming language ALGOL 68 and then discusses some aspects of the object program design that influence the workload of the garbage collector and hence its performance requirements.

The third section describes some hardware characteristics of the Cyber computer system and the internal representation of data for ALGOL 68 pertinent to the garbage collector design.

The fourth section finally discusses the garbage collector design and implementation in some detail.

2. The garbage collection problem

ALGOL 68 is a block-structured language.
As usual, local or automatic data is allocated on a run-time stack, containing at any point in time one stack-segment for each active block; the contiguous allocation of those stack-segments in a single memory-segment in the order of block-nesting minimizes the memory required for the storage of local data.

Global data i.e. data with unlimited life-time or scope, is allocated in ALGOL 68 by a so-called global generator that yields a pointer to the newly allocated data; the data-item is said to be the object of the pointer; the area in memory containing the dynamically allocated data is conventionally called a heap.
In ALGOL 68, the object of a pointer may be local as well as global. Generally, global data in a program are not necessarily required until the program terminates; in order to reuse the memory allocated to global data, a mechanism for deallocation is needed. The ALGOL 68 language has no explicit deallocation statement, rather the object of a pointer remains allocated as long as the pointer continues to exist; the process that retrieves the

memory allocated to objects to which no pointer exists, is called a garbage collection.

If a pointer to a local object allocated on the stack is assigned to a pointer or variable with a longer life-time than the pointer's object, then upon expiration of the object's life-time the pointer continues to exist but its object does not.
This phenomenon is called the dangling pointer problem.
The ALGOL 68 scope-rules prohibit such assignments by demanding that in an assignment the scope of the source must include the scope of the target or in other words that the life-time of the source must at least be equal to the life-time of the target.
For some assignments the scopes of source and target cannot be determined at compile time; rather than to incorporate a run-time scope check in the object code, the Cyber ALGOL 68 compiler changes the scope of the source to global and allocates it on the heap.

This approach results in a simplification of the compiler and the generated object code at the expense of more data being allocated on the heap; it is not a deviation from the ALGOL 68 standard but rather a super-language feature, since it assigns a sensible meaning to an action the result of which is called undefined by the standard.

Arrays in ALGOL 68 consist of two parts: the bound information and the elements that may require to be allocated separately. For simplicity of stack handling all allocation in the run-time stack is done at compile time; the size of the stack-segment for every block is fixed by the compiler; therefore, the elements of a local array are not allocated on the run-time stack but on the heap. Here again a simplification of the object code was achieved at the cost of an increase of heap related activity. The bound information is combined with a reference to the elements of the array into a new kind of pointer called an array-descriptor.

ALGOL 68 includes operations on pointers that yield a pointer to part of the data referenced by the pointer operated upon. The objects of two pointers may overlap without one object being contained in the other. Because any type of data may be dynamically allocated, the object of a pointer may contain one or more pointers.

The task of the garbage collector is to make the memory vacated by data no longer reachable through existent pointers, available for allocation of data.

Because this process in the general case frees a number of disjoint memory segments, the garbage collector must be able to move the data to be retained into a contiguous memory segment and to adjust all pointers referencing the moved items accordingly.

The treatment of the scope rule and local arrays tends to increase the amount of data on the heap and hence of garbage collector activity.
In summary, the Cyber ALGOL 68 garbage collector must be capable to handle pointers to any size objects with no restrictions on overlapping of objects, to perform data compaction and pointer adjustment and execute very efficiently.

3. Control Data Cyber hardware and internal representation of data

In the Control Data Cyber computer system in use at the time of the ALGOL 68 project memory consists of a single one dimensional array of consecutively numbered 60-bit

words. Addresses are 18 bits wide; the memory available to a user program is limited to 2**17 or 131072 words; all addresses used in a program are positive 18 bit integers.

The lower part of memory contains the ALGOL 68 object program code as well as the necessary run-time routines and possibly the code and local data of procedures written in other languages. The run-time stack is allocated directly above the program; during the execution of the program the stack will grow and shrink as program blocks are activated and deactivated. The heap is a contiguous segment of memory allocated at the top of available memory and growing downwards as objects are allocated on the heap. The memory segment between the top of the stack and the bottom of the heap is used to allocate stack segments and new heap objects; when this segment is exhausted, the garbage collector is activated to compact the accessible data in the heap upwards and thus create allocatable space between stack and heap. In case not enough memory space can be freed by the garbage collector, an operating system request is issued to increase the size of available memory.

Memory allocation is done in integral 60-bits words exclusively. In the Cyber computer integer and floating point numbers are represented in ones complement mode in full 60-bit words. In the Cyber ALGOL 68 system integer values are restricted to the range (1-2**48,2**48-1) which implies that the 12 most significant bits are all identical viz. 1 for negative and 0 for positive numbers. A floating point number may assume the value

mantissa * 2**exponent,

where mantissa is an integer value in the range (1-2**48, 2**48-1) and exponent is an integer value in the range (1-2**10,2**10-1). The internal representation of a floating point number consists of three fields designated s, e and m:

— s occupies the most significant bit and contains the sign of the mantissa;

— e occupies the second through twelfth bits and represents the exponent. The nature of its representation is not relevant here.

— m occupies the lower 48 bits and contains the lower 48 bits of the mantissa in ones complement representation.

Any floating point number with an exponent of 2**10-1 is considered out of range or infinite; special hardware instructions exist to test floating point numbers for an out of range condition. Any hardware floating point operation yielding an out of range result produces a result with exponent 2**10-1 and a m field of all zeroes. All other simple plain values, i.e. values of type BOOL, CHAR, BITS or BYTES are represented by a full 60-bit word that is not out of range, are all zeroes.

Generally values that do not contain pointers are termed plain values. Thus bit patterns that represent an out of range floating point value and show a non-zero mantissa are not used to represent plain values. Such bit patterns are used exclusively for the representation of pointers.

Pointers in ALGOL 68 are of two kinds: references and array descriptors. A reference is a value of which the mode (the ALGOL 68 term for type) begins with a ref-symbol; a reference occupies one 60-bit word; an array descriptor arises from an array, i.e. a value of which the mode begins with row, and contains the address of the array elements and the bound information; an array descriptor occupies 2*n+2 60-bit words where n represents the number of dimensions of the array.

Bit patterns with the upper 12 bits equal to 011111111111 or 100000000000 and the lower 48 bits not all equal to zero, represent the first word of an array descriptor and a reference respectively. In the sequel we will, for reasons of brevity, discuss references only.

The internal representation of a reference consists of a 60-bit word with the following fields:

NAME	SIZE		VALUE
sign	1	bit	1
flag	11	bits	represents an exponent of 2**10-1
size	12	bits	the size in words of the object -1
trace	1	bit	1 if object contains pointers, 0 if object is plain
spec	17	bits	reserved for other purposes
addr	18	bits	address of first word of object

The sign and flag fields characterize a word as a reference, the addr and size fields determine the object of the pointer and the trace field distinguishes a pointer to a plain object from a pointer to a non-plain object. This distinction obviates the need to inspect all words of a plain object in the tracing phase of the garbage collector described below and it allows the allocation in the heap of objects such as input/output buffers, for which dynamic allocation is required, that are not directly manipulated by the program and that may contain any conceivable bit patterns.

4. Garbage collector

The garbage collector problem logically consists of three tasks: tracing, pointer adjustment and data compaction. The tracing task comprises locating all existing pointers and creating a data structure facilitating access to the pointers found. Because in the Cyber ALGOL 68 system pointers contain not only the address but also the size of their objects, this data structure contains all information required for the pointer adjustment and data compaction tasks.

Because data is upward compacted, these tasks can be implemented efficiently only if the pointers can be accessed in the order of descending last word address of their object; therefore, an additional phase called sorting that reorganizes said data structure, is inserted after the tracing phase. The third phase then, in a single scan through the pointers finds all objects to be retained in the order of their original last word address, reallocates them contiguously from the top of memory downward, preserving the existing

overlapping between them, adjusts the pointers accordingly and creates a chain of unused memory segments of holes.

The fourth and last phase in a single scan of this chain of holes performs the data compaction, moving all data residing between the holes upward into one compact memory segment.

TRACING

An object is accessible when it resides in the stack-segment of an active block in the program or when it is the object of an accessible pointer. Since objects may contain pointers, the set of accessible objects is recursively defined.

Conceptually the tracing phase performs as follows:
Three chains of pointers are maintained: a ready-chain containing pointers that have been traced and of which all pointers in its object have been traced and two follow-chains of pointers that have been traced and two follow-chains of pointers that have been traced and of which the object may contain pointers that have not been traced. The tracing phase first scans the run-time stack and chains all pointers to plain objects in the ready-chain and all pointers to objects containing pointers in the first follow-chain, marking all pointers encountered as traced. Then the first follow-chain is scanned and for each pointer in the chain, the pointers in its object are chained: pointers to plain objects in the ready-chain and pointers to non-plain objects in the second follow-chain; again all pointers encountered are marked as traced; after this scan the first follow-chain is appended to the ready-chain; if the second follow-chain is empty, the tracing is completed, otherwise the roles of the first and second chains are reserved and this step is repeated.

The actual implementation is a refinement of the above scheme. Pointers are not moved in the tracing phase; the only fields ever changed are the address and flag fields. The address field will be used as a link in the chain into which the pointer is chained. A chain of pointers requires a single header word to hold the address at the first pointer in the chain; new entries are always appended at the head of the chain. Eventually pointers must be chained in the order of the last word of their object; the 17-bit last word address of the pointers object is computed and divided in two parts; the 9 most significant bits denote the 256-word block containing the address and the 8 least significant bits denote the offset of the address in the block. The block-number is stored in the flag field of the pointer, thereby marking the pointer as traced. For each possible value of the offset, 0 through 511, a separate ready-chain and a separate follow-chain is used; the pointer is linked in the chain of the appropriate type corresponding to the value of the offset-part of its objects last word address. This technique not only allows to chain pointers in situ, it also partially orders the pointers for later processing.

The tracing phase is divided in two steps:

The first step scans the run-time stack and links every pointer it finds into the ready-or follow-chain designated by the offset-part of the address of the last word of its object; a pointer to a plain object is chained in a ready-chain and a pointer to a non-plain object in a

follow-chain.

The second step is an iterative process; in each iteration all follow-chains are scanned; the process is terminated when all follow-chains are empty. The scan of a follow-chain first copies the head of the chain in a local variable and clears the head of the chain ; then starting with the pointer addressed by the local variable, all pointers in the chain are processed and the chain headed by the local variable is appended in front of the corresponding ready-chain; i.e. the ready-chain for same relative address. For each pointer processed the last word address of its object is reconstructed from the block-number currently in its flag-field and the offset particular to the follow-chain in process; the object of the pointer is scanned; unmarked pointers to plain and non-plain objects are recognized and chained in the ready- and follow-chains corresponding to their offset respectively. When the iterative process is terminated, all accessible pointers have been chained in the ready-chain corresponding to the offset of the last word of their object.

SORTING

For each block in memory a new-chain is established.
The ready-chains created by the tracing phase are scanned in the order of increasing offset; every pointer in each chain is rechained in the new-chain corresponding to the block containing the last word of their object. Since pointers are always inserted at the head of a chain and rechaining is done in order of increasing offset, the pointers in each new-chain are linked in order of decreasing offset; all accessible pointers may thus be accessed in decreasing order of address of the last word of their objects by scanning the new-chains in decreasing order of block-number. The sorting phase thus amounts to the second pass of a two-column bucket-sort well known from the hollerith card-sorters.

POINTER ADJUSTMENT

The compacting phase will move all accessible objects in the heap into a contiguous area of memory extending downward from the top of available memory; the ordering of objects in memory will not be changed; the move will proceed from the highest to the lowest source-address; the first word move will be placed in the highest address available to the program. Thus every word will be moved upward in memory over an offset equal to the number of words found unused between top of memory and its original address.

The pointer adjustment phase performs a single scan of all accessible pointers in order of decreasing last word address of of their object; this scan is split in two parts; the first part processes pointers to objects in the heap, the second part processes pointers to objects in or below the run-time stack. The first part of the scan maintains the current offset and lowest address of accessible objects located so far as well as a list of segments of unused words or holes; for each pointer scanned it checks for unused words between the last word of the object and current lowest address; if such a hole is found, it is appended to the list of holes and the current offset is adjusted; the pointer is reconstructed in its original form and the current offset is added to its address field. The second part of the scan only

restructures the pointers into their original form.

COMPACTING

This phase uses the information in the list of holes to move all data between the holes upward into a compact area extending downward from the top of available memory.

The memory requirements of the garbage collector are very modest:

Headers of chains of pointers:

ready-chains	256
follow-and new-chains:max(256,512) =	512
local variables	24
code	308
total	1100 words

-- . --

De Theorie van de Administratieve Automatisering.

E.J. Nijenhuis
PTT

Van der Poel was één van de eersten in Nederland die zich bezig hield met administratieve automatisering. Omstreeks 1960 was op het Dr. Neher Laboratorium een salarisadministratie operationeel, op de ZEBRA. Daarna heeft hij dit terrein verlaten om er - zo ver mij bekend - nooit meer terug te keren.

Rond deze tijd splitsten de computergebruikers zich in twee kampen: de administratieve automatiseerders en de rest. Beide werelden bestonden naast elkaar, nagenoeg zonder contact met elkaar te hebben en zonder elkaar te begrijpen.
Het is misschien beter niet op de oorzaken in te gaan.

Om een vergelijkingsmaatstaf te hebben, kunnen we de theorie van de administratieve automatisering misschien het beste vergelijken met die van andere terreinen van het vakgebied: de kerninformatica en de toepassingen op het gebied van wetenschap en techniek.
De kerninformatica, bouw van operating systemen, compilers en dergelijke toepassingsonafhankelijke zaken, heeft een goede theoretische basis. Dit komt waarschijnlijk doordat de computers pas zijn ontstaan toen er al een goede theorie was.

Het begon eigenlijk al in 1900, toen Hilbert de vraag aan de orde stelde of elk wiskundig probleem in principe oplosbaar was, of elke functie berekend kon worden. Dit was aanleiding naast de traditionele wiskundige objecten als getallen, verzamelingen, functies etc. ook het rekenproces als onderwerp van wiskundige studie aan te vatten.
Dit is op uiteenlopende manieren aangepakt:

Church ontwikkelde de lambda calculus, de spelregels voor het invullen van parameters in formules.

Schönfinkel bestudeerde de combinatoren, wat oneerbiedig geformuleerd:
een soort knopenspel dat u kunt spelen met een rij gekleurde knopen. Men pakt steeds de voorste knoop weg, de kleur van deze knoop bepaalt hoe de andere knopen verschoven moeten worden.

Turing maakte de Turing machine.
Hij maakte een wiskundig model van een rekenaar: de kennis van een rekenaar - een hoeveelheid rekenregels en formules - werd gereduceerd tot een automaat. Verder beschikt de rekenaar over een onbeperkte hoeveelheid ruitjespapier; dit werd voorgesteld door een tape. De automaat kan deze tape lezen en beschrijven. Deze eenvoudige constructie bleek voldoende om precies al die functies te berekenen waar ook Church en Schönfinkel via hun benadering op waren uitgekomen.

De Turing machine kon met de bestaande technologie zonder grote problemen gemaakt worden, maar daarvoor was hij niet bedoeld. Het was een ideaal gereedschap voor mathematici om de mogelijkheden te bestuderen.

Eén van de ontdekkingen was de universele Turing machine, die allerlei andere machines kon nabootsen. Hiermee was aangetoond dat dezelfde machine voor de meest uiteenlopende zaken gebruikt kon worden en dat compilers etc. in principe realiseerbaar zijn.

Toen gedurende de oorlog Von Neumann en anderen machines gingen ontwerpen voor practisch gebruik waren de mogelijkheden en de principiële beperkingen al goed bekend. Men kon er op vertrouwen dat de ingeslagen weg niet dood zou lopen. Hierdoor kon b.v. de ENIAC in relatief korte tijd ontworpen en gebouwd worden.

Technisch/wetenschappelijk rekenwerk heeft altijd een goede theoretische basis gehad. De computers veranderden hier niets aan. Er konden alleen berekeningen worden aangevat die vroeger door hun omvang niet uitvoerbaar waren.

De programmering vereiste al spoedig veel tijd van de schaarse specialisten. Deze tijd werd voornamelijk besteed aan zaken die meer te maken hadden met de computer dan met de aard van het probleem. Men had behoefte aan beter gereedschap. Rond 1960 werd dit gevonden in de vorm van programmeertalen als FORTRAN en ALGOL. Door de kennis van de theoretische mogelijkheden enerzijds en de behoeften anderzijds was het niet al te moelijk te bepalen wat deze talen moesten kunnen, wat niet zeggen wil dat er geen problemen waren.

Voor niet numerieke problemen werd de programmeertaal LISP bedacht door McCarthy. De grote rol die LISP kon spelen bij artificial intelligence, expert systemen etc. houdt duidelijk verband met de afstamming van een eerbiedwaardige voorvader: de lambda calculus van Church.

De combinatoren krijgen de laatste jaren steeds meer aandacht. Ze worden tegenwoordig zelfs practisch gebruikt bij de implementatie van functionele talen.

De administratieve automatisering heeft zich geheel anders ontwikkeld. Het prille begin: de ponskaartmachines van Hollerith werden het eerst gebruikt bij de Amerikaanse volkstelling van 1890. De machine kon selecteren en tellen, beide goed gedefinieerde begrippen. Misschien was dit eerste geautomatiseerde administratieve proces ook wel één van de weinige waarbij een goed fundamenteel begrip aanwezig was.

In de eerste helft van onze eeuw ontsproten de boekhoudmachines uit een huwelijk van telmachines en schrijfmachines. Het waren electromechanische apparaten met telwieltjes waarvan de stand afgedrukt kon worden op papier. Het werk van een boekhouder werd vereenvoudigd, doordat de machine kon optellen en zo nodig de stand van een telwerk kon overbrengen naar een ander telwerk. Voor steeds herhaalde handelingen was het prettig dat ze konden worden geprogrammeerd met functielinialen. Rond '58 waren de boekhoudmachines uitgegroeid tot gecompliceerde apparaten, gekoppeld aan ponskaartapparatuur. Deze machines waren op grote schaal in gebruik.

Intussen waren er pioniers die meenden dat de nieuwe computers ook in de administratieve wereld een belangrijke rol konden spelen. In het algemeen ging het hier om relatief eenvoudig rekenwerk dat vaak herhaald werd. De programmering was natuurlijk ook hierbij weer een probleem. Helaas was er nu geen theorie die aangaf wat de machine moest kunnen voor administratieve processen. Een groep IBM medewerkers heeft het probleem toen geniaal opgelost: men definieerde een standaard voor een boekhoudmachine en men ontwierp een emulator voor deze standaard. Het resultaat werd aan de wereld aangeboden als de programmeertaal COBOL. De ponskaarten waren slechts physiek aanwezig voor de in- en uitvoer, voor de interne verwerking werden ze in de machine nagebootst. De gebruikers waren zeer verheugd, niet alleen wegens de snelheid en de papierbesparing maar ook omdat de intern gesimuleerde ponskaarten niet meer beperkt waren tot 80 kolommen.

De conversie van bestaande processen van de boekhoudmachines naar computers was relatief eenvoudig: het was niet nodig om geheel nieuwe methoden te ontwikkelen en aan een nieuwe begrippenwereld te wennen. De bestaande methoden bleven bruikbaar.

Sedertdien zijn er wel wat "toeters en bellen" vanuit andere programmeertalen in latere versies van COBOL doorgedrongen, maar de basis is dezelfde gebleven; compatibiliteit staat hoog in het vaandel. De denkwereld van de administratieve programmeur verschilt nog steeds weinig van de boekhoudmachines van '58.

Met de komst van de massageheugens ontstonden de database-systemen, bedoeld om verschillende programma's en verschillende gebruikers gelijktijdig toegang te geven tot de gegevens. De toepassingsonafhankelijke problemen, zoals het voorkomen van conflicten en deadlock, werden behoorlijk opgelost; dit lag op het terrein van de kerninformatici.

De wijze waarop de gegevens opgeslagen en gehanteerd moesten worden was een veel groter probleem. Naast programmeurs kwamen er systeemanalisten en functionele ontwerpers. Data-modellering werd een modewoord.

Allerlei ontwerpmethoden volgden elkaar op. Het waren recepten om een probleem om te vormen tot het past in de infrastructuur van COBOL en de bestaande database-systemen. Geen van die ontwerpmethoden werkte geheel bevredigend. Ze konden ook niet de vergelijking doorstaan met ontwerpmethoden uit de techniek.

Steeds meer begon men de behoefte te voelen aan een theorie. Men keek jaloers naar de vorderingen op andere vakgebieden waar de technieken en de methoden een theoretische basis hadden. Het werd ook gewoonte wiskundige bewoordingen te gebruiken, zij het vaak met een afwijkende betekenis. (Een Codasyl SET lijkt meer op een functie dan op een verzameling, zelfs gelijkheid is niet altijd transitief en symmetrisch).

In 1970 publiceerde Codd zijn relationele data-model. Het eerste gegevensmodel dat was gebaseerd op wiskunde. Dit model werd in de administratieve wereld met gejuich ontvangen. In vrij korte tijd waren alleen nog database-systemen verkoopbaar die pretendeerden relationeel te zijn. De acceptatie van het model berustte echter meer op de hoop dat er eindelijk een theorie geboden zou worden dan op de werkelijke merites van

het model.

Zoals Turing een model had gemaakt van een rekenaar, zo maakte Codd een model van een boekhouder. In plaats van het ruitjespapier van de rekenaar kwam het tabellenpapier van de boekhouder. De tabellen komen min of meer overeen met het wiskundige begrip relatie. Het werk van een boekhouder kon nu uitgedrukt worden in algebraïsche bewerkingen op de tabellen.

In plaats van een theorie die inzicht geeft in het wezen van een administratie was het een formalisering geworden van de klassieke papieren boekhouding. Het relationele model zou misschien een goed gereedschap zijn - analoog aan de Turing machine - om de mogelijkheden en beperkingen van een traditionele boekhouding te analyseren. Dit was echter niet de bedoeling. Het wordt gebruikt voor practische systemen en daarvoor is de benadering nauwelijks beter dan een Turing machine.

Niet alleen ontbreekt een theorie - het model geeft geen inzicht in de architectuur van een administratief proces maar draagt alleen bouwstenen aan - het model mist ook eigenschappen die nodig zijn als gereedschap bij de realisering.

Het is in een administratie vaak niet te voorkomen dat dezelfde regel twee maal of vaker in een tabel voorkomt. Die regel kan echter niet twee maal element zijn van een wiskundige relatie. In practische "relationele" database-systemen doet men daarom water bij de relationele wijn door dit wel mogelijk te maken.

Misschien nog ernstiger is het geringe aantal gegevenstypen: (de afstamming loochent zich niet) slechts getallen en rijtjes letters. Op nagenoeg elk ander terrein worden talen zonder pointers of soortgelijke constructies als onbruikbaar beschouwd.

Het is gebruikelijk dat we verbanden kunnen leggen tussen objecten (in de machine dus een pointer naar de corresponderende gegevensstructuur) zonder de omweg via één of meer eigenschappen (zoals een naam).

In het relationele model kunnen we alleen naar een object verwijzen in termen van de attributen. Hierdoor ontstaat de verplichting te zorgen dat elke relatie is voorzien van een groepje van één of meer attributen dat uniek is voor een object. Om de zaak in de hand te houden is er een gecompliceerd stel spelregels nodig. Een onoverzichtelijk systeem van normaalvormen dat niet eens uitgedrukt kan worden binnen het relationele formalisme.

Dit is zo ingewikkeld doordat de informatie op een ongeschikte manier wordt gerepresenteerd. Het doet denken aan het uitvoeren van een deling met romeinse cijfers: de complicaties hangen samen met de notatie en niet met het wezen van de deling.

Codd had overigens weinig keus, de administratieve wereld is aan COBOL gewend en in COBOL bestaan alleen letter- en cijferrijtjes.

Tot nu toe was het een wat droevig verhaal. Ik zal proberen in majeur te eindigen. Meer nog dan het voorafgaande is het volgende een persoonlijke en subjectieve indruk.

Het lijkt er op dat de kloof tussen de administratieve wereld en de meer exact georiënteerde wereld begint op te vullen. Ervaringen met programmageneratoren maken

duidelijk dat er andere manieren zijn om een computer aan te spreken. Het woord exact wordt vaker gebruikt dan vroeger. De beschouwingen over data-modellering, ontwerpmethoden, etc. worden weliswaar gehinderd door de representatie van de gegevens, maar inhoudelijk is er steeds meer dat - althans bij mij - de indruk wekt een stap in de goede richting te zijn. Het zijn misschien condensatiekernen waaruit een samenhangende theorie kan groeien.

Ik krijg steeds meer de indruk dat enerzijds de administratieve wereld de steun nodig heeft van de meer exact georiënteerde academische wereld om tot een consistent beeld te komen van het wezen van administratie, en dat anderzijds het administratieve gebeuren (en niet alleen hulpmiddelen zoals databases) een boeiend onderwerp van studie kan zijn voor fundamenteel onderzoek op universitair niveau.

Memories of forty years with Computers

Professor R J Ord-Smith BSc PhD FIMA FBCS

University of Bradford

Forsake not an old friend
for the new is not comparable to him;
a new friend is as a new wine;
when it is old thou shalt drink it with pleasure.

The Bible: Ecclesiasticus Chapter 10

It is a very great pleasure to be in Delft as part of a valediction for Wim van der Poel who I regard amongst my most precious old friends. We have been colleagues for 33 years, separated too much by busy lives and from living in two countries. During the years 1955-1962, however, we worked closely together on the commercial development of the Zebra computer, he as its creator and I as a member of the Information Processing Division of Standard Telephone and Cables in Britain who undertook its manufacture. But that is out of chronology and I shall come back to it.

I have chosen for my theme, and by way of a shameful indulgence in looking with nostalgia at my own career, the remarkable good fortune I and my contemporaries, including Wim, have had in the timing of our sojourn on this planet. Again and again, and without careful planning, events have synchronised my life with involvement with computers throughout the period of their development.

Though both our early lives were influenced by the last world war neither Wim nor I were quite old enough to be involved as combatants. Yet my conscription into the Royal Air Force only months after the cessation of hostilities *imposed* on me one of the best introductions to the technology of the early computers possible with a splendid year-long Radar Mechanics course. This technology was still secret and we were allowed the same compromise as was given to the team who had developed the COLOSSUS and other code cracking computers in the most remarkably successful secrecy only a couple of years previously: we were allowed to keep note books under threat of the firing squad if their contents were disclosed. I still have mine; I doubt if I would be shot for opening them now! What a lot of what has become everyday technology was there: all of the basic binary logic of computers; the quartz clocks of our wristwatches; the magnetrons of our microwave ovens. My small claim to fame in those days was to hit on the idea, subsequently implemented, of mounting an H_2S radar scanner, supposed to go underneath a Lancaster bomber, on top of a water tower on our camp in the middle of Salisbury Plain reproducing recognisable features of the surrounding topography.

Forty years on I was priviledged to be invited to a day at the Institute of Electrical Engineers in London devoted to a gathering of an amazing number of those fine men who had worked on COLOSSUS and the decyphering of the ENIGMA and FISH codes. I could be tempted to spend all my time repeating their wonderful anecdotes. What was

most fascinating was the primitive nature of their technology by today's standards yet the advanced techniques they achieved. Using 5 hole paper tape, like the first Zebras still did, they were reading it photoelectrically at 50 ft/sec, some 5000-6000 characters/sec. COLOSSUS subsequently greatly extended these processing speeds by becoming the first stored program 5 fold parallel processor. For me the real fascination was to see the same valve technology, even the same V numbered valves as I had used in the RAF. At the end of the war many of the COLOSSUS team dispersed to Manchester, Cambridge and the National Physical Laboratory to develop the Mark I, EDSAC and Pilot ACE computers noted later in van der Poel's doctoral studies in which Zebra was conceived.

My own financial fortunes boomed when I completed my national service in 1948 and four shillings a day became £4 a week with an ex-serviceman's grant to read mathematics at what is now Southhampton University and was then a London college. Having paid my keep, I saved enough to buy a motorcycle.

Following graduation, I continued at Southampton with postgraduate doctoral studies aided by a grant from the Department for Scientific and Industrial Research now the Science and Engineering Research Council. My studies involved calculating the contribution to the binding of energy of light atomic nuclei arising from the interaction of different states. Working in particular with Lithium 7 I faced the need to calculate the eigen values and eigen vectors of energy matrices greater than 40 x 40 which became impracticable by hand calculation. Fortunately, DSIR owned the National Physical Laboratory and the ACE computer and I was able to use this, being one of a small handful of computers available. My programs were among the first to use a new magnetic drum store on ACE. They took an hour or two, would take a minute or so on a modern micro and a fraction of a second on a transputer.

I faced an important choice at the end of my research days: I obtained a Civil Service Commission to continue Nuclear Research at the Cavendish Laboratory at Cambridge under Professor Sir Neville Mott; I was also offered a job with Standard Telephones and Cables in computer developments. Two issues helped me decide on the latter. My work on Lithium was coincidentally temporarily classified at a time when Lithium was a possible importance in nuclear fusion studies; that irked me. The Commission offered £600 p.a. and S.T.C. £840 p.a. I would like to believe that the first reason was the more important.

In 1955 S.T.C. had already been involved in an unsuccessful collaboration with Lyons Teashops to develop a commercial computer based on a ten state valve device. Surely decimal processing was more logical than binary for commercial work! Lyons had founded LEO Computers and S.T.C. were working on an interim new development known as STEP1 when I joined them.

At the same time as the events I have described, an important development had taken place in the Netherlands with the establishment in Amsterdam on 11th February 1946 of the Mathematical Centre. Its Computation Department was under the leadership of Professor van Wijngaarden. Of especial interest to us was one of his doctoral research students who had published work on digital computers in the very early 1950s. His thesis

included descriptions of no less than three computers, the first, on which the ideas for the other two were based, being kept up his sleeve till last, one so simple it could only subtract. This was of course Wim van der Poel. The computer called PETRA was constructed and installed by Wim at the Dr Neher Laboratory of the Netherlands PTT at Leidschendam. In his thesis he does not think it worth telling his readers what the name of his computers mean. I will tell you that the name of the other was devised by his wife Annie: Very Elementary Binary Reckoning Automat in Dutch giving the acronym ZEBRA. It was clear that Zebra had real commercial potential and, via Netherland Standard Electric, their sister company S.T.C. undertook to engineer, construct and market it. From 1956 into the early 1960s I spent many happy months in Holland working with Wim during which time we became the firm friends we remain. I am very proud of my friendship with Wim and Annie and their sons Hans and Peter, grateful for their kindness and hospitality at all times and delighted to be part of this valediction.

I am pleased too to have acquired another old friend in those happy days; Dr Gerrit van der Meij whose mastery of every problem, theoretical, technical and physical is an inspiration to us all. Gerrit's work included much of the development of Zebra's software including its Simple Code with full floating point facilities and later, and most innovative for its day, a complete description of an implementation of ALGOL, written in ALGOL.

During the years ZEBRA developed from first implementation in valve technology to transistor versions, was one of the first European computers to offer ferrite core storage and eventually sold more than 50 machines throughout the World: Britain, Holland, Belgium, Germany, France, Switzerland, Portugal, Sweden, Australia, Canada, South Africa. The last ZEBRA, and one of the original valve versions at that, ran till 1981 driving one of the largest flat bed plotters in the world designing sails for the world's most modern racing yachts near Southampton in England. It did not, as we had hoped, end up in the Science Museum with Babbage's Analytical Engine and the precious NPL ACE but they video recorded it in operation on its last day and prior to its dismantling and have that in their archives. There isn't time to dwell on the many novel features which ZEBRA possessed. Its most especial feature was that it did not decode the 15 bits which made up the operation part of its instructions but each of them independently and simultaneously controlled fundamental operations within the machine. That has never been imitated successfully. Perhaps the most remarkable thing about ZEBRA was the love it engendered in its followers. It is still unique I think among the old machines in bringing a look of wonderment to the faces of a still large contingent of old codgers.

Remarkable too was the enormous range of users its customers found for ZEBRA:

one of the first Airline Reservation systems for Air France; transport aircraft weight balancing for Swedish Airlines; bridge design in Portugal and in Britian including the Forth Bridge; in the Netherlands developments of the Delta Estuaries, telephone traffic studies, crystallographic research at Groningen and Utrecht Universities, wind tunnel research at the Royal Netherlands Aeronautical Laboratories, a number of research projects from 1958 at Delft; photogrammetric work in Switzerland; submarine design, animal food production control, tobacco production, borough council administration,

research at three universities and two polytechnics in Britain; NAAFI in Germany; oil production in Canada; canning fruit in Australia;

are a sample of its variety.

In 1962 I moved to the University at Bradford, following a Zebra I had helped to install there. Though Zebra has gone, they don't get rid of me till September 1990. Many links with Zebra continue. Prof van der Poel has been external examiner for one of my Ph.D. students (now my Deputy Director Dr Stanley Houghton) and my University has honoured Wim with a Doctor of Technology. I mentioned Zebra having had ferrite core storage in its later models. This used the expertise of Peter Comerford who joined S.T.C. having had done early research on ferrite cores in the University of Wales. He now has a Bradford doctorate and has for a number of years been a member of our Postgraduate School of Studies in Computer Science and designer of digital church and theatre organs in manufacture and sale in Britain and Europe. John Shapland was also a member of the S.T.C. team and nicely joins together several of the strands in my own career. He too had worked with the RAF radar. Being a little older, he saw war service and taught at the RAF Officer Training College at Cranwell. He has been awarded Masters degree by Bradford for work in multi state logic and, though in semi-retirement, continues in active research having found a new use for the old ferrite cores. They possess cavity resonances, giving very high voltage very short time duration pulses finding important new applications. The old original cores have the best properties and no one knows how they were manufactured!

Whilst we use special occasions to look backwards, our daily task is to look forwards. My own Computer Centre of which I am Director at Bradford has a central service installation with some 800 connected campus devices, connections to the British Academic Network JANET, the European Advanced Research Network EARN and hence connections worldwide. Delft became part of this complex on 16th November 1985. The CYBER installation at Bradford has a third of a million times the storage capacity of Zebra. We have just received £700,000 from the University Grants Committee for a pilot scheme to make Bradford one of the first Universities offering campus network connection to every member of staff and student.

With ever increasing power in the hands of users of local workstations, it is clear that the main developments are in providing ever faster broad band networks giving access to data bases and other large repositories of information including pictures and colour. This is putting Computer Centres and Libraries on convergent courses. Already two of our Universities have chosen to put their Computer Centres under the direction of their Librarian. In the United States this is happening too and sometimes by the shock tactic of a fait accompli.

I try to find time to take personal interest in some of the modern developments. I admit personal satisfaction in the high speed colour graphics allowing developments of the work of Newton in the 17th, Cayley in the 19th, Julia and Fatou in the 20th centuries in mathematical iteration, to expose themselves in the sheer beauty of the symmetries of the Mandelbrot Set and related studies. We are currently studying proposals to install

transputers at Bradford capable of carrying out matrix calculations in under a second which I idly calculate would have taken Zebra 214 days if it had 1000 times the storage.

Enough! I end with another quotation from Ecclesiasticus, Chapter 24.

> The wisdom of a learned man cometh by opportunity
> of leisure;
> and he that hath little business shall become wise.

This quotation comes from the Apocrypha and the example of Wim van der Poel makes it apocryphal. I have never known him to have little business. Let us all hope for him that he may now find the time for leisure and sometimes for little business that he polish his wisdom yet further.

Juggling with combinators

Henk Barendregt
Computer Science Department
Catholic University
Nijmegen, The Netherlands

At this special day in honour of professor van der Poel it is appropriate to speak about one of his scientific interests: combinators. I will try to do this in the style of one of his more private interests, namely that of magic. By the way, nothing is mysterious about magic. By making appropriate use of the phenomena one can obtain unexpected results. This can be done with cards but, as we will see, also with combinators.

Nothing in this paper is new. Emphasis is on unexpected properties of combinators. If proofs are deleted, then they can be found in Barendregt [1984].

1. Rules of the game

The rules are simple. There are certain objects called *combinators*. These are built up from two basic combinators, **K** and **S** using a binary operation called *application*. If A and B are combinators, then so is AB, the result of *applying* A *to* B. Intuitively we can think of A as a function and of B as the argument. There are no other combinators than those built up from **K** and **S** using application.

Application will not be associative: in general $(AB)C \neq A(BC)$. We will use the convenient convention of association to the left: ABC denotes (AB)C and ABCD denotes ((AB)C)D; etcetera.

Now that we have the combinators and some notation for them, we can state the basic axioms that they satisfy.

1.1 Axioms of combinatory logic. For all combinators A, B and C one has

(1) $\mathbf{K}AB = A$;

(2) $\mathbf{S}ABC = AC(BC)$;

(3) $\mathbf{K} \neq \mathbf{S}$.

The theory axiomatised by these axioms is called combinatory logic, notation CL. It is known that this theory is consistent.

1.2 Theorem (Curry). Combinatory logic is consistent.

What is more, is that the theory is highly undecidable.

1.3 Theorem (Grzegorczyk). The equational theory combinatory logic is essentially

undecidable.

This means that given a consistent extension T of CL, then it is not decidable whether an equation A = B is derivable in T. In particular this also holds for T = CL.

One may wonder how such a simple axiom system 1.1 gives rise to undecidability. In section 3 we wiil see that combinators form a universal computation model. Therefore CL and consistent extensions are subject to the unsolvability of the halting problem.

2. Control

An important aspect of combinators is that they provide control of application. The following is an example.

2.1 Trick. There are combinators **I**, A and B such that

$$\mathbf{I}X = X;$$

$$AXY = Y;$$

$$BX = XX.$$

for all combinators X and Y.
Proof. Take $\mathbf{I} = \mathbf{SKK}$, $A = \mathbf{SK}$ and $B = \mathbf{S(SKK)(SKK)}$.
Then e.g. $\mathbf{I}X = \mathbf{SKK}X = \mathbf{K}X(\mathbf{K}X) = X;$
$AXY = \mathbf{SK}XY = \mathbf{K}Y(XY) = Y.$ □

The combinator **I** will be encounterd more often.

How does this trick work? In order to understand it we introduce so called *open combinators*, containing variables and define an abstraction operator on these.

2.2. Definition. (i) The set $\mathcal{X}$ of open combinators is defined as follows.

$$\mathbf{K}, \mathbf{S} \in \mathcal{X};$$

$$x_0, x_1, x_2, \ldots \in \mathcal{X};$$

$$A, B \in \mathcal{X} \Rightarrow (AB) \in \mathcal{X}.$$

(ii) If A is an open combinator, then the set of variables in A, notation FV(A), is defined as follows.

$$FV(\mathbf{K}) = FV(\mathbf{S}) = \emptyset;$$

$$FV(x_i) = \{x_i\};$$

$$FV(AB) = FV(A) \cup FV(B).$$

Ordinary combinators are open combinators A with $FV(A) = \emptyset$. These are also called *closed* combinators.

(iii) An open combinator equation A=B is called *valid* if after any substitution of closed combinators for the variables, we obtain an equation $A^S = B^S$ that is derivable from the axioms 1.1. For example $\mathbf{S}(\mathbf{K}xy) = \mathbf{S}x$ is valid. We may think of valid equations as being derived from the axioms 1.1, but with A, B now ranging over open combinators.

Instead of the variables $x_0, x_1, x_2, \ldots$ we often will write x, y, z, ... Outermost parentheses are omitted. So for example S(Kx)(Ky) is an open combinator and

$FV(S(Kx)(Ky)) = \{x,y\}$.

2.3 Theorem. For every open combinator P there exists an open combinator $\lambda x.P$ such that

1. $(\lambda x.P)x = P$ is valid;

2. $FV(\lambda x.P) = FV(P) - \{x\}$.

Proof. Induction on the structure of P. If P=x, then take $\lambda x.P = \mathbf{I}$. If P does not contain x, then take $\lambda x.P = \mathbf{K}P$. If P does contain x and is of the form QR, then take $\lambda x.P = \mathbf{S}(\lambda x.Q)(\lambda x.R)$. □

2.4 Corollary. For every open combinator P and every sequence of variables $\vec{x} = x_1,...,x_n$ there exists an open combinator F such that

1. $F\vec{x} = P$ is valid;

2. $FV(F) = FV(P) - \{\vec{x}\}$.

In particular, if $FV(P) = \{\vec{x}\}$, then F is a (closed) combinator.
Proof. Take $F = \lambda x_1(\lambda x_2 \ldots (\lambda x_n.P)...)$. Then using 2.3 n times it follows that

$Fx_1x_2...x_n = P$ and $FV(F) = FV(P) - \{x_1,...,x_n\}$. □

Notation $\lambda\vec{x}.P = \lambda x_1...x_n.P = \lambda x_1.(\lambda x_2. \ldots (\lambda x_n.P)$. This is the explanation of the trick in 2.1. Indeed $A = \lambda xy.y$, $B = \lambda x.xx$ is another solution for 2.1.

3. Self reproduction

However, one can do much better.

3.1 Trick. There are combinators F and G such that for all combinators X and Y one has

$FXY = XG(YF)$;
$GX = XFG$. □

In this section we show how such combinators can be found.

3.2 Fixed-point theorem. For every combinator F there exists a combinator A such that

$FA = A$.

Proof. Using 2.4 find a combinator D such that

$Dx = F(xx)$.

Take $A = DD$. Then

$A = DD = F(DD) = FA$. □

3.3 Theorem. There exists a fixed-point combinator **Y** such that the fixed-points can be found uniformly:

$A = \mathbf{Y}F \Rightarrow FA = A$.

Proof. Take $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. □

If we write $P[\vec{x}]$, this means that P is an open combinator with free variables among the $\{x\}$. Then $P[\vec{A}]$ is the result of substituting the $\vec{A}$ for the $\vec{x}$ in P.

3.4 Corollary. Let $C[\vec{x}, f]$ be given. Then there exists a combinator F such that

$Fx = C[\vec{x}, F]$.

Proof. Take $F = \mathbf{Y}(\lambda f\vec{x}.C[\vec{x}, f])$. □

For example we can find an F such that
$$FX = XF.$$

3.5 Double fixed-point theorem. Given F, G there exists A, B such that
$$FAB = A$$
$$GAB = B.$$
Proof (Smullyan). By 3.4 let N satisfy
$$Nxyz = x\,(Nyyz)(Nzyz).$$
Take $A = NFFG$ and $B = NGFG$. □

3.6 Corollary. Given $C[\vec{x}, f, g]$ and $D[\vec{y}, f, g]$. Then one can find F and G such that
$$F\,x = C[\vec{x}, F, G]$$
$$G\,y = D[\vec{y}, F, G].$$
Proof. We'd like
$$F = \lambda\vec{x}.C[\vec{x}, F, G] = (\lambda fg\vec{x}.C[\vec{x}, f, g])FG,$$
$$G = \lambda\vec{y}.D[\vec{y}, F, G] = (\lambda fg\vec{y}.D[\vec{y}, f, g])FG.$$
Such F, G exist by 3.5. □

Using corollary 3.6 we can understand how to find the F and G in trick 3.1.

The double fixed-point theorem and its corollary can easily be generalized to n-fold versions.

4. Computing

In sections 2 and 3 we have seen that combinators give control over symbolic manipulations. A fortiori we can obtain control over arithmetic computations.

In order to do this, we have to introduce special combinators for truth values, the conditional, ordered pairs and numerals. Our representation of numerals is motivated by its simplicity. It is not efficient: in order to represent a number n, a combinator of complexity O(n) is needed. A more efficient way of doing numerical computations using combinators is given in van der Poel, Schaap and van de Mey [1980], where one of the representations of number n is of complexity O(log n).

4.1 Definition (truth values). Define
$$\textbf{true} = \mathbf{K} \text{ and } \textbf{false} = \mathbf{KI}.$$
Then $\textbf{true}\,XY = X$ and $\textbf{false}\,XY = Y$.

4.2 Definition (conditional).
The conditional
$$\textbf{if } B \textbf{ then } X \textbf{ else } Y$$
can be represented by
$$BXY.$$
Indeed, if $B = \textbf{true}$, then $BXY = X$ and if $B = \textbf{false}$, then $BXY = Y$.

4.3 Definition (ordered pairs).
Ordered pairs of combinators can be represented as follows.

$[X,Y] = \lambda z.zXY.$

One can reconstruct both components from the pair

$[X,Y]$ **true** $= X,$

$[X,Y]$ **false** $= Y.$

4.4 Definition (numerals). Define

$\underline{0} = \mathbf{I}$

$\underline{n+1} = [\mathbf{false}, \underline{n}]$

4.5 Definition. Let f: N→N be a numerical function. Then f can be *represented by the conbinator* F if for all n

$F\ \underline{n} = \underline{f(n)}.$

Similarly representability for functions of more arguments is defined; e.g. a function g of two arguments is represented by a combinator G if

$G\ \underline{n}\ \underline{m} = \underline{g(n,m)}.$

4.6 Lemma. There are combinators S_+, P_-, $Z_?$ for successor, predecessor and test for zero such that for all n

$S_+\ \underline{n} = \underline{n+1},$

$P_-\ \underline{n+1} = \underline{n},$

$Z_?\ \underline{0} = \mathbf{true},$

$Z_?\ \underline{n+1} = \mathbf{false}.$

Proof. Take $S_+ = \lambda x.[\mathbf{false},x],$

$P_- = \lambda x.x\mathbf{false},$

$Z_? = \lambda x.x\mathbf{true}.$

4.7 Theorem (Kleene). Let f be a computable function. Then f can be represented by a combinator.

Proofsketch. We show how to deal with recursion and minimalization. These are the essential algorithmic components for obtaining the class of computable functions.

Recursion. Let f be defined by

$f(0) = 13$

$f(n+1) = g(f(n)),$

where g is a known function that can be represented by the combinator G. This is a simple version of recursion, called iteration, but the method of representation in typical. One can rewrite f as follows.

$f(x) =$ **if** $x=0$ **then** 13 **else** $g(f(x-1)).$

Now the representation of this f is simply a combinator F such that

$Fx =$ **if** $Z_?x$ **then** $\underline{13}$ **else** $G(F(P\_x)).$

Such an F exists as we saw in section 3.

Minimalisation. Let b be a given computable binary predicate on numbers. We want to compute

$f(x) = \mu y.\ b(x,y)$

i.e. the least y such that b(x,y) holds.

We may assume that b is already represented by the combinator B, that is

$b(n,m) \Leftrightarrow B\ \underline{n}\ \underline{m} =$ **true**

Now f can be represented by F satisfying

$Fx = Hx\underline{0}$

where hxy = **if** Bxy **then** y **else** $Bx(S_+y)$. □

4.8 Trick. Let $x^{\sim n}$ be a sequence of n times x. There exists a combinator P such that

$$P(\lambda x.x^{\sim n}) = \textbf{true} \quad \text{if n is a prime number;}$$
$$= \textbf{false} \quad \text{else.}$$

The solution can be found in van der Poel et al [1980], where the $\lambda x.x^{\sim n}$ are used as numerals.

5. Algebras

The solution of trick 4.8 makes use of the combinator

$$M = \mathbf{SI}$$

that satisfies the following partial associative laws:

$$MMM = M(MM),$$
$$MMMM = M(M(MM)),$$

etcetera.

Note however that (MM)(MM) ≠ MMMM. The LHS has no normal form, but the RHS does. So the associativity is not complete.

The combinator is called M because G. van der Mey proved that

$$MY = Y \Rightarrow YF = F(YF).$$

That is, every fixedpoint of M is a fixedpoint operator.

One may wonder whether there are fully associative combinators.

5.1 Trick. (i) There exist distinct combinators M_e, M_a, M_b, M_c such that these behave like the Klein fourgroup {e, a, b, c}:

$$M_eM_a = M_aM_e = M_a,\ M_aM_a = M_e \text{ etcetera.}$$

(ii) One can also embed a monster group.

The solution shows that the examples are somewhat exagerated.

5.2 Proposition (Barendregt, Dezani and Klop).
Given is a computable applicative structure (A, .). That is, A is a countable set, say $\mathbb{N}$ the set of natural numbers, and . is a computable binary operation on A. Then (A, .) can be embedded isomorphically into the combinators.
Proof. Let A=$\mathbb{N}$ and let a.b = f(a,b) be a computable function. Let the combinator F represent f. Define.

$$M_a = [G,\underline{a}],$$

where G is still to be determined. Compute

$$M_aM_b = [G.\underline{a}][G,\underline{b}]$$
$$= [G,\underline{b}]\ G\ \underline{a}$$
$$= GG\underline{b}\ \underline{a}$$

If we take G = λpqr.[p,Frq], then this computation continues as follows.

$$= [G, F\ \underline{a}\ \underline{b}]$$
$$= [G, \underline{f(a,b)}] = M_{f(a,b)}.$$

Moreover the M_a's are all distinct. Therefore $H(a) = M_a$ is the required embedding. □

By some cardinality argument applied to representable functions it can be shown that not every countable applicative structure can be embedded in the combinators.

It is however possible to embed arbitrary applicative structures in so called combinatory algebras.

5.3 Definition. A *combinatory algebra* is an applicative structure (A, ., K, S) with K, S∈ A such that the axioms 1, 2 and 3 of 1.1. hold.

The following is proved in Engeler [1981].

5.4 Proposition. Given an applicative structure (A, .). Then there exists a combinatory algebra D_A such that (A, .) can be embedded isomorphically into D_A.

6. Equations

An equation P=Q between two combinators is called *consistent* if from P=Q one cannot derive **K=S**. Equivalently, if P=Q is true in some combinatory algebra.
For example

K=KKK is consistent,
I=S is inconsistent.

The first equation is derivable. From **I=S** one derives **IKKS = SKKS**, hence **K=S.**

Tnere are also 'independent' equations, i.e. consistent ones that cannot be derived.

6.1 Proposition. Let Ω = **SII(SII)**. Then for every combinator P one has that Ω=P is consistent.
Proof. Jacopini [1975] has shown this by a proof theoretic argument. Baeten end Boerboom [1979] have shown this by constructing combinatory algebras. □

6.2 Trick (Bel). There exists a magic triple, that three combinators P, Q, R such that

P=Q is consistent
Q=R is consistent
R=P is consistent

but P=Q=R is not consistent.

Proof. Take P=**I**, Q=Ω and R=Ω**S**.
That **I**=Ω is consistent follows from 6.1. Similarly Ω=Ω**S** is consistent, since the equation **I** = Ω**S** follows from Ω=**KI** and we can use 6.1.
But **I**=Ω=Ω**S** implies **I**=**IS**=**S** and we have seen that this equation is inconsistent. □

M. Bel has generalised his trick to magic n-tuples.

References

Barendregt, H.P.
[1984] *The lambda calculus, its syntax and semantics*, North Holland, Amsterdam.

Baeten, J. and B.Boerboom
[1979] Ω can be anything it shouldn't be, *Indag.Math.* 41, 111-120.

Engeler, E.
[1981] Algebras and combinatory, *Algebra universalis* 13 no. 3, 289-392.

Jacopini, G.
[1975] A condition for identifying two elements of whatever model of combinatory logic, Springer LMCS 37, 213-219.

van der Poel, W.L., C.E. Schaap and G. van der Mey
[1980] New arithmetical operators in the theory of combinators, *Proc. Kon. Ned. Ak. Wetensch.*, Series A, vol. 83 no. 3, 271-325.

Prof. dr. ir. W.L. van der Poel

gezien door de ogen van zijn staf

Beste Wim

Het lijdt in onze ogen geen twijfel dat je een veelzijdig mens bent. Zo iemand beschrijven in een enkele bladzij, of zelfs in een paar bladzijden, is ondoenlijk en doet geen recht aan de persoon. Desalniettemin vraagt de gelegenheid om een korte beschrijving van jou als hoogleraar binnen de vakgroep Technische Informatica van de TU Delft. Wij, je staf, geven daarom een - sterk gekleurde en incomplete - beschrijving van jou, onze eigen interpretatie van de hoogleraar Prof. van der Poel.

Je gezicht naar de buitenwereld

Je carrière in de wereld van de electronische rekenautomaten begon al vroeg. Reeds in de jaren veertig maakte je het ontwerp van een rekenmachine. Je carrière begon internationaal bezien zelfs zo vroeg in het begin van het vakgebied dat je in 1985 de *Computer Pioneer Award* van de IEEE kreeg uitgereikt.

Tijdens je studie, Technische Natuurkunde, participeerde je actief in de ontwikkelingen rond de Testudo, een relais-gebaseerde rekenmachine. Een aardige karakteristiek van de machine was dat - naar je eigen zeggen - berekeningen die door een persoon in ca. 8 uren konden worden gedaan, 16 machine-uren vroegen. Door de machine 's nachts op het karwei te zetten was toch iedereen tevreden. 's Morgens op het werk had men dan de uitkomsten van de berekeningen van de vorige nacht.

Na je studententijd op de (toen nog) T.H. ging je bij het dr. Neher Laboratorium van de PTT in Leidschendam werken. Onder leiding van prof. Kosten ontwikkelden jullie daar de PTERA, De PTT Eenvoudige RekenAutomaat. Later werd de ZEBRA ontwikkeld, de Zeer Eenvoudige Binaire RekenAutomaat, een computersysteem met een drum als geheugen, en een kiesschijf voor het invoeren van data. Deze laatste machine, waar je proefschrift betrekking op had, is ook commercieel ontwikkeld. Diverse exemplaren hebben in Nederlandse rekencentra gestaan. Eén ervan stond op het rekencentrum (de Wiskundige Dienst) van de T.H..

In 1962 werd je benoemd als hoogleraar op een bijzondere stoel: *Logica gericht op de toepassing van rekenautomaten.* In 1966 volgde je benoeming als buitengewoon hoogleraar en al snel, een jaar later, werd je benoemd tot gewoon hoogleraar in de "Zuivere en toegepaste wiskunde".

Ook in de internationale wereld roerde jij je. Je werd voorzitter van IFIP WG 2.1, een werkgroep die een belangrijke rol had bij het tot stand komen van ALGOL-60. Na de ontwikkeling van ALGOL-60 was je sterk betrokken bij het tot standkomen van ALGOL-68. Vanuit je positie in IFIP WG 2.1 en de enorme belangstelling voor systeemimplementatietalen, heb je aan de wieg gestaan van IFIP WG 2.4; de werkgroep die zich onder andere met systeemimplementatietalen bezig houdt. Je trad op als editor van de Working Conference (1973) en bent sindsdien een trouw bezoeker van de werkgroepbijeenkomsten geweest.

Zo ben je ook betrokken geweest bij de Normalisatie, bij de Nederlandse Programmeertalen Commissie en (ook internationaal) bij TC-97.

Je gezicht naar het werk

Je spreidt een enorm enthousiasme in alles wat jouw interesse heeft ten toon. Dit enthousiasme wordt, zowel naar je staf als naar de meer dan honderd afstudeerders die je hebt gehad, uitgestraald. Altijd was - en ben - je bereid om over die dingen waar je enthousiasme op dat moment naar uitging gevraagd en ongevraagd uit te weiden en advies te verlenen.

De zaken waar je belangstelling naar uitgaat beïnvloeden ook je manier van werken. Om de belangrijkste te noemen: *puzzelen en goochelen* en *eenvoud en orthogonaliteit.*

1. *Puzzelen* doe je om complexe problemen te kraken en de oplossing tot iets hanteerbaars terug te brengen; *Goochelen* doe je vervolgens om de gevonden eenvoudige oplossing buiten het bereik van je publiek te houden of te brengen;

2. *Eenvoud* en *orthogonaliteit* komen tot uiting in je voorliefde voor kleine primitieven en orthogonale koppelingsmechanismen. Op basis van enkele (syntactisch en conceptueel) kleine primitieven en zo'n orthogonaal uitbreidingsmechanisme wordt alles ontwikkeld. Een uitspraak die bij ons bekend staat als de *krijtstelling*: iets dat op een schoolbord niet met een enkele kleur krijt duidelijk kan worden gemaakt, kan ook niet duidelijk worden gemaakt door van meer kleuren gebruik te maken, is volgens ons ook een voorbeeld van je voorliefde voor eenvoud.

Je hebt een uitgesproken belangstelling voor (rare) puzzeltjes. Niet alleen op het gebied van complexe in-elkaar-zet constructies gebaseerd op eenvoudige blokjes, die wij ook mochten proberen, maar ook op het gebied van computers zelf. Even een voorbeeldje met betrekking tot het vakgebied. Je gaf eens één van de aardigste oplossingen voor het niet reële probleem van wat het minimale aantal instructies waarmee een instructieset moet worden uitgerust is. Het blijkt een enkele instructie te zijn. Lang voordat de RISC architecturen in de mode begonnen te komen had jij er al een voorkeur voor. (Gemakshalve noemen we een PDP-8 ook maar een RISC.) Volgens een artikel uit 1952 heeft de eenvoudigste machine slechts één enkele instructie, de ABA (Aftrek en Berg op in Accu), tel daarbij je uitspraak op over het gewenste aantal resources: *nul, een of oneindig veel*, en je komt op een zeer speciale RISC uit.

Een andere wetenschap (hobby), die naar ons idee tevens voortvloeit uit de mogelijkheid om flink te kunnen puzzelen heeft betrekking op combinatoren en lambdavormen. In tegenstelling tot vele anderen die zich met dit gebied bezighouden, heb je een pragmatische kijk op de combinatoren. Een belangrijke randvoorwaarde is dat een oplossing op de machine te doen moet zijn. Een verder uitgangspunt van al het werk op het gebied van combinatoren, is dat het op de computer zichtbaar gemaakt moet kunnen worden. Hier komt je praktische benadering sterk tot uitdrukking in de vele "vondsten" die je deed, geholpen door studenten die je door jouw enthousiasme aanstak om mee te puzzelen en te programmeren. Veel taak- en afstudeerwerken heb je laten verrichten op dit gebied. In de loop der jaren is er, mede door deze werken, een enorme expertise

verkregen.

Het gebied van de cryptografie ligt, voor ons als relatieve buitenstaanders op dit gebied, ook in de puzzelsfeer. Door computers werd er nachten lang gerekend aan bepaalde priemeigenschappen die belangrijk zijn voor cryptografische sleutels. Je bent de mening toegedaan dat computerprogramma's openbaar moeten zijn. Iets dat je nastreeft door programma's van anderen te disassembleren of te laten herschrijven, zie o.a. D. Sandee met PDP-8 TRAC, R. Dekkers met een openbare (niet complete) C implementatie van MS-DOS en J. Wesselius met een disassemblage van Methods. Ondanks deze mening is een van jouw belangstellingsgebieden het beschermen van programmatuur tegen illegaal gebruik. Ook deze interesse is al geruime tijd aanwezig. N. Leijnse was de eerste die haar afstudeerwerk op dit gebied deed. Sindsdien zijn er verschillende studenten geweest die in het kader van taak- of afstudeerwerk programmatuur om te kraken of om kraken tegen te gaan hebben ontwikkeld. Eerst werd een versie van een protectieprogramma onder CP/M ontwikkeld, waarbij het programma gedeeltelijk in het schermgeheugen zat. Later heeft dit o.a. geleid tot een beschermingsprogramma onder MS/DOS. Dit programma, door jou ontwikkeld vanuit programmatuur die door studenten is gemaakt, gold als onkraakbaar door anderen. De enige manier om het te kraken was je bekend. Nadat een slimme student ook die manier had gevonden, werd het programma zodanig verbeterd dat het nu zeker kraak-proof moet zijn.

Overdraagbaarheid was ook een stokpaardje van je. Toen niemand nog over overdraagbaarheid sprak hadden wij het (om een gevleugelde uitdrukking te gebruiken) al hoog in het vaandel staan. Kenmerken van overdraagbare programmatuur werden sterk benadrukt, ook al bleek de feitelijke portabiliteit van veel programmatuur niet verder te gaan dan de portabiliteit van de listing. Om die overdraagbaarheid te vergemakkelijken, moesten nieuw aan te schaffen machines minimaal in staat zijn software van de oude te lezen.

Een detail uitvergroot

Op het specifieke vakgebied "Programmeertalen" (een naam die de laatste jaren het belangrijkste label was van de groep die onder je leiding stond) was je interesse herkenbaar en in een zekere mate voorspelbaar. Gegeven de eerder genoemde kenmerken van je belangstelling was het duidelijk dat je interesse voor massieve, alomvattende talen zoals PL/I en Ada beperkt zou zijn. Ook hier geldt je voorliefde voor het kleine; kleine taaltjes en systeempjes waren duidelijk favoriet. Een paar voorbeelden:

1. Lisp en andere functionele talen. Lisp, zeker de Lisp die we kenden op de PDP-8, had een klein aantal primitieven en was op een - min of meer - orthogonale manier uitbreidbaar. Het was dus een schoolvoorbeeld van hoe een taal volgens jou eruit moet zien. Je belangstelling voor Lisp heeft recentelijk geleid tot een actief participeren in Eulisp (in het bijzonder om te verhinderen dat een ISO-Lisp zou bestaan uit een enorm Lisp-dialect). Momenteel ben je lid van de ISO-Lisp werkgroep. Je belangstelling heeft geleid tot het uitproberen van specifieke implementatietechnieken zoals: HISP (halfwoords-Lisp op de PDP-9), ontwikkeld in samenwerking met van der Mey; Lisp op de PDP-11 onder RT/11 in Harvalias geschreven, (den Hoed); CLisp,

wederom een overdraagbare Lisp implementatie, geschreven in C (Barkey); en een nieuwe Lisp in C (Steenweg). Al deze implementaties waren er mede op gericht een maximale overdraagbaarheid te bereiken. Een van de eerste afstudeerwerken die je uitdeelde was het schrijven van een Lisp interpretator in ALGOL (Wilgenburg).

2. Algol-68 heeft je belangstelling natuurlijk mede door je actieve lidmaatschap (en in de periode dat de taal werd ontwikkeld je voorzitterschap) van IFIP WG 2.1, de groep waarin de taal werd ontwikkeld. Technisch gesproken heeft het je belangstelling meer vanwege de orthogonaliteit dan vanwege de omvang van de taal. Je hebt een actieve rol kunnen spelen in de ontwikkeling van een ALGOL-68 compiler, waarop in dit symposiumverslag nog wordt teruggekomen. In het midden van de jaren '70 heeft een aantal studenten (C. Hemerik, E. van Holz) een subset-vertaler gemaakt en kwam je betrokkenheid bij de constructie van ALGOL-68 tot uiting bij het afstudeerwerk van Van Hedel.

3. Eenvoudige systeemimplementatietalen. Van het begin af aan was je sterk verbonden met de ontwikkeling van kleine systeem-implementatietalen in huis. Nadat AAS (*A*lgorithmic *AS*sembler) was ontwikkeld op de PDP-9 (later ALIAS genoemd omdat de naam AAS al voor *A*n *A*dressing *S*ystem werd gebruikt aan de T.H. Twente), bracht je van een van je reizen MiniAl mee. In totaal hebben acht studenten in hun afstudeerperiode gewerkt aan MiniAl en MidiAl compilers: op de PDP-8 (Luijff), op de PDP-9 (Boiten), op de PDP-11 (van Zanten, van Loon, Kamphorst, den Eersten en Schiet) en, meer recentelijk, op de micro's (Werry). Overigens bevestigden de diverse implementaties van MidiAl op de PDP-11 je vooroordelen over complexiteit: de PDP-8 implementatie van MidiAl bleef maar sneller lopen dan de implementaties op de veel complexere PDP-11.
   Daarnaast werd op de PDP-11 de ontwikkeling van een eigen low-level taal voortgezet met de definitie en implementatie van HARVALIAS, een kruising tussen HARVEY en ALIAS (P. van der Eijk).
   Een specifieke klasse van systeemimplementatietalen is de klasse der draadcode-talen waarvan Forth de exponent bij uitstek is. Snelle implementaties werden gemaakt (Oort en Vollebregt). Een gewone Forth was niet goed genoeg, een eigen High-Level taal op basis van Forth moest ontwikkeld en geïmplementeerd worden (C. van de Hoek).

4. Een andere categorie van talen die - niet verwonderlijk vanwege hun karakteristieken - je belangstelling bleef houden was de categorie van de *gegeneraliseerde macro-processoren.* In het bijzonder TRAC kon je boeien. Nadat de PDP-8 implementatie gedisassembleerd was (Sandee), werden andere implementaties (TRIC, TRUC, TRAC860 en TRAC11) gemaakt (van Katwijk, Mevius, Monteban, en Langelaan). De semantiek van TRAC was bovendien een bewijs van je stelling: *er bestaat geen fout programma, het is gewoon een ander programma.* Maar niet alleen TRAC had je belangstelling, GPM, Stage II en ML/I vond je tenminste even interessant en ook op deze terreinen werden taakwerkers en afstudeerders ingezet.

5. Tenslotte, wederom de combinatoren en de lambda-calculus. Tijdens een bezoek aan IBM (1975) legde je de basis voor het onderzoek aan combinatoren met een eerste

Lisp-implementatie. Sinds die tijd heeft een enorm aantal taakwerkers en afstudeerders gewerkt aan de ontwikkeling van implementaties voor de combinatoren (van der Eijk, Van der Meij en Bekkers) of de verdere ontwikkeling van "Trucs en vondsten" (Schaap, Leeflang)

De mensen om je heen

Toen je in 1967 als gewoon hoogleraar kwam, stond er geen uitgebreide staf klaar. Je eerste medewerker was J. Jager (een eigen afstudeerder), al snel kwamen daar H. Barreveld en C. Stillebroer bij. Ook F. Hogesteeger kwam je staf versterken. Uiteraard wisselde er nogal eens wat. Nadat C. Stillebroer vertrok werd F. Ververs aangesteld en nadat een halve formatieplaats beschikbaar kwam door CRIVA activiteiten van Barreveld, werd een assistentenplaats ingeruild om ook J. van Katwijk aan je staf toe te kunnen voegen. F. Ververs en J. van Katwijk zijn je tot het eind toe trouw gebleven. In 1973 vertrok H. Barreveld naar Sara, als opvolger kwam in 1974 T. Verhaar. Deze had als uitdrukkelijke opdracht het op zich nemen van de organisatie. In het midden van de jaren '70 werd, in afwachting van de komst van een hoogleraar Operating Systems, J. Aalders bij je geparkeerd. Toen I. Herschberg kwam ging J. Aalders naar hem toe. Eind 70'er jaren kwam P. van der Hoff overwaaien van de groep Numerieke Wiskunde.
Met het opgang komen van de opleiding tot informatica-ingenieur groeide de vakgroep. Het zg. amoebe-plan werd aangenomen, een plan waarbij elke leerstoel van de toenmalige vakgroep tot ca. 10 mensen zou groeien en waarna elke groep zou worden gesplitst in twee deelgroepen. Hoewel je van kleinschalig houdt, groeide jouw groep inderdaad tot ca. 10 personen. In 1982 kwamen C. Pronk, R. Huijsman en H. de Hartog de groep versterken. Na het vertrek van H. de Hartog kwamen daar A. Hartveld en J. van der Linde bij. Ook deze twee zijn inmiddels weer vertrokken. Dit vertrek samen met het overlijden van T. Verhaar gaf plaats voor H. Toetenel en L. Dusink en, meer recent, aan P. Kluit. De plaats van P. van der Hoff, die gedetacheerd is als hoofd PCF, is bezet door P. Kruszinsky, terwijl G. Kolk op tijdelijke basis aan de groep is toegevoegd.
Ook op technisch gebied werd de ondersteuning groter, in 1980 kwam T. Biegstraaten, in 1982 D. Kamstra. D. Kamstra verdween en voor hem kwam in 1988 H. Engelen.
Het grote aantal stafleden heeft alles te maken met het groeien van de Informatica opleiding en was bedoeld als basis voor de twee leerstoelen *Programmeertalen en compilers* en *Software Engineering.* Naar het zich laat aanzien zal de afsplitsing van een leerstoel Software Engineering zich in de zeer nabije toekomst voltrekken.

Je was de eerste hoogleraar "Informatica" binnen de afdeling en hebt dus ook de "oudste" afstudeerders van alle Informatica hoogleraren binnen de faculteit. Je eerste afstudeerder was Okker, in 1964. Als we de lijst van afstudeerders en afstudeerwerken doorspitten zijn de titels van afstudeeronderzoeken herkenbaar. In je collegegeven was (ben) je zo enthousiast dat dat een bepaalde soort studenten aantrekt als afstudeerder. Het valt je niet moeilijk om studenten te motiveren een complex stuk doe-werk aan te vatten.
In dit boekwerk is een lijst van afgestudeerden met de titel van hun afstudeerwerk opgenomen. Duidelijke kenmerken van afstudeerwerken zijn:

1. Practisch en constructief: een afstudeerwerk was pas dan compleet als er een listing bijzat. Een demonstratie van de gemaakte programmatuur was een absolute

voorwaarde.

2. Probleemoplossend: de meeste afstudeerwerken die door jou werden uitgedeeld waren bedoeld om een probleem - dat op dat moment sterk in je belangstelling stond - op te lossen of aan te vatten.

3. Begrensd in omvang: een afstudeerder kon ervan op aan, behoudens een enkele uitzondering, dat zijn/haar werk behoorlijk in de daartoe beschikbare tijd kon worden gedaan.

In het voorafgaande is een aantal afstudeerwerken al voor het voetlicht gekomen; het aantal implementaties van talen, taaltjes, systemen en systeempjes is groot geweest. Nu, bij je afscheid, zijn 115 studenten bij je afgestudeerd en staan er ca. 30 ingeschreven met een vakkenpakket met jouw handtekening eronder.

De titel van het symposium

Bij het denken over een titel voor het symposium hebben veel gedachten en opmerkingen ons door het hoofd gespeeld. Iets met spelletjes? Iets met een priemgetal? Iets met een PDP-8? Totdat we een inleidend artikel van je eigen hand vonden, een artikel bij de opening van de 1973 Working Conference on Systems Implementation Languages (de conferentie waarbij IFIP WG 2.4 werd opgericht door jou en anderen). Naar ons idee was een uitspraak in dat artikel karakteristiek voor de manier waarop je blijft doorgaan de problemen die je interesse hebben aan te vallen:

Vooruitgang: Bit voor Bit

Frans
Gea
Hans
Hub
Jan
Kees
Liesbeth
Pawel
Peter
Ronald
Ton
Trudie

Belevenissen met van der Poel

Prof. dr. H.J.A. Duparc

Wim van der Poel is afgestudeerd als natuurkundig ingenieur en neemt voor de tweede keer afscheid van onze instelling, nu als hoogleraar in één der vakgroepen informatica. Wat zou iemand als ik die wiskunde heeft gestudeerd en dat vak in zijn diverse facetten trouw trachtte te blijven voor zinnigs over van der Poel kunnen schrijven?

Dat deze bijna rhetorische vraag een duidelijk positief antwoord verdient, komt door de veelzijdigheid van van der Poel. Hij heeft een grote belangstelling voor bepaalde gedeelten van de wiskunde en daarin vonden wij elkaar steeds weer. Daarbij gaat het helemaal niet om de traditionele Delftse analyse, maar veel meer om zaken die men tegenwoordig met de naam discrete wiskunde aanduidt. Die naam was echter nog niet uitgevonden toen ik ruim 35 jaar geleden van der Poel voor het eerst ontmoette op het Mathematisch Centrum. Ik had mijn proefschrift juist voltooid toen van der Poel met een vraag kwam aanzetten waarvan ik de beantwoording als een vervolg op dat proefschrift kon zien. Het ging om de vraag met een getallentheoretische achtergrond; men kan wel zeggen dat die model heeft gestaan voor verdere verwante vragen waarmee van der Poel mij in de loop der volgende jaren heeft bestookt. Laten we eerst een aanloopje nemen naar zijn eigenlijke gedachten in de vorm van het volgende probleem.

Als een bepaalde eigenschap geldt voor priemgetallen, in hoeverre kan men dan omgekeerd uit het gelden van die eigenschap concluderen dat het erin optredende getal ondeelbaar is? Als die conclusie niet gewettigd is, tabelleer dan uitzonderingen en onderzoek ze op hun frequentie. Na dit aanloopje volgt nu de hoofdgedachte van van der Poel.

Beschouw nu twee eigenschappen die gelden voor priemgetallen. Stel dat ze beide gelden voor eenzelfde getal. Dan is de kans dat dat getal geen priemgetal is uiteraard kleiner, vaak veel kleiner dan in het eerder genoemde voorbeeld. Wie weet is het getal zelfs noodzakelijkerwijze ondeelbaar! Dat zou te bewijzen moeten zijn. En als het toch niet ondeelbaar behoeft te zijn, tabelleer dan de in dit geval veel schaarsere uitzonderingen. Onnodig wellicht te zeggen dat het hier gaat om eigenschappen die met een rekentuig gemakkelijk te onderzoeken zijn. Tenslotte leefde van der Poel al in die dagen in de wereld van de rekenautomaten.

Wij geven van het bovenstaande een voorbeeld. Als (eerste) eigenschap nemen wij een speciaal geval van de (kleine) stelling van Fermat: Als p een oneven ondeelbaar getal is, geldt:

$$2^{p-1} \equiv 1 \pmod{p}.$$

De lang niet schaarse uitzonderingen op de omkering van deze stelling (er zijn er zelfs oneindig veel) zijn reeds een halve eeuw geleden door Poulet een aardig eindje getabelleerd. Naar het bovengenoemde concept van van der Poel moet men nu een tweede eigenschap erbij nemen. Stel dat men daarvoor weer iets Fermat-achtigs neemt,

bijvoorbeeld:

$$3^{p-1} \equiv 1 \pmod{p} \quad \text{met } p \neq 3,$$

dan blijken er nog legio samengestelde getallen te zijn, die beide eigenschappen hebben. Sterker nog, er bestaan samengestelde getallen m die voldoen aan:

$$a^{m-1} \equiv 1 \pmod{m}$$

die voor alle a, voorzover die relatief priem zijn met m. Van deze getallen, Carmichaelgetallen genoemd, weten wij overigens niet of er oneindig veel bestaan.

Terecht koos van der Poel voor zijn tweede eigenschap één van een ander type , namelijk een eigenschap die ontleend was aan de welbekende rij van Fibonacci. Het ging daarbij om de relatie

$$v_p \equiv 1 \pmod{p},$$

waarbij v_n de zogenaamde geassocieerde rij is van die van Fibonacci, ook wel de rij van Lucas genoemd. Samengestelde getallen die zowel aan de relatie van Poulet als aan die van Lucas voldoen, heb ik getallen van van der Poel genoemd maar van der Poel wilde er, bescheiden als hij is, niet aan. Jammer.

De hier genoemde vragen hadden een practische achtergrond. Bepaalde processen in de wiskunde waarbij priemgetallen in het spel zijn, kunnen - al zijn die processen a priori eindig en dus cyclisch - soms lang worden uitgevoerd aleer herhaling optreedt. In diverse toepassingsgebieden kunnen die van nut zijn; wij noemen hier naast de stochastiek (het maken van pseudo-aselecte getallenrijen) de cryptografie. Vaak lukt het die processen "on line" te laten plaatsvinden met hun toepassingen.

Het bleef overigens niet bij processen van het hier beschreven type. De vindingrijke van der Poel kwam vervolgens aanzetten met processen die ik later u-v-processen dan wel u-v-w-processen heb genoemd. Ter illustratie (waarbij alles zich afspeelt in het gebied der natuurlijke getallen):

Construeer een rij getallen g_n volgens het volgende principe.
Schrijf elke g_n gedeeltelijk in het m-tallig stelsel met behulp van de formule

$$g_n = m\, v_n + u_n \quad (\text{met } u_n < m).$$

Definieer vervolgens g_{n+1} met behulp van de formule

$$g_{n+1} = a\, u_n + b\, v_n.$$

Indien geldt $b < m$ is de rij begrensd en dus cyclisch, waarna de vraag opkomt naar de cycluslengte, al dan niet afhankelijk van de keuze van g_0. Het succes blijkt niet grandioos te zijn, het proces heeft iets Fibonacci-achtigs en het was juist de bedoeling iets beters te krijgen. Welnu, de generalisatie lag voor de hand. Stel

$$g_n = m^2 w_n + m v_n + u_n \quad (\text{met } v_n < m \text{ en } u_n < m)$$

en definieer dan

$$g_{n+1} = au_n + bv_n + cw_n.$$

Bij $c < m$ is ook dit proces eindig en dus cyclisch. Helaas, ondanks veel feitenmateriaal hebben we ze nog niet doorgrond en van der Poels informaticabroer van Wijngaarden kwam tot een vreemdsoortige argumentatie die ons moest doen inzien waarom wij er met bepaalde methoden niet achter zouden kunnen komen. Helaas ben ik niet (meer) in staat die hier weer te geven.

Ik introduceerde hier van Wijngaarden bij wie van der Poel op het Mathematisch Centrum af en toe binnenliep en waar trouwens mijn eerste ontmoeting met van der Poel plaatsvond. Tekenend voor die twee Nederlandse nestoren in de informatica is hun belangstelling voor de getallentheorie, een belangstelling die voortvloeide uit hun verwachting dat dit vak van veel nut zou kunnen zijn voor de informatica. In mijn eigen proefschrift wijdde ik twee stellingen (niet de laatste...) aan de wisselwerking tussen algebra/getallentheorie en informatica, iets dat thans al decennia gemeengoed is voor die talloze informatici die dat willen inzien en die getallentheoretici/algebraïci die bereid zijn zich daarop te richten.

Er is meer dat van der Poel en van Wijngaarden bond. In januari 1956 promoveerde van der Poel bij van Wijngaarden aan de gemeente-universiteit van Amsterdam tot doctor in de wis- en natuurkunde. Ik had het genoegen daarbij één van van der Poels paranymfen te mogen zijn. Weinig konden wij drieën toen vermoeden dat een kleine kwart eeuw later de geschiedenis zich in zekere zin zou herhalen, toen in januari 1979 van Wijngaarden een eredoctoraat aangeboden kreeg aan de Technische Hogeschool Delft met van der Poel als erepromotor en ik wederom als één der paranymfen. Men onderkent hun wederzijdse wetenschappelijke verwantschap, leidende tot lering en verering.

Al dan niet in het kader van het bovenstaande, ben ik er trots op en blij over het ten tijde van mijn voorzitterschap van de afdeling voor elkaar te hebben gekregen dat van der Poel definitief bij ons kwam, nadat ik hem eerder al als buitengewoon hoogleraar bij ons had mogen begroeten. Nog levendig herinner ik mij de bespreking bij mij thuis, tezamen met Timman en van Spiegel over van der Poels definitieve komst naar onze onderafdeling, waarbij van der Poel weigerde te komen omdat hij het immoreel vond zijn computerfirma in de steek te laten. Je kunt de industrie blijkbaar ook trouw blijven uit principiële, niet financiële overwegingen. Echter, toen de volgende dag bleek dat die computerfirma werd opgeslokt door een andere stond niets zijn komst naar ons meer in de weg.

En zo kwamen wij weer wat dichter bij elkaar te leven en te werken, hetgeen resulteerde in verdere contacten waarbij nieuwe problemen door van der Poel aan de orde werden gesteld, veelal met dezelfde bedoeling als de bovengenoemde. Zij getuigden van van der Poels niet-aflatende inventiviteit. Die uitte zich overigens niet allen op het hier beschreven gebied (wij weten wel beter), maar ik ben niet in staat al die andere aktiviteiten zo te doorgronden als die op ons aanrakingsgebied. Het is dus maar goed dat ik niet de enige hoef te zijn die van der Poels Delftse werk belicht, want anders zouden wij hem - veelzijdig als hij is - zeer onterecht veel onrecht aandoen.

Language Extensions to Allow Rapid Mode Shifting in the Ada[1] Programming Language

J. van Katwijk
W.J. Toetenel

Faculty of Mathematics and Computer Science
Delft University of Technology
Delft
The Netherlands

1. Introduction

The Ada language [ARM-83] has been designed with applications in the embedded systems domain in mind. Ever since its actual use, comments are made on problems with its effectiveness in applications in the aforementionned domain. Generally three areas of concern are identified: (i) (potential) problems with the efficiency of the implementation of various tasking constructs (see for an overview of problems and some performance data e.g. [Huijsman-87] and [Katwijk-88]), (ii) problems with the distribution of Ada tasking programs over multiprocessor systems and computer networks (see e.g. [Atkinson-88]) and, (iii) limitations in the way tasks may influence each other (see for an overview of the problem area e.g. [Workshop-88]). In a workshop on *Real-time Ada Issues* [Workshop-88] the topic of having tasks influenced each other was central; several authors proposed language extensions to allow a smooth implementation of this *rapid mode shifting.* Of course any extension, including the ones proposed here, should be used reluctantly. The nature of such language extensions, though oriented towards expressing changes of control, deviates from the existing constructs influencing change-of-control such as *if-then-else* statements and *case* statements.

In order to address this third area of concern, we developed some language extensions that were presented at the aforementioned workshop and, based on experiences with these extensions and results of the workshop, we redesigned these extensions. In this paper these language extensions are discussed. First, the need for language extensions with which tasks have possibilities to influence each other is exemplified. Next, a set of straight-forward language extensions addressing this problem are discussed (these language extensions were the main topic of a previous paper [Toetenel-88a]) and the limitations of the extensions are given. The main topic of this paper is a second designed set of language extensions to address the problem of tasks influencing each other. We discuss the relative merits of these extensions and we present an example of their use.

---

1. Ada is a trademark of the U.S. Government (AJPO)

At Delft University of Technology we have developed a prototype implementation of almost the whole Ada programming language (the so-called Ada— implementation [Katwijk-87]). Although the implemented language has some restrictions, it includes the full Ada tasking semantics. During the implementation, modeling the implementation of Ada tasking has got quite some attention; the annotated listing of the tasking kernel used in our implementation is presented in [Toetenel-88b]. One of the obvious advantages of having such an implementation is that it gives us a perfect vehicle for excercising implementation techniques and language extensions. We did already implement and excercise the extensions discussed in [Toetenel-88a] (briefly recalled in section 3). Currently we are in the process of implementing the language extensions that are proposed in this paper. Main objective of this implementation effort is to gain some experience in their practical use.

2. The need for asynchronous transfer of control

A general issue in real-time applications is the ease with which a so-called *mode shift* within an executing task can be executed. The main characteristic of such a mode shift is that a program performing a certain activity is forced to switch quickly into doing something else as a reaction on some external stimuli. It is vital that a request for a mode shift is obeyed fast. Indeed, reaction within an acceptable time limit on the external stimulus that triggered the mode shift is often required by the event causing the external stimulus. A particular case where mode-shifts often are required as a means to solve a problem, is in real-time applications is when there is an explicit dependency of program parts on time constraints. In particular, cyclic tasks e.g. often have the property that the actions specified to be performed within a cycle, have to be performed *within* a given amount of time. Not obeying the constraint on this time limit may give rise to timing-error conditions. Recovery from such an erroneous situation is usually implemented using a construction similar to a mode shift: the task object is brought into some form of *error mode*, in which an attempt is made to recover from the consequences of the error and - possibly - to repair the cause of the error.

The Ada language in its current form enforces the use of polling in implementing a mode shift. Although of polling may lead to an acceptable solution in situations that are *not* time critical, for most time-critical applications the method leads to an unacceptable overhead. Polling is usually implemented in terms of conditional entry call and conditional selective wait constructs. It is believed that the application of these constructs is rather expensive in terms of time (this assumption is acknowledged by empirical results as reported in e.g. [Katwijk-88]) and a situation in which the overhead is deemed unacceptable is soon reached.

The only mechanism provided by the Ada programming language for a task to *asynchronously* affect the behaviour of another task (a prerequisite for a cheap implementation of mode shifts) is the *abort* statement. A first order approximation to the implementation of an asynchronous change in control can be obtained by aborting a task and replacing it with a new instance. Such a solution is hardly appropriate, it is crude and - as discussed by various authors, e.g. [Baker-88] - does not always give the desired results.

Another solution that at first side seems applicable is to have a distinct task object for each mode a task can be in, and to manipulate the priorities of the task objects to achieve the desired effect. A task object that implements an active mode will run with a high priority, tasks corresponding to modes that are not currently active, will get a low priority. Drawbacks of such a solution are twofold. First, even tasks with a low priority may become active if no other tasks are eligible (implying that the task representing the mode not to be executed will execute sooner or later). Second, dynamically changing of priorities is not supported in the Ada language.

Finally, as mentioned earlier, polling is a line of solution believed to involve a dramatic overhead. We believe therefore that in its current form the Ada programming language is not properly equipped with language constructs supporting the implementation of rapid mode shifting. An obvious alternative is to extend the language with suitable constructs.

3. Asynchronous transfer of control: a simple approach

It may be clear from the previous section that, in order to allow the implementation of rapid mode shifting, other approaches are required. If language extensions are considered, extensions supporting some form of asynchronous interruption of the flow of control come first in mind, due to the nature of the problem.

A historical survey shows that two different points of view are apparent when discussing interruptions of the flow of control (see e.g. [Ghezzi-87]). One view supports implicitly called units as procedure calls on receiving a signal, with control returning to the point where the exception occurred (i.e. a *resumption* model). In the other view an exception is reacted upon by *terminating* the currently executing unit with a transfer of control to an exception handler (i.e. a *termination* model). The latter model is the one adopted for handling exceptions in the Ada language. An approach in this spirit seems therefore a suitable basis for extending the Ada programming language. It is indeed a tempting thought to extent the notion of exception handling to allow the raising of an exception in another thread of control. Syntactically, such an extension may look like:

raise *exception_name* **in** *some_task*

In this extended raise statement the notion *exception_name* identifies the exception to be raised, the notion *some_task* identifies the thread of control that is to be interrupted by the exception.

The semantics of raising an exception in another thread of control are, roughly speaking, the same as the semantics of ordinary exceptions. Whenever such an exception is raised, control in the receiving thread of control is transferred to an exception handler enclosing - in the same sense as with ordinary exceptions - the point where the exception is raised.

This solution seems perfectly reasonable from the actor's point of view: once a condition is detected in which another task is to be notified, a remote exception is raised. On the other hand, from the server's point of view this solution may not be desirable at all. Occurrence of exceptions is completely unpredictable and the server does not have any

possibility at all to prevent a dictated change in the flow of control which results from an exception raised by an actor.

Other problems, elaborated on in [Toetenel-88a] have to do with semantics, e.g. what is the meaning of an exception raised by another thread of control when the server is itself in the process of handling an exception, prior to dispatching the latter?

In general, the structure of a server, anticipating a remote exception, will be as depicted below.

```
task body example is
begin
  loop
    begin
          --| do some periodic actions
          --| actions to be performed
    exception
      when some_exception => ... --| normal handling of exceptions
      when special_exception => .. --| react on mode shift
                              --| repair whatever is to be repaired
    end
  end loop;
end example;
```

As may be clear from this structure, the computations to be performed in a cycle may get lost completely once an exception is raised by another thread of control. Our experiences convinced us that the server needs the possibility of choosing how to react upon the stimulus. The presented solution seems therefore, in its current form, too crude. In [Toetenel-88a], we discussed several extensions to the plain model in some detail. These extensions had in common that their purpose was to limit the sensitivity of a thread of control for remote exceptions. Possibilities were created for tasks to allow - either explicit or implicit - selective listening, i.e. the possibility that in some region of text or span of time a thread of control would be deaf to remote exceptions.

The implementation of the above model turned out to be straight-forward. A prototype implementation - including both the syntactical and the semantical extensions - took only a few days.

4. A second approach to modeling asynchronous transfer of control

Experiments with the extended exception mechanism convinced us that, although perfectly adequate from the actor's point of view, exceptions raised by other threads of control introduced an asynchronicity that is undesirable from the server's point of view. Having the control flow radically changed on unpredictable moments and positions, from elsewhere and not based on the semantics of the instructions executed by the current thread of control, leads to uncontrollable and unpredictable situations. Managing the asynchronous interruption of a thread of control turned out to be a main bottleneck. In spite of all extensions that were proposed and that were aiming at a greater degree of controlability, it was

hard to manage the remote exceptions in such a way that the reaction on the external stimulus remained under control.

A close look at the language extensions and the way these extensions are used to implement mode shifting, shows that the desired sequence of events is:

1. an asynchronous event draws the attention of the server to a certain event, and
2. the event is transformed in a synchronous event as quickly as possible, to allow communication to take place between the server and the actor over the cause of the event, and
3. if possible, the server is allowed to continue at the place where the asynchronous transfer of control was received.

The desired sequence of events models the interrupt model as applied in e.g. PL/I (see for a description e.g. [Ghezzi-87]) very close. This model is known to be dangerous, some limitations in its use are in order. In particular, care must be taken not to allow a transfer of control from the interrupt handling routine to other places.

We propose an extension to the Ada language, based on the interrupt model. These extensions will be formulated as much as possible in terms of the Ada primitives. The possibility is created to interrupt another thread of control, following this interruption communication between the interrupted and the interrupting threads of control may take place using the standard rendezvous constructs provided by the Ada language.

1. At the actor side we introduce the construct:

   raise in *taskname*
   *[***select**
   taskname. error_entry (parameters)
   else
   statements
   end select*];*

   The construction is - with purpose - almost completely expressed in terms of existing Ada language constructs. The asynchronous event is started with the *raise* statement acting as an interrupt statement. Once the receiving thread of control (identified by *taskname*) is interrupted, the receiver is - possibly - made ready to *synchronously* transfer data from and to the interrupting thread of control. At the interrupting side, this synchronous transfer of control syntactically - and to a large extent semantically - takes the form of a conditional entry call. The *else* clause will be executed if the interrupted thread of control has no associated *selective wait* construct. The statements in the *else* part can be used to react on the unwillingness of the server to accept a request for communication.

2. To allow serving, the structure of a potential server task is extended with a *remote exception handler.* Syntactically, the handler appears following the regular exception handlers in a task body[2]. An outline of the syntactical structure of the extensions is

given below:

remote *[[conditional_selective_construct] statements exception_handler]* **end**

In more detail:

```
task body ...
.. declarationss
begin
... statements of the task body
[exception ...]      -- exception handlers
-- the extensions to the task body follow here
-- with extra indent
    [remote
      [select
        accept error_entry do
          user-provided statements
        end;
       or else null;
       end select;
       ]
       ... statements for further interrupt handling
    exception
      when others => null;
    ]
end;
```

The keyword *remote* identifies the handling place in the code of a task body of the interruption of the thread of control. Whenever an interrupting exception is received, the (optional) code following the keyword *remote* is executed after which the thread of control is resumed where the exception was caught. The code in the *remote* body is such that a single entry call may be served after which some statements may be executed. If an entry call is served, care must be taken that no exception raised during the handling of the rendezvous causes a transfer of control in the server after handling the *remote code.* Therefore, an exception handler, catching all kinds of exception, is obligatory. The construct is (more or less) enclosed by a predefined exception handler to ensure that no transfer of control outside the handler takes place. Further processing of data in the interrupted mode of the task may be performed by the statements following the conditional selective wait construct.

During handling of the rendezvous, the actor and the server may communicate data that is important for deciding the cause of the need for this mode shift. Based on this information, the server may alter the values of parameter values such that a controlled switch in operation at the actor side may take place. A certain amount of synchronization between actor and server takes place, the server may e.g. restore the conditions for the normal

2. We associate such an exception handler to a task body only, it is not possible to have an interrupt handler associated with inner blocks or procedures.

mode of control of the actor.

Notice that care is taken to allow for situations in which:

1. the actor does not want to communicate data with the server. Therefore the server side contains a conditional selective wait construct, in those cases where the actor only executes a

   raise in *task_name*

   construct, no locking occurs at the server side.

2. the actor wants to communicate data with the server while the latter does not want to take part in this communication and no selective wait construct is specified at the server side (therefore the actor side contains a conditional entry call).

It will be clear that the syntactic form of the proposed language extensions is rather rigid. The expressiveness of the constructs that are used in the extensions, is less than that of the same constructs in other places, basically because there are no alternative constructs in those places. Syntactical shorthands, using other keywords, are required (and also rather obvious).

5. Rapid mode shifting: an example

In this section we briefly discuss the solution for a problem mentioned in [Toetenel-88a]. In that paper paradigms were derived for expressing solutions to the problem of rapid mode shifting using remote exceptions.

The example comprises an implementation for the problem stated in the next informal and incomplete specification:

> *A real-time system consists of a sensory device which records discrete measurement values, resulting from monitoring an analogous signal. The recorded values may alter very quickly (e.g. within 1 ms). Assumed is a range within the values are "correct", outside this range the values can be considered erroneous or even fatal. The monitored signal must be sampled with a rate as high as possible. Once the signal starts to deflect it is assumed that it cannot recover without repair from some controlling device.*
>
> *The system contains a hierarchical sub-system consisting of mutual autonomous layers, that is processing independently from the sensory device provided that the recorded values are correct. If these values start to deflect, some layer of the sub-system must be informed, on which this layer will shift its current mode of execution to a new mode in which it will try to repair the cause of deflection of the signal. It will use different repair strategies for both erroneous and fatal signals. If the recovery succeeds, it will re-enter its normal mode of execution, otherwise it will alarm a more advanced sub-system followed by re-entering its normal execution mode.*

An encoding of the solution to the above mentioned problem expressed in the Ada programming language using the extensions presented in the previous section, is given below.

We distinguish between three modes of operation, expressed in terms of Ada values as:

type *MODE* **is** *(GREEN, RED, ALARM);*

GREEN indicates the normal mode of operation for the system and its sub-systems;

RED indicates an erroneous value of a signal, the mode of operation executes code to repair the deflection of the signal;

ALARM indicates a fatal value. The mode of operation also has to execute code to adjust the signal source.

The structure of the solution is straight-forward, the task *SAMPLER* periodically samples a sensor device. As soon as the sampler has detected an *erroneous* or a *fatal* result value, an asynchronous activity is initiated in which the *LEVEL_X* task is interrupted to take actions on the erroneous value. Having detected an erroneous or a fatal value, sampling will stop until a repaired value is received, indicating (i) the corrected value, and (ii) the fact that the sampling device is repaired. Synchronous waiting is implemented, taking advantage of the rendezvous being synchronous.

```
p1-1   task body SAMPLER is
p1-2   begin
p1-3     --| initialising the sampling
p1-4     loop
p1-5       RESULT := SAMPLE (...);
p1-6       --| process resulting values
p1-7       if ERRONEOUS_OR_FATAL (RESULT)
p1-8       then
p1-9         --| place interrupt and pass result value
p1-10        raise in LEVEL_X
p1-11        select
p1-12          LEVEL_X. ERROR (RESULT);
p1-13        else null;
p1-14        end select;
p1-15      end if;
p1-16    end loop;
p1-17  end SAMPLER;
```

The outline for the *LEVEL_X* task body is given below. As soon as a remote exception is caught, a (conditional) selective wait construct is executed to obtain the data involved from the sampling task. The code in this repair "mode" dispatches between *erroneous* and *fatal* values and attempts a repair of the value and the sensory device. If no repair is possible, a remote exception is executed to put yet another task into a repair mode.

```
p1-18  task body LEVEL_X is
p1-19  --| appropriate declarations
p1-20  begin
p1-21    loop
p1-22      --| do normal actions
p1-23    end loop;
p1-24  --| exception handlers, if any
p1-25  remote
p1-26    select
p1-27      accept  ERROR (X: in out value_type) do
p1-28        if ERRONEOUS (X)
p1-29        then
p1-30          --| repair the erroneous signal and pass it back
p1-31        elsif FATAL (X)
p1-32        then
p1-33          --| repair the signal and pass it back
p1-34          if no repair is possible
p1-35          then
p1-36            raise in LEVEL_Y
p1-37            select
p1-38              LEVEL_Y. ERROR (X);
p1-39            else null;
p1-40            end select;
p1-41          end if;
p1-42        end if;
p1-43      end
p1-44    or else null;
p1-45    end select;
p1-46  end LEVEL_X;
```

6. Conclusions

In this paper we proposed some language extensions for the Ada programming language to support rapid mode shifting in real-time applications. The extensions have a minor influence on the syntaxis of the Ada programming language, most extensions are expressed in terms of existing language constructs. Semantically, the extensions are also limited; support is provided for interrupting a thread of control with the possibility of returning. We believe that the strength of our model is that the server is itself allowed to decide whether or not to react upon a request for a mode shift and, when expressing its intention, how to react upon the occurrence of an interruption of the thread of control. The extensions have the following properties:

1. it is fairly straightforward to design an implementation strategy such that no overhead is incurred when the extensions are not invoked;

2. as soon as an asynchronous event occurs, a transformation to a synchronous event follows.

3. the semantics are simple and clean.

We must realize that some care must be taken in implementing whatever form of mode shifting. Mode shifting must be seen as a reaction on an exceptional situation, whereby attempts are made to minimize the penalty for appying the construct when it is not used. As such it may be compared with exception handling within a single thread of control.

Currently, we are in the process of implementing these extensions in our Ada— implementation. The syntactical and semantical changes to the compiler part are minor, the tasking kernel needs quite a few changes though.

7. References

[Atkinson-88]
Programming distributed systems in Ada
C. Atkinson
in: IFIP/IFAC Working conference on Hardware and Software for Real-Time Process Control, Warshaw 1988, Preprints Vol I pp 37 - 50.

[ARM-83]
Reference Manual of the Ada Programming language.
U.S. Gouvernment, AJPO 1983.

[Baker-88]
Improving immediacy in Ada
T. Baker
In: [Workshop-88]

[Burns-85]
Concurrent Programming in Ada.
A. Burns,
The Ada Companion Series, Cambridge University Press 1985.

[Gehani-84]
Ada Concurrent Programming,
N. Gehani,
Prentice Hall 1984.

[Ghezzi-87]
Programming language concepts,
C. Ghezzi, M. Jazayeri,
Wiley, second edition 1987.

[Huijsman-87]
Performance Aspects of Ada tasking in embedded systems
R.D. Huijsman, J. van Katwijk, W.J. Toetenel,
Microprocessing and microprogramming 21 (1987) pp 301 - 310.

[Katwijk-87]
The Ada— compiler, On the design and implementation of an Ada compiler,
J. van Katwijk,
PhD Thesis 1987,
Delft University of Technology, Faculty of Mathementics and Computer Science.

[Katwijk-88]
An Ada Tasking implementation,
J. van Katwijk, W.J. Toetenel,
Proceedings of the IFIP-IFAC conference on "Hardware and Software for Real-time process control" 1988 Warshaw. Preprints Vol I pp 15 - 36

[Toetenel-88a]
Asynchronous transfer of control in Ada,
W.J. Toetenel, J. van Katwijk,
In: [Workshop-88]

[Toetenel-88b]
An Ada tasking kernel: annotated listing,
W.J. Toetenel, J. van Katwijk,
Report 88-XX; Reports of the faculty of Mathematics and Computer Science, Faculty of Mathematics and Computer Science, Delft University of Technology, 1988 (To appear 1988).

[Workshop-88]
2nd International Workshop on real-time Ada issues,
Manor House Hotel, Moretonhamstead, Devon England 31 may - june 3 1988.

De tijd van PTERA en ZEBRA.

Prof.Ir.D.H.Wolbers

Sommige boeken worden lange tijd bijzonder intensief gebruikt, zodanig dat kaft en omslag daar de nodige sporen van vertonen. Dat zal niet vaak het geval zijn met proefschriften. Toch prijkt nog steeds in mijn boekenkast een grijs beduimeld boekwerk "The logical principles of some simple computers", dat inderdaad de indicaties heeft van veelvuldig gebruik. Weliswaar ligt dat gebruik nu al geruime tijd achter ons, namelijk de tweede helft van de 50'er jaren. Met de recente ontwikkeling van de microcomputer komt echter de herinnering aan die tijd toch vaak weer terug. Daarbij moet men nu constateren dat "nieuwe ideen" of "programmeertrucs" in feite reeds beschreven waren in dit boekwerk.

Het is dan ook voor mij een voorrecht geweest zowel in gezamelijke PTT tijd als later aan de TH Delft veelvuldig met de schrijver van dat proefschrift te hebben mogen samenwerken in het kader van het gebruik van de twee machines, die in dat boek beschreven staan. De derde daarin aangegeven machine ZERO laat ik nu maar buiten beschouwing, want hoewel theoretisch van belang was voor mij de practische waarde toch iets geringer.

Gaan we eerst nog eens naar de tijd van PTERA, de PTT electronische rekenautomaat. Wim v/d Poel als grondlegger van de constructie van deze machine. Reeds in die tijd bestond de illusie, dat men potentiële gebruikers wel zover zou kunnen brengen, dat deze zelf de benodigde applicatie programma's zouden kunnen ontwerpen. Met enkele zonderlingen is dat inderdaad gelukt en een van die gekken was schrijver dezes. Een sprong was 20, vermenigvuldigen in verschillende uitvoeringen lag in 60 serie, delen overeenkomstig in 70 enz. Instructies moesten in binaire vorm in 5-gaten ponsband worden vastgelegd. Een ingenieus apparaat met 5 piano toetsjes moest dan bediend worden om dit te verzorgen en ook dat werd van de gebruiker verwacht.

In die tijd bestonden mijn werkzaamheden nog voornamelijk uit metingen aan telefoonkabels en daarmee gepaard gaande redelijk ingewikkelde en moeizame berekeningen. Met nomogrammen, een "Curta" handrekenmachine en logarithmentafels werden kabels gekeurd op mogelijk gebruik voor hernieuwd gebruik voor hogere frequenties. De uitvinding van de electronische rekenautomaat leek wel haast de ideale oplossing voor dit vele rekenwerk. Elementaire bewerkingen in de orde van 50 tot 100 msec waren toch waanzinnig snel vergeleken met die handberekeningen.

Weliswaar kostte het de nodige moeite om programma en alle data te proppen in de 1k machine van die tijd. Er was echter een troost want de capaciteit zou verdubbeld worden tot 2k, waarmee alle problemen opgelost zouden zijn. Zover is het echter nooit gekomen want intussen schakelde men over op het volgende geesteskind van v/d Poel, namelijk de nieuwe machine ZEBRA (Zeer Eenvoudige Binaire Reken Automaat), die veel sneller was en bovendien 8 maal zoveel geheugen.

Bepaalde voordelen had de eerste machine PTERA overigens wel. Nog uitgerust met relais kon men aan het rammelen van deze schakelaars horen waar de berekening was en zonodig ook constateren dat er iets mis ging. Tenslotte had iedere subroutine zoals bgtg of wortel zijn eigen cadans.

Als "gebruiker" kon ik toen nog mopperen tegen de "beheerder", dat de machine weer niet in orde was, waarop ik terecht gewezen werd dat mijn programma's dan wel weer fouten zouden bevatten. Hoewel de schaalvergroting in geheugenomvang alsmede de snelheid van de moderne machines als een factor van vele cijfers geschreven moet worden is het verwonderlijk, dat de toenmalige problematiek van gebruiker versus leverancier van computercapaciteit nog nauwelijks veranderd is. Omgekeerd echter ook de samenwerking tussen de twee

partijen. Wanneer ik weer met nieuwe wensen en ideeën kwam voor andere berekeningen werd in gezamenlijk overleg van soms vele uren een oplossing gezocht die nog precies in de machine zou passen. Met het "pianospel" op de toetsen werd door v/d Poel de nieuwste versie snel ingebracht waarna in spanning gewacht werd of het geheel zou werken. Een schatting achteraf mijnerzijds geeft een score van ongeveer 50% dat het goed ging.

Een wat andere situatie deed zich later voor in de ZEBRA tijd, toen we beiden aan de computerkant stonden. Wim nog met de ZEBRA bij PTT en ik met die van de TH Delft. Het was de tijd van de ZEBRA club en het gebruik van de "simple code".

Met bewondering kijk ik dan terug op de soms geniale vondsten op programmeergebied, die door v/d Poel werden tentoongespreid. Door sommigen werd dit ten onrechte wel afgedaan als programmeertrucs, maar in feite werd daarmee de grondslag gelegd van veel softwareontwikkeling, die in latere jaren de kerninformatica gebracht heeft tot het niveau waarop we thans werken.

Van 1958 tot ongeveer 1963 hebben we met deze machine gewerkt en is ook de beduimeling van het genoemde proefschrift ontstaan. Thans lopen we beiden met een programmeerbare zakrekenmachine van enkele honderden guldens met ongeveer vergelijkbare geheugencapaciteit en zelfs nog iets grotere verwerkingssnelheid. Voor diegenen die de hier kort aangegeven begintijd van de computers nog hebben meegemaakt is het duidelijk van welke ontzaggelijke waarde de bijdrage van v/d Poel geweest is in de ontwikkeling van de computers en in het bijzonder in de daarvoor benodigde software. Deze waarde heeft ook terecht internationaal de nodige erkenning gevonden.

Voor de Technische Universiteit Delft was het een voorrecht, dat zoveel jaren de kennis en ervaring van Wim v/d Poel haar ten dienste heeft gestaan en hopelijk nog een aantal jaren in wat andere vorm nog zal voortduren.

Delft, 14 september 1988.

EXTENSIBLE LANGUAGES

Ton Biegstraaten en Frans Ververs

Delft University of Technology
Faculty of Mathematics and Computer Science
Delft, The Netherlands

1. Introduction

In this paper we first give our view on extensible languages, followed by a classification of these languages. In the next two chapters we present two examples of extensible languages in which professor Van der Poel has been involved in the last decennia: LISP and FORTH.

2. Definition and classification

Our first approach in defining the notion of an extensible language resulted in:
A language is *fully extensible* if

1. there exists a mechanism in the language to define new user functional and data abstractions, and control structures.
2. user-defined abstractions are treated in the same sense as system-defined abstractions with respect to syntax and semantics. The only difference may be found in the efficiency of run-time and/or use of storage.

Considering the above definition there are almost no fully extensible languages, with the exception of FORTH and Smalltalk. Therefore we shall relax our definition in such a way that a language is considered extensible, if in some way functional, data abstractions or control structures can be added to the language.
Starting from this definition we can make a classification of (partially) extensible languages:

Languages with functional extension

The notion of functional or procedural extension has been known for long time. All modern languages offer the possibility to define new parameterized functions and/or procedures.

In some languages, like Algol68 ([1]) and Ada ([3]), it is also possible to define your own infix-operators, i.e. extending the built-in set of operators.

Languages with functional and data abstraction extension

Modern programming languages, like Modula-2 ([2]) and Ada, offer the concept of modules resp. packages, necessary for the development of large software systems. The consistency of the interfaces between the modules is checked in compilation-time. Modules can be used both for procedural and data abstraction. In a module you can bundle some strongly related procedures and/or functions, establishing some form of

structuring in the solution, i.e. top-down functional design.
It is also possible to create new datastructures together with the corresponding procedures/functions, to be used as building blocks in a object-oriented design approach.

Languages with data type extension

Very few languages have the possibility to extend previously defined data types. The first language in this direction was Simula ([4]) with the class-concept. Other more recent languages are Smalltalk ([5]), Object Pascal ([6]), and C++ ([7]). The most recent language with type extension is Oberon ([8]), designed by N. Wirth. In Oberon it is possible to define a new record type that relates to an existing record type by some rule of compatability. This feature becomes even more important when you realize that this extended type can be defined in a module importing the original type. In that way user-defined new modules can be seen as extensions of existing modules; they can be developed and changed without recompilation of the modules where the original types are defined.

Languages with control structure extension

Some languages offer the possibility to define new control structures. In CLU ([9]) one can define new iterators. In LISP ([10]) one can define high level language control structures such as an IF and WHILE.

Factors influencing the extensibility

Both the syntactical and semantical complexity of a language have influence on the extensibility. In languages with a rather simple syntactical structure, such as LISP and FORTH ([11]), functional extension is quite simple. There are exceptions, such as Algol68, a language with a complex syntactical structure , which offers also good facilities for extension, due to the orthogonal structure of the language. The same arguments hold for the extension of semantical complex languages.

3. LISP

The well-known language LISP was designed at MIT by John McCarthy in 1960. The most striking points of the language are

1. the equivalence between program and data which allows programs to be interpreted as data, and data to be executed as program;
2. the use of recursion as the main control structure;
3. the use of linked lists as the basis data structure;
4. a regular and simple syntax which is both a virtue and a vice;
5. the interpreter of LISP, written in less than two pages LISP

Points 1, 4 and 5 in particular make LISP easy to extend.

3.1 Extensions in LISP

LISP is easy to extend because it is an interpretive language with a very simple syntax. Therefore it is possible to create new functions and macro's in LISP by which we can change our notation (e.g. more like a ordinary high level language), to introduce new control structures (e.g. IF and WHILE) and new operators (e.g. ^). In this paragraph we give some ways to extend the programming language LISP.

Functional extension

The normal functional extension can be shown by giving an example. We choose the function SUPERREV which reverses the elements of a list and the sublists, e.g. SUPERREV ((A (B (C D) E)(F G) H)) is (H (G F) (E (D C) B) A).

```
EX[1](SUPERREV(X)
      (COND
        ((ATOM X) X)
        ((NULL X) NIL)
        (T(APP(SUPERREV(CDR X))(CONS(SUPERREV(CAR X))NIL)))))
```

SUPERREV uses APP which appends his second argument to the first argument, e.g. APP((A B)(C D)) gives (A B C D).

```
EX(APP(X Y)
      (COND
            ((NULL X) Y)
            (T(CONS(CAR X)(APP(CDR X) Y)))))
```

SUPERREV is defined in terms of APP and the standard LISP functions ATOM, CAR, CDR, COND, CONS and NULL. Note also the recursive way of defining APP and SUPERREV.

Control structure extension

The primitive conditional control structure in LISP is
(COND ((A B)(C D)...(T U)), i.e.

```
IF A<>NIL
 THEN B
 ELSE IF C<>NIL
     THEN D
     ELSE ...
       .
       .
          ELSE U;
```

---

1. Here we use Prof. Van der Poel's favourite short-named functions, defined as follows:
```
DEFINE(((EX(LAMBDA(X Y Z)
      (DEFINE(CONS(CONS X(CONS(CONS LAMBDA
                  (CONS Y (CONS Z NIL)))NIL))NIL))))))
EX(FEX(N B)(DEFLIST(LIST(LIST N(LIST LAMBDA(QUOTE(L A))B)))FEXPR))
```

A construction which is more like an if-statement in a high-level language looks like:
(IF A THEN B ELSE IF C THEN D ELSE U).
It is rather easy to write a function IF that has the same effect as the mentioned COND-construction.

```
FEX(IF(PROG NIL
  LIF ((EVAL(CAR L)A)(RETURN(EVAL(CADDR L)A)))
      ((NOT(EQ(CADDDDR L)(QUOTE IF)))(RETURN(EVAL(CADDDDR L)A)))
      (SETQ L(CDDDDDR L))
      (GO LIF)))
```

Note that a recursive call (IF L) at the end is not possible because the argument is not evaluated.
Now we are able to write the definition of a function , e.g. LENGTH, as:

```
EX(LENGTH(L)(IF (NULL L) THEN 0 ELSE (PLUS 1 (LENGTH(CDR L)))))
```

Extension with operators

We can define very easily new operators in LISP such as ˝ for the inner-product of two vectors.

```
DEFLIST(((˝ IP))OPERATOR)

EX(IP(X Y)(COND
      ((NULL X) NIL)
      ((NULL Y) NIL)
      (T(CONS(TIMES(CAR X)(CAR Y))(IP (CDR X)(CDR Y))))))
```

The functions TR and TRANS make it possible to use the notation (X ˝ Y), i.e. as if ˝ were an infix-operator. Actually TR translates an infix-expression into normal LISP.

```
EX(TR(L)(PROG(K)
      ((ATOM L)(RETURN L))
      ((NULL(CDR L))(RETURN(CONS(TR(CAR L))(TR(CDR L)))))
      ((NOT(NUMBERP(CADR L)))(SETQ K(GET(CADR L)(QUOTE OPERATOR))))
      (COND(K(RETURN(CONS K (CONS(TR(CAR L))(TR(CDDR L))))))
      (T(RETURN(CONS(TR(CAR L))(TR(CDR L))))))))

EX(TRANS(N F B)(DEFINE(LIST(LIST N(LIST LAMBDA F(TR B))))))
```

Now we can use (X ˝ Y) in e.g.

```
TRANS(PERPTST(X Y)
      (IF ((X ˝ Y) = 0)
            THEN (PRINT '(X AND Y ARE PERPENDICULAR))
            ELSE (PRINT '(X AND Y ARE NOT PERPENDICULAR))))
```

4. FORTH

4.1 Description of FORTH

Another extensible language is FORTH. It was designed by Charles Moore around 1970 to control a radio telescope and was first described in [11]. Allthough FORTH is especially suited for control applications, it is certainly not limited to that area.
It is very different from any other language in common use, because:

- It basically consists of a *dictionary* of *words* and two stacks, one *datastack* and one *system stack*, usually called the *return stack*. The *return stack* however is not so well hidden, that it can't be used by the user.
- It is a postfix language. Usually data is put on the stack first. A word out of the dictionary can now be executed and can act upon the data on the stack. As a result some output may be written, or data is returned to the stack. After execution of one word another may follow and so on, until the problem at hand is solved.
- Words can be executed, when typed from the terminal or from inside other words.
- Most words are composed of other words. There are only about 40 primitives. These are developed in machine language.
- A special word ":", a colon, is used to extend the language with new words. New words can serve all kind of purposes, such as simple arithmetic combinations. Also complicated control structures as "case" constructs are possible. The word after the colon is the name of the new word. The words that follow the name are compiled in to the new word, until a ";" is reached. For example:
  Make a word "plusprint" that adds the two top most words of the datastack and prints the result on the screen. The definition is:

  ```
  : plusprint + . ;
  ```

  The "+" adds the two top most words of the datastack and returns the result on the datastack. The "." removes the top most word from the datastack and prints it on the screen.
- Words like ":", that make other words, can also be made, using the words "<builds" and "does>". This is especially suited for developing new datastructures. With these words, for example, a new word "array" can be made. It is a recipe how to build an array. "array" "array-name" will make an array with name "array-name".
  The interested reader can refer to [13] and [14].

4.2 Advantages of FORTH

- FORTH is fully extensible. Datatypes as well as control structures can easily be added.
- Words are compiled as indirect threaded code, so space requirements are very small. A typical system within 16kB contains a compiler, an assembler, a simple editor and a virtual memory system. Still there is enough room for a modest application.

- Short development time for applications ([12]).
- The nature of the language stimulates abstract data types and top down development of software.
- It gives a very convenient interactive interface to the underlying hardware, which is very useful for testing.
- With initial development in FORTH, speeding up execution is still possible by recoding frequently used words in assembler.

4.3 Disadvantages of FORTH

- Because of the postfix notation, programs are diffult to read and to trace during execution.
- With about 200 basic words, it is often difficult to follow the internals of a program.
- Words like "+" usually take 16 bit quantities of the stack. 32 bit addition needs another word, so there is no overloading of operators.
- Number crunching is rather slow.

4.4 High Level Forth (HiLF)

The question rises if the extensibility of FORTH can be used to circumvent some of its disadvantages. As part of a masters thesis, C. van der Hoek ([15]) developed a compiler based on FORTH for a very simple language with a pascal like structure.
As an example the next program will initialize two integers, calls a function, which adds its arguments and returns the result. The result is printed.

```
'int I, J ;
'proc ADDELEM('int  I1 ; 'int I2 ; 'int I3) ;
'int TOTAAL ;
'begin
  TOTAAL := I1 + I2 + I3 ;
  'return ( TOTAAL )
'end ;
'main VB
'begin
  I := 1 ;
  J := 2 ;
  PRINT("totaal  is ", ADDELEM(I,  J, 3))
'end ;
```

The resulting code is:

```
0 variable I 0 variable J
: ADDELEM [ smudge ] pushrecord 8 allot
 4 get ! 2 get ! 0 get !
 6 get
 0 get @ 2 get @ + 4 get @ +
 swap !
 6 get @
 poprecord ; smudge

: VB I 1 swap !
 J 2 swap !
 ." totaal is " I @ J @ 3 ADDELEM . ;
```

4.4.1 Comments on the approach A language like HiLF removes some of the disadvantages of FORTH, these are:

- HiLF knows types, so the operations "+" and "d+" can be overloaded and the user must only specify the type of the variables to get the correct behaviour of addition in this example. At the moment only integers and characters are inplemented and characters are implemented as integers, so the mechanism cannot be shown.
- HiLF uses infix notation. This makes the program more readable.

However HiLF is not FORTH:

- it is not extensible, other than by changing the compiler
- HiLF is also not as expressive as FORTH, so only a small subset of the problems that can be solved in FORTH can be solved in HiLF.
- allthough compiled HiLF programs can be used as FORTH words, it is not generally possible to use FORTH words inside a HiLF program. One reason is the location of the activation record. It is in the dictionary and it is assumed that it is the latest added word. So words that change the dictionary can give problems.

4.5 A more extensible approach to extending FORTH

A statement in HiLF is scanned and parsed by a separate program, written in FORTH. The resulting FORTH code is generated directly, so the compiler is one-pass.
In FORTH another mechanism should be possible.
A token is a sequence of characters, just as a word. So it should be possible to convert tokens to words, execute them and interprete or generate code. Necessary is a token scanner, the FORTH scanner only sees a space as word/token separator and that is not sufficient for a language like HiLF.
Also the words must be more "intelligent". The explicit context of a word should be available to that word, and code should be available to decide if the word is applicable in that context.
Every token must be available before it is used, so the resulting compiler is also one pass.
To get the flavour of this approach a simple expression parser is explained. The syntax is

(informally):

```
prog        : declarations statements delimiter
            ;

declaration : int identifier delimiter
            ;

statement   : identifier := expression delimiter
            | print identifier delimiter
            ;
```

`print, int, := ,delimiter` are tokens
The non-terminal "expression" behaves as expected with all precedence rules.

From the terminal the program is interpreted, within a colon definition it is compiled.
To avoid incomprehensible FORTH, the proposed solution is described in words.

Variable definition and use

For each variable enough room is reserved to store type information and the actual value. When the variable is executed (as a FORTH word) a check is made to put the address or the value on the stack. Also the current expression type is matched against the type of the variable. Because strong typing is assumed, an error results for mismatch of types.

Operator definition and use

Each infix operator has its precedence level and a pointer to a table of FORTH operators for each defined type. When the operator has equal or lower precedence than the current level, the previous operator is popped from the stack, its table is searched for the FORTH operator of the current type and the operation is performed. When the precedence is higher nothing happens. In both cases the new operator is pushed on the stack.
Syntax checking is not implemented.

Comments on this approach

- Syntax checking is very difficult when the language becomes more complex. There is no controling parser, so every word must decide from the context state if it is allowed to execute. A thorough analysis of the language syntax is necessary. An intermediate solution could be a scanner/parser that only checks the allowance of a token but not its activities. It is only important for the parser for example if an identifier is a variable, its type does not matter.
- Because the type of an expression is only contained in the operators and variables, introducing another type is not that hard. The type tables must be extended, and a new definition of a variable with such a type must be available. All changes that do not necessitate changes in the parser are possible extensions. In that way this proposed language is not different from FORTH itself, only the FORTH syntax and so the parser is very simple.

So extensibility has its limits and also its price when those limits are pushed. And because it is also not clear if extensibility is a blessing or a curse (the number of FORTH dialects are uncountable), the usefullness of an extensible high level language is at best a point of discussion.
Maybe a more limited goal for extending FORTH is better. Change postfix in infix, keep track of types, accept only strong typing and let operators work with all types.

References

[1] Lindsey, C.H. and Meulen, S.G. van der, Informal introduction to Algol68, North-Holland Publ. Co., 1971.

[2] Pomberger, G., Software Engineering and Modula-2, Prentice-Hall Int., 1986.

[3] Booch, G., Software Components with Ada, The Benjamin/Cummings Publ. Co., 1987.

[4] Birtwistle, G. et al., Simula Begin. Auerbach, 1973.

[5] Goldberg, A., Robson, D., Smalltalk-80: the language and its implementation. Addison-Wesley, 1983.

[6] Tesler, L., Object Pascal Report, Structured Language World, 9(3), pp. 10-14, 1985.

[7] Stroustrup, B., The Programming Language C++, Addison-Wesley, 1986.

[8] Wirth, N., The programming language Oberon. Software Practice and Experience, Vol. 18, Nr. 7, pp. 671-691, July 1988.

[9] Liskov, B. et al., CLU Reference Manual, New York, Springer Verlag, 1981.

[10] Poel, W.L. van der, The Programming languages LISP and TRAC, Delft, 1972.

[11] Moore, C.H. and Rather, E.D., The FORTH program for spectral line observing. Proc. IEEE, Vol. 61, pp. 1346-1349, September 1973.

[12] Moore, C.H., and Rather, E.D., The FORTH approach to operating systems. ACM '76 Proceedings pp. 233-240, October 1976.

[13] Brodie, L., Starting Forth. Prentice Hall, New York (1981).

[14] Brodie, L., Thinking Forth. Prentice Hall, New York (1984).

[15] Hoek, C. van der, High Level Forth. Masters Thesis, Department of Mathematics and Informatics, Technical University Delft, 1981.

For Wim van der Poel

D. J. McConalogue

I am very happy to join my colleagues in paying a personal tribute to Wim van der Poel on the occasion of his retirement from Delft Technical University.

This is an opportunity to give vent to our personal appreciation of Wim, rather than to expand on his significant contributions to informatics over a period of forty years. This has already been repeatedly recognised internationally.

My personal working contact with Wim extends over the last nine years. However, I first became aware of his existence when I entered the field of computers in 1957. Those were the days of mercury delay lines, valves and assembly language, and the terms Computer Science and Informatics were not even struggling to be born. There were about three commercially available computers of which I now remember only one, the STANTEC ZEBRA. The story went that it had been designed by a Dutchman called van der Poel who could not find a company in Holland to build it, so he had to come to the British company, Standard Telephones and Cables, to have it built.

The name stuck in my memory - it was short and easy to remember - and was refreshed by appearing regulary in the literature. I first met Wim personally at one of the annual seminars at Newcastle upon Tyne around 1976. When the post in Technical Applications in Informatics was brought to my attention, I telephoned Wim, as a result of which I am now in Delft.

After these nine years, which have been uniformly pleasant on my side, I have developed a clear picture of Wim. Academics have been divided into two classes: those who live *for* their subject, and those who live *on* it. Most of us lie somewhere between. Wim unambiguously belongs to the first class.

What makes it pleasant to be his colleague is his personal generosity, both emotional and intellectual, which results in his ability to recognize the value of other peoples work, and his complete freedom from meanness in word and deed. If I may be excused one piece of 'pop' psychological analysis, I would say that Wim's "child ego" is very much alive. According to the Transactional Analysis Model, this manifests itself in characteristics such as intellectual curiosity, openness to new ideas, the playfulness that is akin to creativity and unselfconscious enjoyment in what one is doing. With most people, these decrease with increasing age. Their survival in Wim have given him an unparalleled breadth in informatics rangeing from being a formidable expert in the lambda calculus to being an equally accomplished screwdriver-wielding "knutselaar".

Proverbially, no man is an island, and Wim owes a great deal more to Annie than his happy status as father and grandfather. I am also glad to have had the opportunity to get to know and appreciate Annie in her own right.

I may appear to have been pleading his case for Sainthood - a position to which he has shown no visible aspirations - so perhaps I should put in *one* word for the Devils Advocate: I learned early that Wim is not one of the world's most enthusiastic administrators. I cast no stone.

He has now formally retired, but we will not see the end of him for some time yet. Fortunately!

Denis McConalogue

Operating Systems, Applications, and Programming Languages for Distributed Computer Architectures

Henri E. Bal

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

1. INTRODUCTION

Computer scientists spend a substantial part of their time in useless waiting. The two most common cases are: waiting at an airport for a delayed flight to the next conference, and waiting behind a terminal for a compilation to be finished. The first problem is caused by overloaded airports; researchers probably just have to live with it, although careful selection of airports and carriers may somewhat reduce the burden. The second problem is due to overloaded, time-shared computers. In the near future there is hope for solving this problem, as computers are rapidly getting faster and cheaper.

Using a faster computer means it takes more jobs to overload the system. As researchers are highly inventive, they usually don't have any difficulties in doing so. The fall in hardware prices, on the other hand, makes it attractive to spend the computer budget on many *personal computers* (or *workstations*), rather than on a single time-shared machine. Due to a better price/performance ratio, these workstations together typically give a much higher computing capacity than a uniprocessor system. Moreover, each user has a certain amount of computing capacity available all day, even during office hours.

The next logical step is to let the computers outnumber the users. Besides having one machine per user, there is a *pool* of extra processors that can be divided among the users as needed. All these processors are managed by a *distributed operating system* [11], making them look like a single integrated computer system, not much different from a centralized system.

The presence of multiple processors allows several jobs of the same user to be executed in parallel. Additionally, a single job may be split up into many smaller, independent jobs that can be run concurrently. To compile a program consisting of five modules, for example, each module can be compiled on a separate processor, thus speeding up the whole job. Individual programs may be further sped up using *parallel algorithms*. Many applications are suitable for parallel execution on a distributed system. To implement these applications, programming support for process creation and interprocess communication is needed. This support can be provided by a distributed operating system or by a language designed for distributed programming.

In the next three sections we discuss how operating systems, application programs, and programming languages are affected by the evolution from centralized to distributed systems. We have built a prototype distributed computing system, *Amoeba*†, which we are using to implement programming languages and applications. We use the Amoeba system to illustrate our ideas.

This research was supported in part by the Netherlands Organization for Scientific Research (N.W.O.) under grant 125-30-10.

†Amoeba is a joined project of the Vrije Universiteit and the Centrum voor Wiskunde en Informatica, both in Amsterdam. Its principal designers are Andy Tanenbaum, Sape Mullender, and Robbert van Renesse.

2. DISTRIBUTED OPERATING SYSTEMS

Before the wide-scale introduction of VLSI technology, computers were highly expensive machines. Operating system designers aimed at optimal utilization of the central processor, using *multi-programming* and *time-sharing*. These techniques have not only been applied to mainframes, but also to less expensive mini-computers. Even for the PDP-8 a time-sharing operating system exists (Multi-8), although this machine is less powerful than today's micro-computers that nobody would want to share. For the PDP-11 several popular time-sharing operating systems are available (e.g., UNIX*).

The declining prices of computer hardware make time-sharing less and less attractive and stimulate the development of personal computers. Early operating systems for personal computers completely did away with multi-tasking and allowed only one program at a time to execute. Later systems allow multiple programs to run quasi-concurrently.

Personal computers are a step forward, as each user has a constant number of cycles available. Unfortunately, they also introduce new problems. First of all, having multiple workstations with local file storage complicates sharing of file-systems and other resources. Second, each user has only a limited amount of processing power available. This capacity can only be increased by upgrading the workstation. *Network operating systems* therefore support remote file access and remote execution facilities. Still, the user has to determine where a file is located and to decide where to execute programs. Moreover, executing programs on other people's workstations does not help when all workstations are heavily used, so this leads to similar problems as for centralized systems.

To solve the above problems, we have designed a new architecture for a distributed computing system (see Figure 1).

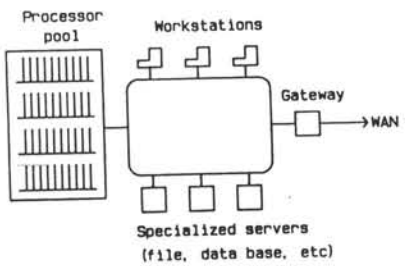


Fig. 1: The Amoeba architecture

The architecture contains four types of components. Each user has a personal workstation, containing a processor, a few megabytes of memory, and a bitmap display. The workstations are used to run interactive programs that require fast response, like window managers, command interpreters, and text editors.

Compute-intensive jobs are run on processors of the *processor pool*. These processors are allocated dynamically to the users. A user may ask for multiple processors, for example to run a parallel program. The computing power of the system can easily be extended by adding more pool processors.

* UNIX is a registered trademark of AT&T Bell Labs

The architecture also contains processors specialized (either by hardware or software) for a specific job. Examples are: file servers, disk servers, directory servers, and accounting servers. Finally, there are *gateways* that connect different sites through a wide-area network. All processors of a single site are connected by a local-area network. Some of the pool processors also share a local bus.

The distributed operating system we have designed for this architecture is called *Amoeba* [8]. The Amoeba kernel runs on every processor and supports little else but *interprocess communication*. Other operating system services are implemented outside the kernel, in specialized servers. A single primitive is provided for communication between all types of machines (workstations, pool processors, servers, and gateways). This primitive, the *transaction*, is similar to a *remote procedure call*. It offers reliable, synchronous communication between two processes.

Amoeba is based on the object-model. Objects (e.g., files, directories, processes) are owned by *server* processes; *clients* perform operations on objects by doing transactions with the servers. Servers are addressed indirectly through port-names, in a location-independent way. Amoeba uses sparse capabilities to protect objects [12].

At present, we have a working prototype of the Amoeba system. We use two different processor pools, one containing 16 MC68020s connected by an Ethernet and another one consisting of 12 MC68020s on a VME-bus. Much attention has been paid to optimizing transactions. A small transaction between two MC68020s over an Ethernet takes 1.4 milliseconds, faster than most other similar systems. Other interesting research areas are wide-area connection of different Amoeba sites [9] and replicated directory servers [10].

3. DISTRIBUTED APPLICATIONS

The pool processors in Amoeba are primarily intended to offload workstations. Applications that would be run in the background in a centralized system (as a ''batch'' job) are executed by a pool processor instead. A single user may run multiple jobs on different machines simultaneously. Simulation programs, for example, usually have to be executed with many different parameter settings. All these runs can be done in parallel, giving a significant decrease in turnaround time.

Parallelism is also the key to speeding up individual jobs. Applications that can be split up into independent, large-grain pieces are best suited for a distributed system. Unfortunately, only few applications fall in this category. Others can be distributed using a parallel algorithm, executed by several cooperating processors. Below, we discuss two such algorithms: parallel branch-and-bound and parallel alpha-beta search.

Branch-and-bound algorithms use a *tree* to structure the search space. Parallelism is obtained by searching parts of the tree in parallel. An important issue is how to distribute the tree among the participating processors. At first thought, one may try to split up the tree in equal pieces, one piece for every processor (as shown in Figure 2 for a 4-city Traveling Salesman Problem). This approach minimizes communication overhead, but it also has a severe disadvantage. If a branch-and-bound algorithm finds a (not necessarily optimal) solution, it uses this solution to cut off (prune) other search paths. In the Traveling Salesman Problem, for example, once a full route for the salesman has been found, partial routes that are longer than this full route cannot lead to an optimal solution, so they need not be searched. When a sequential depth-first TSP algorithm visits a node N, it already knows the best solution contained in the part of the tree to the left of N, so it may use this value to cut off node N. In a parallel

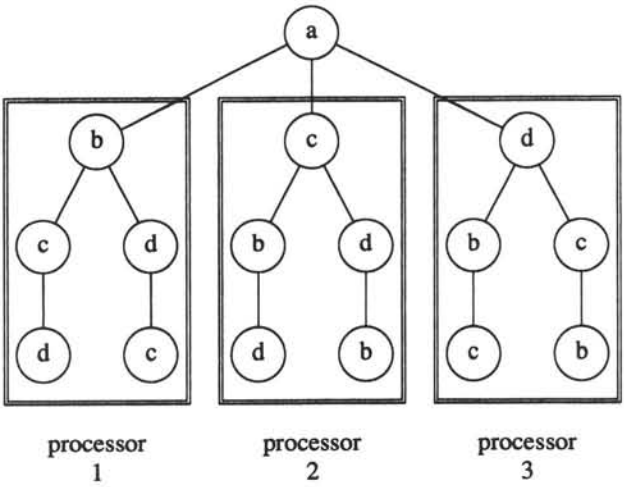


Fig. 2: Distribution of a 4-city TSP search tree

algorithm, the nodes to the left of N need not have been searched yet, so the available bound may be worse than the one used in the sequential version. In other words, a parallel search algorithm may visit more nodes (i.e., try more alternatives) than a sequential algorithm, causing the speedup to be less than linear.

In general, better ways of distributing the search tree are needed. In the algorithm we use, one *manager* process searches the top part of the tree, up to a certain depth D. It then hands out the subtree below that level to an *agent* process. The number of levels to be searched by the manager is a tunable constant. If D is high, the manager generates many small subtrees; if D is low, it generates a few large subtrees (as in Figure 2). In the first case, the total number of nodes visited by the algorithm tends to be low, but the communication overhead is high. A reasonable speedup (4.4 with 6 agent processors) can be obtained by carefully tuning the depth [4].

We have implemented a distributed TSP branch-and-bound algorithm in C, using library routines to access the Amoeba primitives. The lack of shared memory significantly complicates the implementation. The bound used by the parallel branch-and-bound algorithm is essentially a global variable shared by all agents. We simulated the shared variable by keeping a copy of it in the local memory of each agent. The copy is updated whenever the agent receives a new subtree to evaluate.

Alpha-beta search is an efficient method for searching game trees for two-person, zero-sum games. It is used in most chess programs. Like branch-and-bound, alpha-beta is a tree search algorithm that tries to cut off as many branches as possible. Alpha-beta search differs significantly from branch-and-bound, however, in the way the best solution is constructed. A branch-and-bound program (potentially) updates its solution every time a processor visits a leaf node. The processor only needs to know the current best solution and the value associated with the leaf. An alpha-beta program, on the other hand, has to *combine* the values of the leaves and the interior nodes, using the structure of the tree. Some parallel alpha-beta programs realize this by having a dedicated processor for every node (up to a certain level) that collects the results of the child processors. As a disadvantage of this method, processors associated with high level interior nodes spend most of their time waiting for their children to finish.

Our solution [4] avoids this problem by working the other way round. The child processors compute the values for their parent nodes, so there is no need for their parent processors to wait. To do this, an *explicit* tree structure is built, containing the alpha and beta bounds at each node. The search tree is no longer just a concept, but it is actually built as a data structure in memory. This tree is distributed among the processors, each processor containing that part of the tree it is working on.

The speedup achieved for alpha-beta is significantly worse than for branch-and-bound. The main reason is that alpha-beta suffers even more from the decrease in pruning efficiency than TSP. Many improvements have been suggested for parallel alpha-beta search [5], but today's algorithms are still not capable of efficiently exploiting massive parallelism.

Besides TSP and alpha-beta search, we also are working on other distributed applications. We have implemented a parallel version of the UNIX tool Make. *Parallel Make* forks off command blocks in parallel, thus significantly decreasing the time to compile large programs [1]. Another interesting application that has been ported to Amoeba is Finkel and Manber's Distributed Backtracking Package [6]. Finally, we are working on a parallel chess program based on conspiracy numbers rather than alpha-beta.

4. PROGRAMMING LANGUAGES FOR DISTRIBUTED SYSTEMS

Distributed applications can be implemented directly on top of a distributed operating system, using an existing sequential language. The operating system primitives can be accessed through library routines. This style of distributed programming, however, is low-level and much harder than sequential programming. Languages designed for programming distributed systems may present a higher level of abstraction, making the programmer's job easier.

A distributed system has some unique properties that must be dealt with by such a language. First of all, the language should be based on a *parallel* computation model. Operating systems usually support parallel processes, but a variety of language constructs for expressing parallelism exists. Second, the parallel units (whatever they may be) must be able to *communicate* and *synchronize*. Processors in distributed systems usually do not share memory, so the communication and synchronization primitives have to be implemented with message passing. Finally, distributed systems have the *partial failure* property: a failure in one processor will not bring the entire system down. This property enables the construction of fault-tolerant programs, which can survive processor crashes. Some languages (e.g., Argus) let the programmer specify what should be done after a processor crash. Ideally, the language implementation should deal with processor failures. (This approach is taken, for example, by the language NIL.)

It is hard to estimate how many languages for distributed programming exist, as it depends on what counts as a language and what qualifies as "distributed." A recent survey [2] indicates there are at least 100 relevant languages. Many of these extend a procedural language with parallel processes and some form of message passing (e.g., synchronous or asynchronous message passing, remote procedure call, rendez-vous). The basis for this class of languages is Hoare's classic paper on CSP [7]. Other well-known languages are SR, StarMod, Concurrent C, PLITS, Ada*, Distributed Processes, and NIL [2].

Recent research has resulted in higher-level models of parallelism and communication. Concurrent object-oriented languages extend the object model by allowing multiple objects to be active simultaneously. Example languages are Emerald, ABCL/1, and Concurrent Smalltalk.

* Ada is a registered trademark of the U.S. Dept. of Defense, Ada Joint Program Office

Concurrent logic languages are based on the parallelism inherent in theorem proving. Most of them use the *shared logical variable* for communication. The most influential languages in this category are Concurrent Prolog, PARLOG, Delta Prolog, P-Prolog, and GHC.

The absence of shared memory makes distributed programming potentially harder than parallel programming. Several recent languages reduce this gap by providing a programming model based on *abstract shared data*, without requiring physical shared memory [3]. Languages like Concurrent Prolog, for example, can be implemented with or without physical shared memory, even though they are based on communication through shared (logical) variables. Linda supports *distributed data structures*, which can be manipulated simultaneously by different processes, Linda implements distributed data structures in *Tuple Space*, a global, content-addressable, abstract shared memory.

We are working on a high-level language for distributed applications programming [3]. This language, called Orca, provides communication through shared data-objects, which are instances of abstract data types. An object in Orca is a passive entity that can be shared among several processes. Each process can apply operations to the object, as defined by the object's abstract data type. All operations are indivisible, thus providing automatic mutual exclusion. Condition synchronization is expressed through operations that block.

Orca can be implemented on a multiprocessor by putting shared objects in the shared memory and protecting them by lock variables. In a distributed implementation, objects are dynamically migrated and replicated, depending on how often they are read and written by each processor. For this purpose, the run-time system dynamically maintains statistics on the usage of objects [3].

5. CONCLUSIONS

The falling prices and increasing speed of modern computers stimulate the evolution from centralized to distributed systems. Today, distributed systems are a major research area. There are several regular conferences and journals on both the theoretical and practical aspects of distributed systems and distributed computing. Many distributed operating systems are operational, although only few are suitable for daily use as a production system yet.

Electronic mail has long been the only useful application that has actually been implemented in a distributed way. Fortunately, the tide is changing and new applications are emerging that can benefit from the parallel computing power of distributed systems Applications described in the literature include matrix multiplication, heuristic search, sorting, computer chess, compilation, numerical analysis, VLSI design, simulation, and many others.

Efficiently implementing distributed applications is harder than writing sequential programs, because of issues like parallelism, communication, synchronization, and fault-tolerance. Ideally, the compiler and run time system should take care of these issues. Although some achievements have been made in this area, the current state-of-the-art in compiler technology is still far from this ideal situation. Most efforts to ease distributed programming aim at providing high-level language support [2].

ACKNOWLEDGEMENTS

The author would like to thank Jennifer Steiner, Andy Tanenbaum, and Frans Kaashoek for giving useful comments on this paper.

REFERENCES

1. Baalbergen, E.H., "Design and Implementation of Parallel Make," *Computing Systems - The Journal of the USENIX Association* **1**(2) (Spring 1988).
2. Bal, H.E., Steiner, J.G., and Tanenbaum, A.S., "Programming Languages for Distributed Systems," IR-147, Vrije Universiteit, Amsterdam, The Netherlands (Feb. 1988).
3. Bal, H.E. and Tanenbaum, A.S., "Distributed Programming with Shared Data," *Proc. IEEE CS 1988 Int. Conf. on Computer Languages*, Miami, Fl. (Oct. 1988).
4. Bal, H.E., Renesse, R. van, and Tanenbaum, A.S., "Implementing Distributed Algorithms Using Remote Procedure Calls," *Proc. AFIPS Nat. Computer Conf.*, Chicago, Ill. **56**, pp. 499-506, AFIPS Press (June 1987).
5. Bal, H.E. and Renesse, R. van, "A Summary of Parallel Alpha-Beta Search Results," *ICCA Journal* **9**(3), pp. 146-149 (Sept. 1986).
6. Finkel, R. and Manber, U., "DIB - A Distributed Implementation of Backtracking," *ACM Trans. Program. Lang. Syst.* **9**(2), pp. 235-256 (April 1987).
7. Hoare, C.A.R., "Communicating Sequential Processes," *Commun. ACM* **21**(8), pp. 666-677 (August 1978).
8. Mullender, S.J. and Tanenbaum, A.S., "Design of a Capability-Based Distributed Operating System," *Computer J.* **29**(4), pp. 289-299 (August 1986).
9. Renesse, R. van, Tanenbaum, A.S., Staveren, J.M. van, and Hall, J., "Connecting RPC-Based Distributed Systems Using Wide-Area Networks," *Proc. of the 7th Int. Conf. on Distributed Computing Systems*, Berlin, pp. 28-34 (Sept. 1987).
10. Renesse, R. van and Tanenbaum, A. S., "Voting with Ghosts," *Proc. of the 8th Int. Conf. on Distributed Computing Systems*, San Jose, CA, pp. 456-462 (June 1988).
11. Tanenbaum, A.S. and Renesse, R. van, "Distributed Operating Systems," *Computing Surveys* **17**(4), pp. 419-470 (Dec. 1985).
12. Tanenbaum, A.S., Mullender, S.J., and Renesse, R. van, "Using Sparse Capabilities in a Distributed Operating System," *Proc. of the 6th Int. Conf. on Distributed Computing Systems*, Cambridge, MA, pp. 558-563 (May 1986).

In plaats van een afscheidstoespraak

Dr. I. S. Herschberg

Het past me, op de keper beschouwd, nauwelijks hier herinneringen van je op te halen, omdat ik van je carrière aan onze Technische Universiteit alleen maar de laatste tien jaar heb meegemaakt, terwijl de gids van onze instelling in een unieke voetnoot vertelt dat je al in 1962 bijzonder hoogleraar was. Toch vind ik de moed in dit *Festschrift* wat herinneringen vast te leggen, niet alleen uit onze gemeenschappelijke Delftse tijd, maar ook uit eerder dagen, toen onze paden elkaar toch al gekruist hadden.

Je behoort, zoals we allen weten, tot de informatici van het eerste uur, strikt genomen zelfs tot die van een uur met een negatief rangnummer, zeg het min 20ste, om de eenvoudige reden dat het begrip *informaticus* nog helemaal niet in zwang was. Ik weet niet hoe de jonge hoogleraar in de zuivere en toegepaste wiskunde (met de mechanica er nog bij) zichzelf toen aangeduid heeft. Wel weet ik dat velen van ons zichzelf betitelden als *programmeur* en die titel niet zonder trots droegen. Programmeur was, wil ik in herinnering roepen, indertijd geen scheldnaam voor volk van minder allooi, voor jonge mensen, die de kinderschoenen van dat allerverachtelijkste codeurschap waren ontgroeid en hoopten op weg te zijn naar de hogere status van systeem-analist of zelfs systeem-ontwerper met het salaris en de emolumenten die daar tegenwoordig bij schijnen te horen. Integendeel, programmeurschap was meesterschap.

Veel van dat meesterschap was uit nood geboren, dat zij toegegeven. Op machines, waarvan het werkgeheugen op zijn hoogst in weinig duizenden bytes gemeten werd - alleen het begrip byte en de onheuse organisatievorm daarvan kenden we gelukkig nog niet - was elke bespaarde opdracht er weer ééntje. Niet dat het alleen om de ruimtewinst ging, maar er zat in die bezuinigingswoede wel degelijk nog het element van sportiviteit en, zeker in jouw geval, van elegantie. Van elk stukje code werd tot op het bot nagepluisd of er niet één opdracht uit weggepraat kon worden. Het was onze gevestigde, maar absurde mening dat dat altijd wel zou lukken. De absurditeit van de bewering blijkt uit het feit dat het, recursief gezien, niet toepasbaar is: anders was alle code tot nul opdrachten herleidbaar en dat geloofden zelfs wij niet. De elegantie lag erin dat de programmeur tot onverwachte denkkronkels kwam waaraan hij dikwijls zoiets als een bewijs kon ophangen. Om een voorbeeldje te geven: één van je pupillen had de opdracht om voor de X-1 een subroutine te schrijven die de derdemachtswortel uit een breuk zou moeten berekenen. Hij construeert eerst een uitgangsschatting die al zo goed is dat hij kan bewijzen dat precies twee Newton-slagen altijd voldoende zijn voor de gewenste bit-nauwkeurigheid. Tot dusver, zal men zeggen, niets bijzonders. De nood van de eerste uren dook er pas in toen deze jongeman twee constanten nodig had voor zijn lineaire beginschatting. Wat een schandelijke verspilling, vond hij, met die twee constanten zijn programma uit te breiden. Daarop moest toch wel een uitweg te vinden zijn. Die was er ook, want zoals elke numericus wel aanvoelt, zit er een zekere rek in die constanten: ze hoeven niet precies de waarden a en b te hebben; er is een zeker bereik van geldige a's en b's, en bovendien geldt dat, als a een beetje hoog uitvalt, je dat nog wel kunt compenseren door een wat kleinere b

en omgekeerd. Wat doet dus de man? Hij zoekt in het operating system, waarvan de opdrachten mits beschouwd als echte breuken, in het juiste bereik liggen voor *a* en *b* en vindt ze ook nog. Voor wijziging van die twee opdrachten alias constanten is alle vrees ongegrond: waren zij nog niet in een vroege vorm van ready-only geheugen vast en onwijzigbaar verdraad en in die idyllische tijd voor alle X-1's dezelfde? Zo gezegd, zo gedaan. Helaas, er bleek ook dat het zo fijnzinnig uitgekiende algorithme niet zou werken als de radicand precies nul was... Alweer geen nood: de eerste opdracht bestaat mede uit een toets op het nul zijn; alle zowat dertig vervolgopdrachten worden alleen uitgevoerd als de nultoets faalt. Met een ingangswaarde van nul geeft de hele subroutine dan ook geen aanleiding tot het uitvoeren van welke echte opdracht dan ook en het antwoord is kennelijk het juiste antwoord: nul.

Hoewel dit voorbeeld niet van jezelf afkomstig is, is het geheel in je geest. Programmeren is bitspitten en daarbij ervoor zorgdragen dat geen bit beschadigd wordt. In hoeverre het bit ook toen al je liefde was, kan blijken uit nog eerder tijd, de dagen van het ontwerp van je ZEBRA. Had in die machine niet elk bit van een opdracht al zijn eigen functie? Je dissertatie, op het ogenblik, schat ik, al een even grote rariteit als die van Edsger W. Dijkstra, beschrijft het allemaal met de nodige nauwkeurigheid. Korte tijd later, toen een serie-exemplaar van de ZEBRA al bij de Nederlandse Marine in gebruik was, ondekte de eerste Delftse wiskundige ingenieur tijdens zijn diensttijd dat je op deze machine ook kon *onderwater programmeren.* Deze vakterm betekende dat je - o dagen van von Neumann - je machineopdrachten zelf in een register kon bewerken om ze, in gewijzigde vorm, naar het geheugen terug te planten en ze later te laten uitvoeren. We wisten in die pre-informatica dagen gelukkig nog niet dat het onderwater programmeren, om begrijpelijke redenen bij de Marine gretig toegepast, later uitgeroepen zou worden als erger dan de zonde tegen de Heilige Geest en nog veel later plechtig verklaard werd tot een van de hoofdoorzaken waardoor onze machines zo akelig langzaam zijn. Gelukkig wisten we niet beter en zagen we nauwelijks in hoe je bijvoorbeeld anders dan met zo een techniek langs de elementen van een vector kon lopen. Wat een gelukkige naïveteit - en hoeveel liefde werd er niet al besteed aan het bouwen van een zuinig, om niet te zeggen broodmager programma!

Toen je dissertatie, vroeg in 1956, gepubliceerd werd, dachten wij dat je ZEBRA een truc was, een bitzuinige ad-hoc constructie, die dan wel nog eens commercieel verwezenlijkt werd, maar geen toekomst zou hebben. Met schaamrood op de kaken beken ik nu, dat we je genius miskend hadden: in de allerstrikste zin was jouw machine een gemicroprogrammeerde computer. Ik hoef wel niet te memoreren dat die microprogrammering later als innovatie en aanbeveling vermeld zou worden bij de vele opvolgers, waarvan het nog te betwijfelen staat of ze in enige zin inventiever waren dan jouw eersteling.

Bij al die driftige bezigheid met de hardware, de fijnzinnige programmering daarvan en, laten we het kort zeggen, de praktijk, heb je, in je dissertatie al, de fundamenten van ons vak niet uit het oog verloren. In de laatste, allerlaatste bladzijden van je dissertatie breng je een principieel moeilijk probleem naar voren: je buigt je, als een van de eersten na Turing, die je overigens citeert, over de vraag hoeveel verschillende operaties een

machine nu echt nodig heeft, overwegingen van efficiency even terzijde gelaten. Je komt tot de conclusie dat een zuivere één-operatiemachine kan volstaan.

Als ik het voor het zeggen had, en zolang - heel lang, hoop ik - je ons nog als emeritus ter beschikking staat zou ik de ZEBRA maar eens laten simuleren, om een volgende generatie zogeheten informatici wat beter bekend te maken met de charmes van het bit. Als ze dan ook de principes van de microprogrammering daarbij in hun vingers krijgen, zeg ik: des te beter!

Om nog eens wat verder in je carrière te delven: je was ook actief in de ontwikkeling van wat ons de weg gewezen heeft in de programmeertalen, namelijk als voorzitter van de werkgroep 2.1 met als opdracht: *De werkgroep zal verantwoordelijkheid aanvaarden voor het ontwikkelen, de specificatie en de verfijning van ALGOL.*

Voor velen van de aanwezigen op je afscheid zal dit, populair gezegd *ouwe koek* lijken. De passie, waarmee we destijds programmeertalen vergeleken en het floret, waarmee we op een haartje na gewend waren onze disputen daarover te bevechten, behoren allang tot de verleden tijd. We weten inmiddels, dat je vies kunt programmeren in een schone taal en brandschoon kunt werken in een vieze taal. Maar toen het onderwerp van schone taal nog brandheet was, behoorde jij al tot de voorvechters van de eerste brandschone taal. Ook hierin uitte zich je vooruitziende blik, vast en zeker gevoed door je intieme verhouding tot het bit. Laat ik het je maar bekennen: toen in mei 1960 het ALGOL-rapport me voor het eerst onder ogen kwam, dacht ik, zoals velen met mij, dat dit mooi, abstract, wiskundig gebazel was geen computer, die er ooit uit wijs zou kunnen... Je had het beter gezien, zoals me al in september van datzelfde jaar bleek: ene Dijkstra, al eerder genoemd, had op de X-1, ook al eerder genoemd, een complete implementatie beschikbaar. Anders dan hunne hooggeleerden Bauer en Samelson, die hun eigen product nooit begrepen hadden en het hadden gecastreerd door de recursie uit te sluiten, had Dijkstra in september van datzelfde jaar een werkende compiler, die hij testte door een simpel programmaatje voor het toetsen van de primaliteit van een klein geheel getal. Onder het mom van tijdsbesparing testte hij, voor een proefdeling, of de deler wel priem was, waarvoor het programma uiteraard zichzelf aanriep... Door je voorzitterschap alleen al van de werkgroep 2.1 heb je blijk gegeven van een visie, die in die vergane dagen van bijna onmachtige, vrijwel microscopische computers (al besloegen ze genoeg ruimte) waarlijk vooruitziend was.

Je hebt ook, al zullen anderen het wellicht memoreren, als het ware *en passant*, meer dan 100 ingenieurs aan de wereld afgeleverd, om nog maar te zwijgen van de mensen die bij jou een tentamenvervangende taak hebben volbracht. (Boze tongen beweren, dat als ze er te lang over deden, je hun opgave zelf maar gemaakt hebt.)

Menigeen van je abituriënten heeft van jou je liefde tot het zorgvuldig bewerkte bit, mag ik aannemen, meegekregen: zo staat me duidelijk bij, dat je een hele doctorandus tot de graad van (toen nog) doctor in de Technische Wetenschappen hebt kunnen bevorderen op grond van een proefschrift dat, ware de titel in het Nederlands vertaald, *Vullis ophalen*, zou moeten luiden, maar zich in het Engels wat sjieker aandoet als *Garbage Collection.*

Al met al betreur ik, *met* de faculteit, je vrijwillig afscheid. Het is mijn overtuiging dat met jouw heengaan een traditie onderbroken wordt waarzonder zelfs onze modernste studenten, met het hoofd op hol gebracht door fabels over vierde- en vijfde-generatie talen, het eigenlijk bezwaarlijk kunnen stellen. Dat zij zo, en we moeten je wil respecteren. We hebben één troost: je hebt in Delft school gemaakt en een aantal van je opvolgers en pupillen zal de traditie van de kennis van het bit voortzetten. In de hoop dat je nog lang als eigenwijze, zelfs wat overactieve, ja van tijd tot tijd irritante, collega ook na je emiritaat bij ons mag verwijlen, wilde ik je graag toeroepen: "*Vale*".

Teaching Programming

Peter Kluit, Gea Kolk

Technical University of Delft

Introduction

At the Technical Computer Science Department of the Technical University of Delft, education and didactics are not subjects of research. In spite of this teaching programming needs a lot of energy, thinking and discussing. The ideas about 'good programming' are developing. We have to take care of keeping education up to date. The importance of basic-courses is obvious: if we manage to make clear some important ideas and concepts about programming to students in basic-courses, this will be of great benefit later on.

The authors are involved in the first two basic-courses about programming: "Introduction to computer science" (an introduction to Pascal and problem solving with Pascal) and "Data structures and algorithms" (an introduction to program verification, data types and algorithms).

Teaching programming

When discussing teaching programming three questions are relevant :

1. What is a good program ?
2. How do I write good programs ?
3. How do I teach writing good programs?

Someone being unable of answering explicitly even the first of these questions may nevertheless be a good programmer. Teaching programming requires at least some reflections on these questions.

1. What is a good program?

Today's answer to this question is not the same as yesterday's answer, and no doubt tomorrow the answer will be different as well. Moreover even today there is no general agreement on this subject, although certain principles are generally accepted. Back in the fifties a program was considered good, provided that it

did run on test data, and produced the expected output,
made sparingly use of core and CPU-seconds.

Since then several things have happened. Kbytes grew into Mbytes and milliseconds grew into microseconds (microelectronics grows into the small). Together with falling prices this made the latter condition ever less pressing. In the meantime software problems grew into something summarized as the software-crisis. The former condition turned out to be both hard to meet and insufficient. The common algorithm for achieving the first condition was:

```
while not behaves_well(program) do change (program)
```

Depending on the quality of the procedure change this algorithm could be very inefficient. Actually termination is not guaranteed! Moreover, for any program considered complete an infinite process should be started, like:

```
while true do
  wait(bug)
  repair(bug)
  signal(bug)
od
```

where repair(bug) gives rise to an algorithm like the one mentioned above.

The present-day view might be explained by considering a program as a means of communication between man and machine; more precisely as a way to inform the machine about the intentions of a (wo)man. This implies:

a good program is well understood by the machine for which it is written
a good program is well understood by (trained) human beings.

The first of these conditions should be understood as : the program runs, and behaves according to the specification; not once or several times, but always (specification = a recording of the writers intention). This condition could not be checked by testing; testing, as Dijkstra pointed out, may show the presence of bugs, but never there absence. To make sure that a program is correct (the usual wording for the first condition) we need human verification. Hence the second condition.

The second condition gets even more important, however, when the first one is not met. This is the case :

during program development
during debugging (specifications not met)
during maintenance (specification changed)

(The absence of a third condition, that of efficiency, is a slight exaggeration. Resources still are not offered in infinite amounts (though mostly more than zero or one), and evident misuse still should be avoided. Except in special (e.g. real-time) applications, however, efficiency no longer is the first consideration in program design.)

Several inquiries have been held into the way programmers spend their time. Although the numbers may differ, in one thing they agree : programmers spend (too) much time in debugging and maintenance. The judicious word 'too' here represents the opinion that some effort during program development to raise comprehensibility may considerably diminish the effort in debugging and maintenance. As Mills [7] puts it : 'When a project was claimed to be 90-percent done with solid top-down structured programming, it would take only 10 percent more effort to complete it (instead of possibly another 90 percent!)'. Having shown the desirability of the stated requirements, we are left with the question : is it possible to write good programs?

2. How to write good programs.

Structured programming started as the opinion that programs should be open to mathematical proofs, together with some means to achieve this [6]: use only a few transparent control structures (sequence, selection and iteration) and no goto's. Items like abstraction and modularization were hinted at in [6] but greatly evolved in the years to follow, together with the ideas of top-down design and stepwise refinement. Structured programming in Dijkstra's sense requires that a program be developed together with its verification. After the last symbol has been typed, theoretically we should have a correct program at our disposal: bugs are absent, and testing is superfluous (in reality, bugs are scarce, and some testing is advisable). Most aspects of structured programming are generally accepted now, but the idea of program correctness proofs is much less widespread.

Structured programming has something to do with writing good programs. It is certainly not the final answer to the question: How do I write good programs? (Readers expecting to find the final answer to this question in this paper better stop reading now.) Modularization says that a program should be decomposed into sub-programs. Top-down design says that this decomposition should start, not at the bottom (=language) side, but at the problem side (the top) of the pyramid. Modularization and top-down design together say that a problem should be decomposed into sub-problems, according to the adagium 'divide and conquer'. So here is a problem and here is a knife: decompose! Not all decompositions are equal; some are much better than others. Algorithms for finding the best one are absent, however; with respect to this programming is still more an art than a science. Even if people agree that a certain decomposition is better than another, they may find it difficult to explain why it is better. In the same vein the author of the best solution might be unable to explain how he/she found it. With respect to this, neither the question 'What is a good program?' nor the question 'How do I write good programs?' has a complete and general answer. This will be a problem when we try to teach good programming.

One of the keywords that might act as a signpost in a design problem is the word problem-oriented, as opposed to language-oriented. Language-oriented programming means modeling the problem until it fits in the machine. Problem-oriented programming means modeling the machine such that it solves the problem. In a problem-oriented design we ask ourselves: what language would we like to have to solve our problem? We describe (specify) this language, solve the problem in terms of this language and finally extend our basic-language (possibly using one or more intermediate levels) to the language we have specified. This extension often (but not necessarily) consists in adding one or more new data types to the language.

New ideas on good programming, if accepted, will emerge with some delay in programming courses. One way to trace such ideas is to study the evolution of curricula , e.g. the recommendations for Academic Programs in Computer Science by the ACM Curriculum Committee on Computer Science in 1968[2], 1978[3] ,1984[4][5]. We will restrict ourselves to the basic courses CS1 and CS2.

In 1968 the predecessors of the courses CS1 and CS2 are the courses B1 'Introduction to

computing' and B2 'Computers and Programming'. Goal of the course B1 is :'..to provide the student with the basic knowledge and experience necessary to use computers effectively in the solution of problems.'

Subjects to be covered :

- internal structure of computers 38%
- programming language 38%
- design of algorithms 18%

The goal of the course B2 is : '.. to familiarize the student with the basic structure and language of machines,..give him a better understanding of the internal behaviour of computers,.. the use of assembly language,.. use computers more effectively...' Subjects to be covered (not exhaustive): computer structure,machine language, instruction execution , addressing techniques, macro's, program segmentation and linkage.

In 1978 the following objectives are given for the course CS1 'Computer Programming I':

— to introduce problem solving methods and algorithm development;
— to teach a high level programming language that is widely used; and
— to teach how to design, code, debug, and document programs using techniques of good programming style.

Subjects to be covered:

- Computer Organization 5%
- Programming Language 45%
- Algorithm Development 45%

Comparing these figures with those of B1 we see a growing interest in algorithm development at the cost of computer organization. This trend shows up even more evident in the second course.

The course CS2 'Computer Programming II' in 1978 has as its objectives :

— to continue the development of discipline in program design, in style and expression, in debugging and testing, especially for larger programs;
— to introduce algorithm analysis; and
— to introduce basic aspects of string processing, recursion, internal search/sort methods and simple data structures.

Any resemblance with the 1968 course B2 seems to be absent.

In 1984 the descriptions of CS1 and CS2 are updated. There have been shifts in teaching emphasis resulting from the growing discipline of software engineering. The development of good problem solving and programming techniques is emphasized. The objects of the course CS1 'Introduction to program methodology' are :

— to introduce a disciplined approach to problem-solving methods and algorithm

development;
— to introduce procedural and data abstraction;
— to teach program design, coding, debugging, testing, and documentation using good programming style;
— to teach a block-structured high-level programming language;
— to provide a familiarity with the evolution of computer hardware and software technology;
— to provide a foundation for further studies in computer science.

In the course CS2 'Program Design and Implementation' the subjects of CS1 are expanded. CS2 concentrates on data abstraction and issues related to abstract data types and data structures. Many topics formerly in CS7 'Data Structures and Algorithm Analysis ' are now transferred to CS2. New topics in 1984 are :

— Specification-Understanding and articulating problem definition in terms of user requirements, and the translation of these requirements into functional specifications.
— Program correctness- Planning of tests and an introduction to verification using assertions and invariants.

3. How to teach good programming.

Having discussed why and what, we now come to how things should be taught.

Most students are willing, or even eager, to write programs, that is to type screens full of lines of code. They tend to be much less willing to write good programs in the sense we have described above, unless we spend much effort in convincing them of the benefits of this approach. The fastest way to obtain a program seems to be to start with the first line of code, then write the second line, etc.. Certainly it is the most attractive way, as it soon leads to the satisfying feeling:'I have a program'. Much later, when they really have a program (that is: a correct one) students seldom realize that the main portion of the job was done after this satisfying moment. In view of the addictive nature of this method, students should be kept from writing even a single program this way.

Teaching a programming-language is not too hard a job. We just show the language elements, and explain their syntax and semantics. This is exactly what students want to be taught, as they are convinced that programming and coding are identical. There has been a tendency to overemphasize syntax in programming courses. Of course some knowledge of syntax is necessary. As syntactic mistakes usually are easier to find and repair than semantic ones, there is reason to concentrate on the meaning of language elements. In terms of what could we explain this meaning? Formal semantics does not seem appropriate in a first course. An explanation in terms of bits, bytes and addressing-modes would be too much machine-oriented. The proper thing to do seems to be to offer a model of the virtual machine that is suggested by the language. However for Pascal and related languages this model gets fairly complicated if we have to explain things like parameter-passing and scope rules. From this point of view, procedures and functions are advanced topics, to be taught only at the end of the course. A course like this would be very incomplete, as no attention is given as to how to make the language-elements work together to

solve the problem at hand. This kind of language-oriented course leads to language-oriented programs, consisting of pages full of amorphous code.

Teaching programming in a problem-oriented style is a much harder job. We already mentioned the fact that we do not have a complete and general answer to the question 'what is a good program?', nor to the question 'how do I write a good program?'. Nevertheless we are supposed to disseminate the principles of good programming. From a problem-oriented point of view, the design of algorithms and programs should be taught first, before introducing a programming language. Designing algorithms without a language to express them is fairly hard. The use of pseudo-code only partially solves the problem. Pseudocode pretends to be independent of the programming language to be used, but in fact it is not, so students get puzzled as to why pseudocode looks the way it looks. The best strategy for the moment seems to be to intermingle the two; with an accent on design, and sufficient language to express the algorithms.

A study of about 40 books on 'programming in Pascal' shows the trends in this direction . Most books of the seventies teach Pascal, and nothing but Pascal. The order in which the subjects are discussed is more or less standard: simple data types, looping and selection, structured data types, procedures and functions. In the books of the eighties more attention is given to algorithm design. Flow-charts (Gries preferred to call them flaw-charts) go out of use, and sometimes are replaced by Nassi-Schneidermann diagrams. The more recent books show a different ordering of the subjects: procedures and functions (the basic tools of abstraction) are discussed as early as possible.

Abstraction requires that a procedure is understood and used starting from its specification, and nothing but its specification, and that it is written starting from its specification and nothing but its specification. In other words, all information that is passed between the writer and the user of a procedure is passed via the specification (and not, for example via some vague thoughts in the head of the programmer, who unfortunately happens to be both user and writer). If we allow students to write specifications in a language of their own, these specifications will be vague, ambiguous, incomplete and in short insufficient. Taking abstraction serious therefore implies introducing some kind of specification-language. On the other hand, introducing one more formal language at an early stage in the course most probably will give rise to more problems than it solves.

The philosophy of problem-oriented, as opposed to language-oriented programming has its consequences for the language to be chosen. This language should be simple, in order not to distract attention from the programming problems. On the other hand it should have sufficient expressive power to illustrate the concepts and ideas we want to introduce (cf. [8]). The ACM Curriculum Committee considers the use of a block-structured high-level language essential to teach good programming style and methods properly. In the '84 recommendations Pascal, PL/1 and Ada (and Modula-2 for CS2) are mentioned as possible choices. FORTRAN and BASIC, however popular, definitely are considered not suitable for teaching purposes. In the ACM recommendations non-imperative languages are strikingly out of scope. Now that programming concepts are considered more important than programming languages, the choice of the kind of language to use (imperative, functional, logic,..) should precede the choice of a particular

language. Several of the difficulties mentioned above could be avoided using a functional language. No doubt several new difficulties will emerge. Experiments with a first programming course using a functional language are hopeful, however [1]. Functional languages provide a steep way to high-level programming. Many people may prefer the long and winding road.

References

[1] Harold Abelson, e.a. , Structure and Interpretation of Computer Programs, MIT Press , 1987.

[2] ACM Curriculum Committee on Computer Science. Curriculum '68, recommendations for academic programs in computer science.Comm. ACM 11 3 (March 1968), 151-197.

[3] ACM Curriculum Committee on Computer Science. Curriculum '78, recommendations for the undergraduate program in computer science.Comm. ACM 22 3 (March 1997),147-166.

[4] ACM Curriculum Committee Task Force for CS1. Recommended Curriculum for CS1,1984. Comm. ACM 27 10 (October 1984), 998-1001.

[5] ACM Curriculum Task Force for CS2. Recommended Curriculum for CS2,1984. Comm. ACM 28 8 (August 1985), 815-818.

[6] Edsger W. Dijkstra, Structured Programming. In : Software Engineering Techniques, NATO Science Committee,Rome 1969, 88-93.

[7] Harlan D. Mills. Structured Programming: Retrospect and Prospect. IEEE Software (November 1986), 58-66

[8] W.L. van der Poel, The programming language HILT and its use in teaching. In : Programming Teaching Techniques, Proceedings of the IFIP TC-2 Working Conference on Programming Teaching Techniques, Zakopane, Poland, September 18-22,1972. 15-42

Rekenen is moeilijk

S.C. van Westrhenen
Technische Universiteit Delft

September 19, 1988

1 Rekenen in calculi

Het idee om het menselijk redeneren te reduceren tot het rekenen in een calcul is al oud. Een eerste uitwerking van dit idee vindt men bij de Catelaanse filosoof en theoloog Raymundus Lullus (1232-1315). Deze briljante geleerde ontwikkelde een geometrisch systeem van driehoeken, vierhoeken en cirkels met behulp waarvan hy door manipulatie met 'dwingende kracht' een aantal theologische waarheiden kon bewijzen. Met dit systeem ging hij naar Afrika om de aldaar wonende arabieren te bekeren. Helaas gebruikten deze ook mechanische hulpmiddelen,namelijk stenen, waarmede zij hem stenigden.

Een andere bekende poging het redeneren te herleiden tot het rekenen in een calcul komt van de Duitse filosoof G. Leibnitz (1646-1716). Deze wilde een logica ontwerpen waarmede men op elk terrein van wetenschap, waar een strenge redenering mogelijk is, redeneringen kan beoordelen (ars judicandi) of een redenering kan ontwerpen (ars inveniendi). Deze denkbeelden zijn in de negentiende eeuw, bij het ontstaan van de mathematische logica, ten dele verwezenlijkt.

In de dertiger jaren van deze eeuw is door Gödel, Church en anderen bewezen, dat niet elke wiskundige theorie axiomatiseerbaar of beslisbaar is. In het laatste geval bestaat er geen beslissingsprogramma waarmede men voor een willekeurige formule uit de theorie in een eindig aantal stappen kan nagaan of de bewering waar of onwaar is. Dit toont aan, dat formele calculi niet altijd toereikend zijn voor de formalisering van het redeneren.

Door de komst van de computer na de tweede wereldoorlog is de studie van het rekenen in logische calculi enorm gestimuleerd. Tal van 'theorem provers' zijn ontworpen en geprogrammeerd. De eerste publicatie, waarin het gebruik van de computer bij het bewijzen van theorema's werd

beschreven, verscheen in 1957. Deze was getiteld ' The Logic Theory Machine' en geschreven door Newell, Shaw en Simon. In deze publicatie werd een programma beschreven waarmede men theorema's kon bewijzen uit de propositielogica uit de 'Principia Mathematica'. Na deze publicatie zijn veel andere publicaties over dit onderwerp verschenen, zie ook [1].

In het onderzoek naar het rekenen in calculi - of het ontwerpen van theorem provers - kan men twee stromingen onderscheiden, namelijk de 'logische' en de 'mensgerichte' benadering. Bij de logische benadering zijn de theorem provers gebaseerd op een logisch systeem, eventueel aangevuld met sterk syntax gerichte heuristische regels voor het gebruik van bijvoorbeeld de afleidingsregels.

Bij de tweede benadering ligt de nadruk vooral op de simulatie van het probleemoplossend gedrag van de mens. Shaw, Newell en Simon zijn typische vertegenwoordigers van deze 'mensgerichte' benadering. De 'Geometry theorem-proving machine' van Gelernter was een mijlpaal op het gebied van het gebruik van A.I. bij het bewijzen van theorema's.

Daarentegen hebben Hao Wang en Gillmoore, Davis en Putnam in het begin zestiger jaren 'theorem provers' geschreven die gebaseerd zijn op het Theorema van Herbrand, de afleidingssystemen van Gentzen of de semantische tableau's van E.W. Beth. In dit kader was ook de vondst van de resolutie door J.A. Robinson in 1967 een belangrijke vooruitgang.

De moderne theorem provers voor het bewijzen van wiskundige stellingen, bijvoorbeeld van Bledsoe, worden diverse hulpmiddelen zoals logica, non-standard analyse, heuristische technieken gecombineerd om het bewijs van een gegeven bewering te vinden. De theorem prover van Bledsoe is in staat stellingen op het gebied van verzamelingenleer, analyse, elementaire topologie, te bewijzen. Sommige van deze theorem provers waren zelfs in staat 'openstaande' stellingen te bewijzen. Zie [2].

Theorem provers zijn vaak zeer belangrijke onderdelen van kennisverwerkende systemen of KBS (knowledge based systems). Deze systemen vinden toepassing in bijvoorbeeld diagnostische systemen, expertsystemen, patroon- en beeldverwerking etc..

2 Komplexiteit van arithmetische en logische calculi

Sinds de zeventiger jaren heeft men zich geworpen op de studie van de komplexiteit van het rekenen in calculi. Als spoedig bleek dat het rekenen in

calculi zoals de aritmetiek en de logica principieel zeer inefficient is. Tevens is door deze onderzoeksresultaten het verschil tussen beslisbare en onbeslisbare theorieën vervaagd. Immers, in het beslisbare geval duurt de berekening soms te lang en in het onbeslisbare geval, als er geen positief antwoord mogelijk is, stopt de computer niet. In beide gevallen komt er geen antwoord in een redelijke tijd.

Om de komplexiteit van algoritmen met elkaar te kunnen vergelijken worden de algoritmen geprogrammeerd op deterministische of indeterministische Turingmachines. Door vergelijking van de rekentijden en het ruimtebeslag op de Turingmachine kan men de komplexiteit van algoritmen met elkaar vergelijken. Zie [1,2]

De introductie van niet-deterministische Turingmachines is meer een technische truc, want een berekening op een niet-deterministische Turingmachines M kunnen gesimuleerd worden op deterministische Turingmachine M_n. Het verband tussen de rekentijden T_M en T_N is $T_N(n) \approx c.2^{d.T_M(n)}$ met c en d positieve constanten en n is de lengte van de invoersymboolrij. Het verband tussen de ruimtekomplexiteit van beide berekeningen is kwadratisch (bewezen door Savitch), namelijk $S_N(n) \approx [S_M(n)]^2$

Een van de eerste stellingen over de komplexiteit van beslissingsalgoritmen (op Turingmachines geprogrammeerd) van beslisbare theorieën is de stelling van Rabin-Fischer.

Stelling 1 (Rabin-Fischer 1974)
Σ is een beslissingsprogramma voor de rekenkunde (N;+,=;0,1). Dan bestaat er een constante $c > 0$ zodat er oneindig veel formules F met lengte n, waarvoor het programma Σ minstens $2^{2^{c.n}}$ stappen moet uitvoeren.

Het bewijs kan geleverd worden te veronderstellen dat het aantal stappen van Σ kleiner is dan de genoemde ondergrens. Door gödelisering van de de beslissingsprogramma's en diagonalisering voert dit tot een contradictie. Zie [4,5].

Voor de Presburger Arithmetiek (N; +, $\leq$, 0) en de reële getallen met de optelling (R; +, =; 1), respectievelijk aangeduid door PA en RO gelden voor de tijds- en ruimtekomplexiteit van de betreffende beslissingsprogramma's de volgende ondergrenzen.

Stelling 2 Ondergrenzen tijd en bovengrenzen ruimte.
PA: NTIME(2^{cn} EXPSPACE;
RO: NTIME($2^{2^{cn}}$) 7ex DSPACE($2^{2^{dn}}$).

Voor beslisbare logische theorieën gelden soortgelijke ondergrenzen voor de tijd- en ruimtekomplexiteit van de beslissingsprogramma's. Zie [5]. Voor beslisbare logische theorieën gelden soortgelijke resultaten.

Laat SAT, MON(=)en FUN respectievelijk aanduiden de verzameling van alle vervulbare formules uit de propositielogica, de verzameling van alle vervulbare formules uit de monadische predikatenlogica en de verzameling van alle vervulbare formules uit de predikatenlogica waarin slechts één functiesymbool en de gelijkheid voorkomt.

Voor deze theorieën geldt:

Stelling 3 Ondergrenzen
SAT is NP-compleet.
MON: NTIME($2^{c\frac{n}{log(n)}}$) (ondergrens tijd) NTIME($2^{d\frac{n}{log(n)}}$) (bovengrens ruimte).

Bovenstaande resultaten geven aan dat betrekkelijk elementaire wiskundige theorieën de beste beslissingsalgoritmen toch nog altijd exponentieel in ruimte of tijd zijn.

Het is daarom niet zo verwonderlijk dat men in theorem provers diverse heuristieken gebruikt tracht te gebruiken.

In 1961 lanceerde de Nederlandse logicus E.W. Beth in zijn boek 'Formal Methods' het idee statistiek als hulpmiddel te gebruiken.

Het gebruik van de waarschijnlijkheidsrekening als heuristiek kan een voudig gedemonstreerd worden aan een eenvoudig geval. Laat PRENEX de verzameling van alle onvervulbare formules in prenexnormaalvorm uit de predikatenlogica zijn. Stel men wil nagaan of de formule $\forall x\ \forall y\ P(x,y)$ tot PRENEX gehoort. Als $F \in PRENEX$ dan moet er volgens de stelling van Herbrand een eindige onvervulbare verzameling A_n van n verschillende instantiaties P(a,b) bestaan.

Men zou kunnen proberen de onvervulbare verzameling A_n sneller door trekking te verkrijgen. Men kan deze instantiaties van F nummeren en daarna trekken uit een waarschijnlijksheidsverdeling. Na iedere trekking gaat men na of de tot nu toe verkregen verkregen verzameling van instantiaties A_n (n=1,2,...) onvervulbaar is.

Is de verzameling A_n onvervulbaar dan stopt men, anders wordt een nieuwe trekking gedaan waarna men A_n onderzoekt etc.. De stochastische bewijslengte van F is gelijk aan n als A_n onvervulbaar is.

Stelling 4 Als F is een onvervulbare verzameling clauses,dan is de stochastische bewijslengte eindig met waarschijnlijkheid 1.

Verder voldoet de procedure aan plezierige statistische eigenschappen zoals asymptotisch goed en Bayes. Zie [6]

Men kan de formule F ook in prenexnormaalvorm schijven met alleen alkwantoren. De existentiekwantoren worden dan vervangen door Herbrandfunctoren. In plaasts van instantiaties van F trekt men instantiaties van de disjunctieleden of clauses. De onvervulbaarheid van de verzamelingen A_i van clauses kan gecontroleerd worden met behulp van de genormaliseerde resolutiemethode van Tsjeitin.

De keuze van de clauses die geresolveerd worden vindt met een stochastische trekkingsmethode plaast. Voor deze methode geldt het volgende:

Stelling 5 Voor oneindig veel $n \in N$ bestaan er onvervulbare formules van lengte n met een stochastische bewijslengte groter dan $2^{\sqrt{n}}$

Men kan bewijzen dat $E(\log(L(n)) \approx n^{1.5}$ Zie [7] Uit deze opmerkingen blijkt dus de bewijsprogramma's voor de bovenstaande logische calculi een hoge komplexiteit hebben. Anders gezegd: Rekenen is moeilijker dan je denkt.

3 Literatuur- verwijzingen

1. Cook s. + Rechhow R. - Onthe length of proofs in the Propositional Calculus. A.C.M. Proc. 6th. symp. on the Theory of Comp. Washington 1974. pp. 135-148.

2. Ferrante J. + Rackoff D.W. - The computational Complexity of Logical theories.Springer Lect. Notes nr.718, 1979.

3. Loveland D.W. - Automated Theorem Proving: A Quarter-Century Review. Contemporary Math. Automated Theorem Proving. AMS vol. 29, 1984

4. Specker A. + Strassen v. - Komplexität von Entscheidungsproblemen. Springer-Verlag Lecture Notes in Comp. Science nr. 43. 1976.

5. Wagner K. + Wechsung G. - Computational Complexity. Reidel Publ. Comp. 1986.

6. Westrhenen S.C. van - Statistical Studies of Theoremhood in Classical Propositional and First Order Predicate Calculus. J.A.C.M. 19 (1972), pp. 347-365.

7. idem - Note on a Probalistic Proof Procedure for the first Order Predicate Calculus. Delfth Progress Report 6 (1981), pp.170 - 176.

SOFTWARE REUSABILITY and PROGRAMMING LANGUAGES

E.M. Dusink
Faculty of Mathematics and Computer Science
Delft University of Technology
2628 BL
Julianalaan 132
Delft
The Netherlands
uucp: ...mcvax!dutinfd!betje

1. Introduction

It is generally accepted that an increased reuse of products from the software life cycle contributes to a significant improvement of programmer's productivity. Already in 1968, McIlroy (1976) proposed a software components repository from which software parts could be assembled. The idea was elaborated on by Barnes (1982). The importance of reusability is stressed (a.o.) in discussions on improving the software-production process. Furthermore, reuse is recognised to be an important mechanism in the envisioned *Software Factory* (MacAnAirchinnigh 1987). Unfortunately, in current practice, software reuse is hardly evolved beyond its most primitive forms: the use of subroutine libraries and sometimes the unstructured and partial reuse of previously written fragments of program text. Arguments for this phenomenon are found in Tracz (1988), he identifies a number of technical, organisational, and psychological reasons why the reuse of software often fails.

Reuse can exist in all phases of the software life cycle. The reuse mentioned in this paper is only the reuse found in the implementation phase and thus contributes only for a small part of the possible gains. But as long as software reuse is not evolved beyond the use of subroutine libraries and the partial reuse of previously written fragments of program text the programming language used is important.

In this paper the impact of the language features on the two reuse paradigms is described. This means the mentioning of good as well as missing features with regard to reuse. Besides specific reuse characteristics, programming languages have of course to satisfy 'normal' requirements with which good programming languages can be designed. In (Ledgard 1981) and (Wirth 1974) requirements are given for the design of programming languages. These requirements are of indirect relevance to reuse since a language in which it is easy to program good algorithms also helps to program reusable algorithms.

2. Reuse paradigms and classes of programming languages

Reuse of software can be considered as a means to support the construction of new programs using in a systematical way existing designs, design fragments, program texts, documentation, or other forms of program representation. Within this interpretation of

reuse, two different approaches can be identified (Ververs 1988): the *transformational* approach and the *compositional* approach.

- In the *transformational* approach programs are written in terms of abstract specifications using a wide-spectrum language (a language which can be used during the whole development process) or an application language. Program transformations are then applied to achieve efficient target programs. Thus programs are constructed by transformation. This approach is also known as a *wide-spectrum approach to software reuse* (Cheatham 1984).
- In the *compositional* approach (the software IC approach) software components, e.g. packages, functions and other forms of modules, are used as basic building blocks in the software construction process. Programs are constructed by combining existing software components (either in source form or some other representation form);

In both approaches software and/or designs are reused, the important difference between the approaches is the way components[1] are reused, while furthermore the way new software is constructed is different.

Before components can be reused, they have to be written in some language. Language characteristics can influence the ease with which these components can be written. Languages can be classified in several ways. With most classification schemes languages can be put in more than one class. In the following a distinction will be made between the imperative and declarative languages. The imperative languages can be divided into the 'classic' languages like Pascal, Basic, and Fortran and the object-oriented languages like Smalltalk, Simula, and C++. The declarative languages can be divided into the functional languages as pure Lisp, Hope, and Miranda and the logic languages as Prolog and Eqlog.

3. The transformational approach

The transformational approach can be categorised as a form of *automatic programming.* The programmer 'programs' in terms of very high-level abstract algorithms written in some wide-spectrum language, or perhaps using domain-oriented specification languages. The abstract specifications are translated into an efficient program in some executable language with the help of transformation schemes. With the transformational approach, reuse is obtained in two ways. First, as e.g. Cheatham (1984) points out, abstract algorithms provided by the user are candidate for reuse due to their generality. Second, the transformation and interpretation schemes are themselves candidate for reuse (Feather 1983).

Two aspects have to be considered when looking at the influence of programming languages on reuse. The first is the ability to write highly abstract specifications, the other is the ability to write transformation schemes for the specification language.

---

1. Because this paper is about the relation between programming languages and reuse, the term component should be interpreted as a piece of code.

The literature about the transformational approach is vague about the desired characteristics for wide-spectrum and domain-oriented languages. The literature is even vague about what these languages are. However, examples of wide-spectrum languages are CIP (CIP 1985), SETL (Schwartz 1986). The rest of this chapter contains comments on the distinction between imperative and declarative languages with respect to reuse. and some comments on wide-spectrum and domain-oriented languages.

In the transformational approach only the 'what' is necessary, the transformation rules which contain the domain knowledge provide the 'how'. Writing components in imperative languages delivers as abstract specifications as the language allows. A negative aspect of imperative languages is the tendency to ease the description of the 'how' instead of easing the description of the 'what'. The construction of transformation schemes for imperative languages is difficult because the context dependency of statements. From the two subclasses of the declarative languages a logic language in which only the 'what' has to be described instead of the 'how' appears to be more fit for the transformational approach than a functional language where one tends to describe the 'how' also. For an easy implementation of transformation schemes the use of declarative languages with their total lack of dependence on the place of occurrence is a good choice.

In the wide-spectrum approach one language suffice during the whole development of software. Using a single base language limits the transformation alternatives to the primitives used by the language (Shaw 1984). Specification languages, which also describe the behaviour, need to have a minimal gap between informal conceptualisation and formal specification. A drawback is that in most cases automatic transformation is not yet possible. In Gist (Feather 1983), for example, automatic transformation is not yet possible.

In the domain approach the languages have to contain knowledge about the selected application domain, and the facilities for expression and abstraction have to be very powerful. This will result in a large number of languages, at least one for every application domain. Because almost every application involves more than one application domain, planning to live with any predetermined number of special purpose features will restrict the usefulness of the language. In a domain-oriented language the objects and operations represent information about a problem domain. The 'what' is important and reused. The objects and operations are modeled and implemented in another domain-oriented language which is more oriented towards efficient, executable code of which the objects and operations are modelled and implemented in another domain-oriented language and so on, with a general purpose language at the bottom. These modelling connections represent different design possibilities (Neighbors 1984).

4. The compositional approach

In the compositional approach the engineer, the programmer, combines (instances of) pre-fabricated components, i.e. existing program fragments, to form larger components or programs. Two aspects have to be considered when looking at the influence of programming languages on the reusability of components: the writing of components and the combining of components. The number of *guidelines for reusability*, written down for e.g. the potential reuse of Ada packages and subprograms is quite large, the guidelines from Bott

et al (1986) are probably the best-known illustration. From these guidelines and from the problems mentioned by the authors of these guidelines, desired language features can be derived. Since combining pieces of program is a complex activity every bit of support from a programming language is welcome. Since glueing requires some understanding of the syntax and semantics of the components, these features should be easily understood. No attention was given to composition from mixed languages, in (Hayes 1988) a system for solving this problem is described.

4.1 Imperative languages and the compositional approach

In the class of the imperative languages two approaches can be distinguished. One which is procedure oriented, the 'classical' languages, and one which is object oriented, the object-oriented languages. These two approaches influence the design of algorithms. Not all imperative languages support both approaches. (For more information about the theory of type handling, see: Danforth 1988)

These two approaches are represented by the 'classical' languages like C, Fortran, and Pascal and the object-oriented languages like Smalltalk and Simula. The features and required features of these groups in relation with reuse will be described below.

4.1.1 Writing components in 'classical' languages

Bott et al. (1986) gave implementation guidelines which, when and if applied, should result in reusable components. The implementation guidelines of these authors were translated into requirements for programming languages by the author and follow below.

Components must be independent from their environment. Thus, information passing should only be allowed via parameters; not via global variables or whatsoever. Further, components should not cause side effects, thus, once again the conclusion, information passing can only be allowed via parameters. To support the internal coherence of components nesting of procedures, packages and the like should not be allowed. Rickert (1986) argued that the use of internal procedures can lead to assumptions on the environment in which they are used and that not enough thought is given to the interface of the components.

To be able to manipulate user-defined types exactly like language defined ones, a language should offer the possibility of a package-like structure in which a user can define his own data type with the belonging operations. Outside this 'package' variables of this 'package-type' can be declared and these can only be accessed via the operations defined in the package. In short, variables of user-defined types are treated exactly as language defined ones, which can only be manipulated by predefined operations. Furthermore, because of the packaging mechanism and the non-nesting of modules, an incorporating mechanism like the 'with clause' of Ada is necessary. This also states exactly the dependence of a component on the environment in which it can be used.

To obtain data abstraction and functional abstraction a generic facility must be provided. As genericity of components can lead to many generic parameters, of which some can have, in most cases, default initialisations. The generic facility has to be such that the generic parameters can be grouped, with the possibility of default initialisations for every

group. In this way, the number of parameters to be handled can reduced. For a safe coupling of components strong type checking is necessary with a possibility of a kind of coercions. The coercions should be user-specified on the level of the used data types.

4.1.2 Writing components in object-oriented languages
Some authors claim that reusable components need to be object-oriented (Dennis 1986, Ledbetter 1985, Cox 1984). Nevertheless, no implementation guidelines from which language features could be retrieved could be found. In the following some loose remarks found in literature are gathered.

Most features named in the paragraph about 'classical' languages are also of relevance to the object-oriented languages. If they don't apply, they are named in the following, together with other remarks on the object-oriented languages. Components are classes in the object-oriented approach. Nesting should be allowed in the form of subclasses for, as Deutsch (1983) stated, the possibility to rely on subclasses to provide missing parts provides a way in which abstract algorithms can be written and concrete parts can be reused. The incorporation mechanism is found in the form of inheritance. As inheritance is the only form of incorporation it seems unnecessary restrictive to have only one super class (the hierarchy is tree-like) from which can be inherited. Multiple inheritance should be allowed. See also Wegner (1984). Kaiser (1987) also arguments for possibilities of combining/merging classes instead of only having the possibility of refining classes. Also in object-oriented languages, user-defined data types should be treated exactly as language defined ones. It has to be possible to refine or generalise in subclasses, which means that redeclaration/undeclaration of operations defined higher in the hierarchy has to be possible.

4.2 Declarative languages and the compositional approach

In declarative languages the notion of state does not exist and often referential transparency does exist. Thus, the meaning of an expression is independent of its context. This last characteristic is very desirable with regard to reuse because composition becomes very easy. The meaning of components is easier to understand, the whole is exactly the sum of the parts. This class can be divided into functional languages and logic languages. In the first subclass the importance of function application is stressed so the fundamental computational component is the function. Miranda, SASL, KRC, ML, pure Lisp, Hope, FP (Eisenbach 1987) are examples of this subclass. The other subclass has the relation as the fundamental unit. Prolog and Eqlog (Goguen 1984) are examples of this subclass.

On these two classes of languages no literature was found.

4.2.1 Functional Languages and the compositional paradigm
To be really referential transparent, no assignment in whatever form should be allowed. In functional languages functions are seen as first class citizens, the distinction between data and functions disappears. Data and functions can be structured and manipulated with equal facility. The disappearance of the distinction between data and functions can be explicated by the mathematical function which is a shorthand for a map of data to data. The ease of manipulating functions opens new perspectives for the writing of software. It also seems to give more possibilities for the composition of components.

Some languages offer the possibility of polymorphic data types. Such a provision can be seen as a kind of generic facility. Genericity makes functions wider applicable. The possibility of defining data types should be a step in the right direction. In most languages however, it is not easy to construct other data types than offered by the language.

4.2.2 Logic languages and the compositional paradigm

Although logic languages are seen as declarative languages, they lack referential transparency. The result of a program is influenced by the order in which the rules are given. To be really referential transparent and to have fair calculation strategies, the order of the rules should be irrelevant. (This seems to be more an implementation question for these languages than an inherent feature.) The organisation of the rules is not a part of the problem solving, but it is only a matter of efficiency.

Predicates which say something about the 'how', like the cut operator in Prolog, or which give the possibility to a program to adapt itself, like the retract and asserta of Prolog, have to be abandoned. It takes too much time to understand the workings of a program with the above mentioned features, and thus these features reduces the reuse capabilities.

In the context of the compositional approach, some possibility for grouping clauses to construct components and to call them, is also necessary as well as some form of parameterisation and typing. As far as I know, only Eqlog has modularity and typing. To extend the possibilities of logic programming, the addition of the equals predicate to the Horn clauses gives the possibilities of functional programming (Middelburg 1986).

5. Current research: state of the art

In the following paragraphs, two languages designed for delivering reusable components and two models for connecting components, with the first model supporting the compositional paradigm and the second supporting both paradigms are mentioned.

5.1 The language EIFFEL

The statically typed language EIFFEL is specially defined to support both genericity and inheritance. Having full inheritance and genericity in the same language is found to result in a redundant and overly complex design and a borderline at unconstrained genericity has thus been put in the Eiffel language. For more details see (Meyer 1986).

5.2 The language Meld

Meld, an object-oriented language, is specially designed to build reusable software components. The authors state the flaws in the three most popular form of reuse (libraries, generators, object-oriented programming) and hope to overcome them with their language. The authors of the article call Meld a declarative language. For more details see (Kaiser 1987).

5.3 Connecting through sockets and plugs

Elliott et al (1987) describe a *model* for software components in which particular attention is paid to connecting components. A component provides in its interface *sockets* through

which it exports access to its services and *plugs* through which access to services of another component is requested. Plugs and sockets are part of the visible interface of a component, they can have various shapes and sizes. Identical sockets or plugs can have different semantics.

Furthermore, a proposed implementation of the model in terms of Ada language constructs is given by the authors. The implementation is described in terms of guidelines with respect to the use of Ada to describe/implement components, sockets and plugs. The first few guidelines suggest an Ada binding in terms of generic packages whose specifications have no context dependencies. Other guidelines impose a disciplined, consistent and explicit style of software engineering.

5.4 LIL, connecting through theories and views

A more formal approach to a solution of the problem with software reuse is presented by Goguen (1986). His approach depends on three major semantic concepts: theories (to associate semantic descriptions with software components by providing axioms), views (to describe semantically correct bindings), and distinction between horizontal and vertical composition. Together, these three concepts give a powerful notion of generic software. A module interconnection language resembling Ada's specification part, plus commands for interconnecting components to form systems, is developed. The syntax is Ada-like. The goals are: narrowing interfaces, allowing more flexible bindings of generic components, allowing interactive version and configuration management, and integrating hierarchical design methodology. LIL is said to support top-down, bottom-up, transformation, instantiation, and data driven and data flow programming.

6. Conclusion

Most research is done on the most widely used class of languages, the imperative ones. With languages special designed to support reuse no experience is gathered yet. Imperative languages are by nature well suited to the compositional approach. Declarative languages deliver less problems for the transformation schemes and are in that aspect better suited for the transformational approach.

7. References

[Barnes-1982] Barnes, J.P.G. Programming in Ada. Addison-Wesley Pub. Co., 1982

[Bott-1986] Bott, M.F., Elliott, A., Gautier, R.J. Ada reuse guidelines. ECLIPSE/REUSE/DST/ADA_GUIDE/RP, 1986. Software Sciences Ltd.

[Cheatham-1984] Cheatham, T.E. Jr. Reusability Through Program Transformations. IEEE Transactions on Software Engineering, V 10 (5), 589-594, September 1984

[CIP-1985] The Munich Project CIP, Volume 1: The Wide Spectrum Language CIP-L. Series: Lecture Notes in Computer Science, eds. G. Goos, J. Hartmanis Springer-Verlag, Berlin 1985

[Cox-1984] Cox, B.J. Message/Object Programming: An Evolutionary Change in Programming Technology. IEEE Software, V 1 (1), 50-61, January 1984

[Danforth-1988] Danforth, S., Tomlinson, C. Type Theories and Object-Oriented Programming. ACM Computing surveys V 20 (1), 29-72, March 1988

[Dennis-1986] Dennis, R. St., Stachour, P., Frankowski, E., Onuegbe, E. Measurable characteristics of reusable Ada (R) software ACM SIGAda Ada letters, V VI (2), 41-50 March/April 1986

[Deutsch-1983] Deutsch, L.P. Reusability in the Smalltalk-80 Programming system. Proc. of the ITT Workshop on Reusability in Programming, pp 72-76, Stratford, Connecticut, September 7-9, 1983

[Elliott-1987] Elliott, A., Gautier, R.J., Welch, P.H. Engineering Reusable Software Components in Ada (Some Problems and Some Guidelines). Ada-Europe Reuse Working Group, internal report, 1987

[Eisenbach-1987] Eisenbach, S. FUNCTIONAL PROGRAMMING: Languages, Tools and Architectures. series: computers and their applications, series eds: Brian L. Meek Ellis Horwood limited, 1987

[Feather-1983] Feather, M. Reuse in the context of a transformation based methodology. Proc. of the Workshop on Reusability in Programming, September 7-9 1983. Newport, RI

[Goguen-1984] Goguen, J.A., Meseguez, J. Equality, types, modules and generics for logic programming. Proc. 2nd International Conference on logic programming, pp 115-125, 1984

[Goguen-1986] Goguen, J.A. Reusing and Interconnecting Software Components. IEEE Computer, V 19 (2), 16-28, February 1986

[Hayes-1988] Hayes, R., Manweiler, S.W., Schlichting, R.D. A Simple System for Constructing Distributed, Mixed-Language Programs. Software - Practice and experience, V 18 (7), 641-660, July 1988

[Kaiser-1987] Kaiser, G.E., Garlan, D. Melding Software Systems from reusable building blocks. IEEE Software, V 4 (4), 17-24, July 1987

[Ledbetter-1985] Ledbetter, L., Cox, B. Software-ICs: A plan for building reusable software components. Byte, 307-316, June 1985

[Ledgard-1981] Ledgard, H., Marcotty, M. The Programming Language Landscape. The SRA Computer Science Series, SRA, 1981

[MacAnAirchinnigh-1987] Mac an Airchinnigh, M. Conceptual Model of an Ada Software Factory. Ada-Europe Environment Working Group, internal report, 1987

[McIlroy-1976] McIlroy, M.D. Mass-Produced Software Components. Software Engineering Concepts and Techniques, pp 88-98. 1968 NATO Conference on Software Engineering, eds. J.M. Buxton, P. Naur, B. Randell, Petrocelli/Charter, Brussels, Belgium, 1976

[Meyer-1986] Meyer, B. Genericity versus Inheritance. Proceedings OOPSLA'86 Conference, ACM SIGPLAN Notices V 21 (11), 391-405, November 1986

[Middelburg-1986] Middelburg, C.A. Programmeren in Logica. Colloquium programmeertalen, pp 39-60 Eds.: Poirters, J.A.A.M., Schoenmakers, G.J. Academic Service 1986

[Neighbors-1984] Neighbors, J.M. The Draco approach to constructing software from reusable components. IEEE Transactions on Software Engineering, V 10 (5), 564-573, September 1984

[Rickert-1986] Rickert, N.W. Preconditions for widespread reuse of code. ACM Sigsoft Software Engineering Notes, V 11 (2), 21, April 1986

[Schwartz-1986] Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., Schonberg, E. Programming with sets: An introduction to SETL. series: texts and monographs in computer science, eds. d. Gries, Springer-Verlag, New York, 1986

[Shaw-1984] Shaw, M. Abstraction Techniques in Modern Programming Languages IEEE Software V 1 (4), 10-26, October 1984

[Tracz-1988] Tracz, W. Software Reuse Myths. ACM Sigsoft Software Engineering Notes, V 13 (1), 17-21, January 1988

[Ververs-1988] Ververs, F., Katwijk, J. van, Dusink, L. Directions in reusing software. Report of the Faculty of Mathematics and Informatics, 88-58 T.U. Delft, the Netherlands, 1988

[Wegner-1984] Wegner, P. Capital-intensive software technology. IEEE Software, V 1 (6), 7-54, July 1984

[Wirth-1974] Wirth, N. On the design of programming languages Programming methodology I, series: Information processing 74 North-Holland publishing company, 1974

HET OBJECT TUSSEN INFORMATION ENGINEERING EN SOFTWARE ENGINEERING

DR. H.G. SOL
Hoogleraar Informatiesystemen
Technische Universiteit Delft

(SCHIJN)TEGENSTELLINGEN

In de traditionele levenscyclus voor de ontwikkeling van informatiesystemen wordt veelal onderscheid gemaakt tussen enerzijds probleemanalyse en systeemontwerp en anderzijds bouw en implementatie. Deze op fasering gerichte tegenstelling tussen ontwerp en bouw van informatiesystemen wordt al in de zestiger jaren door Langefors aangepast naar een inhoudelijk gericht onderscheid tussen 'infologie' en 'datalogie'.
De aanduiding van de vakgebieden Information Engineering (IE) en Software Engineering (SE) heden ten dage lijkt deze tegenstelling voort te zetten. Zelfs zijn vanuit SE pretenties waar te nemen om IE tot haar vakgebied te rekenen, wanneer wordt aangedrongen op een grotere formalisatie van een slecht-gestructureerd gebied als IE. Daarentegen wordt wel gesteld dat door de opkomst van vierde generatie hulpmiddelen (4GT) en werkbanken alleen IE als vakgebied overblijft.
Het lijkt zinvol deze (schijn)tegenstelling eens aan een nader onderzoek te onderwerpen. Daartoe willen we eerst enkele uitgangspunten van het vakgebied IE toelichten.

Informatiesystemen worden ontwikkeld om organisaties efficiënter en effectiever te laten functioneren. Immers, informatiesystemen (IS) zijn en produceren afbeeldingen van reële systemen (RS). Een informatiesysteem omschrijven we daarbij als een geheel van informatieverzamelingen, apparatuur met bijbehorende programmatuur, en van personen met de procedures volgens welke zij werken.
De laatste decennia zien we een verschuiving van transactieverwerkende systemen (TV), via systemen voor management informatievoorziening (MIV) in de richting van systemen voor persoonlijk computergebruik (PCG). Alhoewel we voorzichtig moeten zijn met indelingen van informatiesystemen, is wel een cesuur aan te geven tussen enerzijds transactieverwerkende systemen en systemen voor management informatievoorziening die op basis van de primaire systemen op reguliere basis rapportages verzorgen, en anderzijds systemen voor persoonlijk computergebruik.
Een goed vergelijkingskader voor de beoordeling van informatiesystemen en voor de vergelijking van methodieken en hulpmiddelen wordt geboden door de vragen te stellen:

- welke beheerswijze of projectmanagement-aanpak wordt gehanteerd;
- welke werkwijze of fasering wordt gevolgd, bijvoorbeeld een lineaire of 'hapklare brokken' fasering, een interactieve fasering of een incrementele of 'prototyping' fasering;
- welke afbeeldingswijze of modelleringsaanpak wordt gebruikt, met een onderverdeling in de vraagstellingen: waartoe gaan we informatiseren, wat gaat er om aan informatiestromen, hoe moeten we deze in gegevens- en programmastrcturen vertalen, en met welke technische hulpmiddelen moet een applicatie gerealiseerd worden;
- welke denkwijze of Weltanschauung beheerst de aanpak.

Dit kader kunnen we toepassen op de indeling in informatiesystemen, waarbij dan wel het volgende beeld ontstaat:

	TV	MIV	PCG
soort problemen	(goed) gestructureerde processen	(goed) gedefinieerde informatie voorziening	slecht gestructureerde beslis singsondersteuning
denkwijze	functiegericht	gegevensgericht	probleemgericht
modelleringswijze	deductief, accent op hoe en waarmee	inductief, accent op wat	hypothetisch-inductief, accent op waartoe
werkwijze	lineair	iteratief	incrementeel
beheerswijze	projectgroep	harmonie	participatief

Ook kan aan de hand van dit kader het onderscheid tussen IE en SE worden geanalyseerd:
Uitgangspunt in de denkwijze bij IE is dat informatiebehoeften in principe niet volledig en vaststaand gespecificeerd kunnen worden, omdat een IS een dynamische afbeelding geeft van een RS. Bij SE lijkt dit uitgangspunt principieel anders te liggen.
In de modelleringsaanpak lijkt IE zich te richten op de waartoe en wat vraagstelling, SE op het hoe en waarmee. IE ziet een verschuiving van het waarmee en hoe naar het wat en waartoe.
Binnen SE wordt veelal een lineaire fasering aangehangen, alhoewel er stromingen ontstaan die deze paradigmatische keuze ter discussie stellen en vervangen door een evolutionaire aanpak. Binnen IE lijkt de golfbeweging van een lineaire naar een incrementele aanpak nu terug te keren naar het inzicht dat een goede fasering in een duidelijke procesgang cruciaal is voor met name complexere informatiesystemen.
De projectmanagement aanpak lijkt binnen IE ondergeschikt te worden aan de nadruk op de wat en waartoe vraagstelling. Binnen SE krijgen geïntegreerde omgevingen voor projectmanagement (IPSE) veel aandacht.

NAAR INTEGRATIE

Informatiesystemen mogen nooit verworden tot een doel op zich. De opgave voor IE ligt in de verbetering van de prestaties van informatiewerkers in organisaties door het gebruik van informatietechnologie. De toekomst van informatiesystemen zien we daarbij als netwerken van systemen waar de taken, zowel reguliere als ad-hoc, van informatiewerkers worden ondersteund.

Ook tekenen zich nieuwe soorten informatiesystemen af: systemen voor real-time toepassingen, interorganizationele informatie-systemen, kennis-gestuurde systemen en kantoorsystemen.
Het wordt nodig aansluiting te zoeken bij de primaire processen in de organisatie, die parallel aan elkaar verlopen en waarvan de afhankelijkheid niet op voorhand duidelijk is.
Hierom is het gewenst te komen tot nieuwe wijze om IS te specificeren: een formalisering en vastlegging van componenten die 'purposeful' parallel aan elkaar werken.
Een "object gerichte benadering' moet dit mogelijk maken. Binnen IE zijn de ingrediënten hiervoor reeds lang aanwezig, zie Sol (1982). Binnen SE hoeven we maar te verwijzen naar de rijke historie van object-gerichte programmeertalen. Een integratie ligt dan ook voor de hand, zie Bots en Sol (1988):
In een gemeenschappelijke denkwijze dienen we te starten bij de verbetering van de primaire processen in de organisatie. Informatiewerkers kunnen aanzienlijke ondersteuning krijgen in hun taakuitoefening aangevuld met een betere afstemming en communicatie. Bij deze aanpak wordt een drietal niveaus in beschouwing genomen:
a.micro
Hier gaat het om de functionaliteit op de werkplek, gericht op taak-verbetering van een individu of groep door gebruikmaking van de daar aanwezige kennis en ervaring.
b.meso
Hier gaat het om de inhoudelijke afstemming van de werkplekken door de beschrijving en de verbetering van de processen die voor afstemming zorgdragen of die boven werkplek-niveau uitstijgen. Centraal staan flexibele rapportagemogelijkheden en faciliteiten voor aggregatie en disaggregatie en informatieverzamelingen.
c.macro
Hier gaat het om de strategische positionering van de organisatie in een maatschappelijke sector waar ingespeeld moet worden op de informatische infrastructuur. Het karakter is hier sterk probleemgericht.
Belangrijke overwegingen bij deze denkwijze zijn dat door de nadruk op de beschrijving van de werkplekken een goede herkenning van taken en processen kan worden verkregen. Op basis hiervan worden meetpunten vastgelegd voor taak- en procesverbetering, zowel kwalitatief als kwantitatief. Ook is hier een goede aansluiting voor aanpassing en inlering te realiseren.

NAAR OMGEVINGEN VOOR SYSTEEMONTWIKKELING

Een hierbij aansluitende afbeeldingswijze legt een sterke nadruk op het specificeren van beschrijvende en dynamische modellen op micro-, meso- en macro-niveau, ieder gericht op de vragen waartoe, wat, hoe en waarmee.
Deze afbeeldingswijze gaat uit van de specificatie van reële objecten en artefacten, waar gegevens- en proces-kenmerken gecombineerd worden. De verbanden, die via regels en afhankelijk van de tijd bestaan, dienen hier verifieerbaar en valideerbaar te worden weergegeven. Voor een uitwerking zie Bots en Sol (1988) en Cohen en Sol (1988). Kenmerkend is een declaratieve en tegelijkertijd procedureel vertaalbare specificatie.

De werkwijze, die bij deze denkwijze en afbeeldingswijze aansluit, kent het volgende stappenplan:
- maak taak- en proces-specificaties op de werkplekken;
- simuleer hiermee om van een 'understanding level' naar een 'design level' te komen;
- bevries de dynamische modellen;
- los de afstemming tussen de verschillende werkplekken op meso-niveau op;
- transformeer het ontwerp naar een efficiënte architectuur en dialoogstructuur.

Het is duidelijk dat hier een belangrijke rol ligt voor computer aided systems engineering (CASE) met hulpmiddelen om op basis van schetsen van systeem specificaties applicaties te genereren.
Kenmerkend voor de afbeeldingswijze en de werkwijze is dat uit gegaan wordt van een principieel parallelle specificatie, die vervolgens efficiënt gerealiseerd moet worden op een (semi) parallelle architectuur.

De beheerswijze wordt gekarakteriseerd door een 'middle out' aanpak, door participatie via verankering van de (parallelle) kenmodellen en maakmodellen bij de betrokkenen, en door evoluerend ontwerpen.
Voorzichtigheid zij hier echter geboden voor 'doe het zelf' ontwikkeling, ondanks de beter wordende functionaliteit van vierde generatie hulpmiddelen: de informatische abstractie stelt hoge eisen aan de betrokkenen. Het modelbouw proces blijkt veel belangrijker te zijn dan de hulpmiddelen, zie bijvoorbeeld Wijers en Sol (1988) en Bots en Sol (1988).

Deze denkwijze, afbeeldingswijze, werkwijze en beheerswijze komen te samen in een geïntegreerde ontwikkelomgeving, gevoed niet alleen vanuit IE en SE, maar ook vanuit Communication Engineering, Computer Engineering en Kennis Engineering.
Er lijken voldoende ingrediënten aanwezig om object-gebaseerde, context gebonden ontwikkelomgevingen te realiseren: de geïntegreerde ontwikkelomgeving als object tussen IE en SE. Laat het geen artefact worden.

Referenties

Bots, P.W.G. and H.G. Sol, "Shaping Organizational Information Systems through Co-ordination Support", in: Lee, R.M., A.M. McCosh and P. Migliarese (editors), Organizational Decision Support Systems, Proceedings of the IFIP WG 8.3 Working Conference on Organizational Decision Support Systems, North-Holland, Amsterdam, the Netherlands, 1988, 139-154.

Cohen, M.A.H. and H.G. Sol, "A Simulation Environment for the Development of Information Systems", in: Huntsinger, R.C., W.J. Karplus, E.J. Kerckhoffs and G.C. Steenkiste, Simulation Environments and Symbol and Number Processing on Multi and Array Processors, Proceedings of the European Simulation Multiconference, Nice, 1988, 204-210.

Sol, H.G., Simulation in Information Systems Development, Ph.D. Thesis, University of Groningen, Groningen, The Netherlands, 1982.

Sol, H.G. and M.B.M. van der Ven, "Integrating GDSS in the Organisation: The Case of a GDSS for International Transfer Pricing," in : Lee, R.M., A.M. McCosh and P. Migliarese (editors), Organizational Decision Support Systems, Proceedings of the IFIP WG 8.3 Working Conference on Organizational Decision Support Systems, North-Holland, Amsterdam, the Netherlands, 1988, 129-138.

Sol, H.G. and A.A. Verrijn Stuart, "Information Planning for Personal Computing: Putting the User in Profile", in Jeffery R.: Proceedings of the Joint International Symposium Information Systems, Australian Computer Society, Sydney, Australia, 1988.

Van Schaik, F.D.J., Effectiveness of Decision Support Systems, Ph.D. Thesis, Delft University of Technology, Delft, the Netherlands, 1988.

Van Weelderen, J.A. and H.G. Sol, "The Xpection-project: Development of an Expert Support System for the Maintenance of Boiler Components Operating in the Creep Range", in: Proceedings of the Symposium on Expert Systems Application to Power Systems, Royal Institute of Technology, Stockholm-Helsinki, 1988, 9/15-9/22.

Wijers, G.M., Sol, H.G., Intelligent development environments for Information Systems, Report 87-85, Delft University of Technology, Delft, The Netherlands.

EEN ADVIES VAN EEN INFORMATICA-PIONIER

R. Westermann

Bij gelegenheden als deze ligt het voor de hand om herinneringen op te halen. En hoewel de toekomst belangrijk is, past het in dit geval om het verleden even te doen herleven. Ik wil dit gaarne doen en hiermee een bijdrage leveren aan het afscheid van Prof. W.L. van der Poel als hoogleraar Informatica.

Het is niet overdreven als ik stel dat ik een belangrijk deel van mijn kennis van de automatisering aan Prof. W.L. van der Poel te danken heb. En dit niet omdat hij mij persoonlijk het vak heeft geleerd. Maar wel omdat zijn lessen in de vorm van hoorcolleges en discussies mij duidelijk hebben gemaakt waar het in dit vak om draait.

Wat ook uitermate belangrijk was op het moment dat ik als student wiskunde mij meer en meer tot computerkunde voelde aangetrokken is het enthousiasme en de uitstraling waarmee Prof. W.L. van der Poel zijn studenten begrip voor het vak probeerde bij te brengen.

Ondanks het feit dat door sommigen wel eens aan zijn onderwijskundige capaciteiten getwijfeld werd, omdat hij zich teveel als hobbyist zou manifesteren, sloot zijn manier van onderwijs perfect aan bij mijn eigen opvattingen over hoe een vak onderwezen en geleerd moet worden.

Op de achtergrond was immers altijd de wetenschap aanwezig, dat Prof. W.L. van der Poel zijn sporen op automatiseringsgebied al ruimschoots had verdiend en dat hij op diverse onderdelen van de wiskunde, zoals de combinatorische logica zijn mannetje stond.

Het beste is mij echter het volgende advies bijgebleven, dat nog steeds actueel is en waar eenieder ook nu nog zijn voordeel mee kan doen.

En dat is deze raadgeving, die het best gepresenteerd kan worden als volgt. De raad luidde: "Als je eenmaal de fout in een programma hebt gevonden, waarbij niet direct duidelijk oorzaak en gevolg zijn aan te geven, verspil dan niet je energie met het trachten te reconstrueren hoe het verbeteren van de fout inderdaad heeft kunnen leiden tot het gewenste effect".

Het is evident dat dit advies voortkwam uit ervaring. De praktijk leerde en leert nog steeds dat je inderdaad veel tijd kwijt bent met pogingen tot reconstructie, maar dat die pogingen in veel gevallen op niets uitlopen.

Dit voorbeeld van een pragmatische benadering van het vak programmeren is illustratief voor de instelling waarmee Prof. W.L. van der Poel met het vak omging en zijn studenten onderwees. Deze aanpak sprak mij in ieder geval enorm aan. Het onderschrijft tevens de stelling dat programmeren ondanks alle vorderingen, die gemaakt zijn in de afgelopen jaren nog steeds een ambachtelijke bezigheid is. Vandaar waarschijnlijk dat dit advies, hoewel reeds zo'n vijftien jaar oud, nog steeds niets van zijn waarde heeft ingeboet.

Tot slot wil ik deze bijdrage beëindigen door te onderstrepen dat ik het zeer op prijs stel dat ik op de vraag wie mijn afstudeerhoogleraar in Delft geweest is, kan antwoorden: de informatica-pionier Prof. W.L. van der Poel.

R. Westermann, 16 september 1988, Pijnacker

Het SCARCE project, een scalable architectuur met ondersteuning voor Lisp en Prolog.

Prof. dr. ir. A.J. van de Goor
Drs. H. Corporaal

Vakgroep Computer Architectuur en Digitale Techniek
Faculteit der Electrotechniek
Technische Universiteit Delft

Ontstaansgeschiedenis van het Scarce project.

Tegen het eind van 1985 kwam een viertal zeer enthousiaste studenten bij ons met de vraag of wij een geschikte afstudeeropdracht wisten. De mogelijkheid dat er niet direct een opdracht voorradig was, hadden zij al voorzien. Geïnspireerd door de colleges van prof. Van der Poel betreffende de taal Lisp, deden zij het voorstel om als afstudeerwerk een Lispcomputer te ontwerpen. Onze vakgroep had immers als onderzoekskader het ontwikkelen van architecturen welke effectief hogere programmeertalen ondersteunen. De taal Lisp, zo hadden deze studenten geleerd, is daarbij als studieobject uiterst interessant en wel om twee redenen. Ten eerste is de taal Lisp ondanks (of misschien dankzij) zijn lange ontstaansgeschiedenis de meest gebruikte taal binnen de wereld van de kunstmatige intelligentie; dit vanwege haar elegante en krachtige programmeeromgeving. Daarnaast biedt de taal Lisp voor computerontwerpers een extra uitdaging. De flexibiliteit van de taal maakt haar erg reken- en geheugenintensief. Een efficiënte implementatie van de taal en hardware ondersteuning is daarom erg belangrijk.

In onze vakgroep was net het multiprocessor UNIX project afgerond. De ervaring daarmee, en ook met eerdere architectuurprojecten, leerde dat het ontwerp en de bouw van een compleet computersysteem inclusief benodigde software uiterst nuttig en leerzaam is, maar tevens zeer veel tijd en energie vergt. (Dit laatste ziet men overigens niet terug vertaald in de getallen van de output-matrix van de afdeling.) Doch het viertal studenten liet zich hierdoor niet afschrikken en ging zeer hard aan de slag. Spoedig breidde het aantal studenten zich uit en vormden zij samen de zg. Lispgroep. Later werd de naam SCARCE voor het project gekozen, hetgeen staat voor Scalable Architecture Experiment.

De scope van het project is al in een vroeg stadium verbreed. In samenwerking met de vakgroep theoretische informatica o.l.v. prof. Van Westrhenen en met IBM, die het project met computerapparatuur ondersteunt, is besloten om ook de taal Prolog te ondersteunen. Deze taal heeft een aantal gemeenschappelijke kenmerken met de taal Lisp, zodat ondersteuning van beide talen door een enkele architectuur verantwoord is. Bovendien profileert Prolog zich duidelijk als 2e taal naast Lisp in de kunstmatige-intelligentiewereld. De informatica vakgroep neemt de implementatie van Prolog voor

haar rekening, terwijl op electrotechniek de Lisp-implementatie en de architectuur ontwikkeld worden.

Binnen onze vakgroep was nauwelijks kennis omtrent de taal Lisp aanwezig. Het lag dan ook voor de hand om de enige geschikte persoon binnen de TUD, prof. Van der Poel, die reeds een jarenlange ervaring met deze taal had, te vragen om het project mee op te starten. Prof. Van der Poel heeft met groot enthousiasme deze taak op zich genomen en heeft daartoe veelvuldig de gezamenlijke Lispgroep bijeenkomsten bezocht. Dat daarbij vaak zeer levendige discussies ontstonden die elk tijdsschema tartten, zal mensen die prof. Van der Poel goed kennen nauwelijks verbazen. Bij hem stonden eigenschappen als eenvoud en elegantie zeer hoog in het vaandel geschreven, zowel op het gebied van de taal als van de architectuur. Dit druiste in tegen ontwikkelingen die zeker in het eerste jaar van het project volop in de belangstelling stonden. Enerzijds de opkomst van de taal Common Lisp, een mengsel van een groot aantal bestaande Lisp dialecten, met als uitkomst een taal die door de gemiddelde programmeur niet meer te bevatten was. De taal bevat meer dan 700 primitieve functies, waarbij vele functies ook nog in allerlei variaties kunnen worden gebruikt. Momenteel wordt er internationaal aan standaardisatie van de taal Lisp gewerkt op basis van Common Lisp. Het ziet er naar uit dat het aantal functies drastisch zal worden uitgebreid. Het is niet voor niets dat prof. Van der Poel deze taal dan ook als "een monster" betitelde. Anderzijds was er de ontwikkeling van microgeprogrammeerde machines met vele zeer complexe instructies ter ondersteuning van de genoemde taal Common Lisp. Te noemen zijn de Symbolics 3600 en de Explorer van Texas Instruments. Deze machines vormden een duidelijke afspiegeling van de ontwikkelingen op het taalgebied. Prof. van de Poel, zo bleek keer op keer, herkende zichzelf nauwelijks in deze ontwikkelingen. Boeiend kon hij vertellen over Lisp 1.5, één van de allereerste Lisp standaarden, waarin je met een twintigtal functies toch eigenlijk alles kon doen. Sterker werd zijn betoog nog als het ging over computerarchitecturen. Met name over de PDP-8 met zijn eenvoudige instructieset raakte hij niet uitgepraat. Het toppunt van eenvoud was wel de z.g. 1 instructiemachine. Na zeer lang praten wilde hij dan wel toegeven dat je 1 of 2 instructies zou kunnen toevoegen, maar dan kreeg je toch ook een zeer krachtige machine.

Natuurlijk zal de lezer begrijpen dat prof. van der Poel niet iemand is die wetenschap verwart met nostalgie. Integendeel, hij probeerde ons het bewustzijn bij te brengen dat je van iedere taalconstructie en iedere computerinstructie die je toe wilt voegen een wetenschappelijke verantwoording moet afleggen. Deze toevoegingen zijn meestal niet gratis! Ze kunnen wat betreft de Lisptaal ten koste gaan van de elegantie, eenvoud en implementatie-efficiency, en voor wat betreft de architectuur ten koste van extra hardware, toename van de "cycle time", en eenvoud van ontwerp, waardoor een snelle realisatie bemoeilijkt wordt. In dezen had hij een voorspellende blik, hetgeen uit de volgende meer technische beschrijving van het Scarce project zal blijken.

Ontwikkelingen op taal- en architectuurgebied welke het Scarce project beïnvloeden.

Om een processor te ontwerpen die efficiënt een hogere programmeertaal kan ondersteunen, is een duidelijke analyse van de taal noodzakelijk. Zoals reeds vermeld, wijkt de taal Lisp sterk af van gangbare talen als Pascal en C. Zonder compleet te willen zijn, noemen we de volgende belangrijke kenmerken van Lisp:

1. Runtime binding van variabelen met datatypes. Een gevolg hiervan is dat runtime informatie nodig is om van ieder data-object het type te kunnen herkennen. Deze informatie wordt met z.g. tag-bits vastgelegd.
2. Veelvuldig gebruik van relatief kleine functies.
3. Krachtige ondersteuning van lijstverwerking en symboolmanipulatie.
4. Functies kunnen functies als waarde retourneren. Tezamen met lexical scoping resulteert dit in het niet toepasbaar zijn van de traditionele stack-implementatie zoals die bekend is van Algol-achtige talen.
5. Automatisch geheugenbeheer. De programmeur is gevrijwaard van het expliciet de-alloceren van objecten. Dit wordt automatisch door een z.g. garbage collector gedaan, die een onderdeel vormt van het Lisp runtime systeem.
6. Krachtig functie- en macromechanisme. Functies en macro's maken het mogelijk de taal (dynamisch) uit te breiden. Zowel syntax als semantiek zijn eenvoudig te wijzigen.
7. Samenwerking tussen interpreter en compiler. Geïnterpreteerde en gecompileerde code kunnen doorelkaar gebruikt worden. Hierdoor kan de programmeur de voordelen van debugging en snelle prototyping combineren met executie-efficiency van gecompileerde code.

Recente, nog niet volledig uitgekristalliseerde ontwikkelingen voegen nog een aantal krachtige programmeergereedschappen toe, zoals: object oriented programming, lazy evaluation, streams en coroutines. Voor Prolog dienen nog de belangrijke concepten unification en backtracking genoemd te worden.
In principe kunnen de genoemde taalkenmerken door de processor-architectuur ondersteund worden. Dat is ook de weg die door ontwerpers van CISC machines (complex instruction set computers) jarenlang is bewandeld. Het architectuurniveau wordt daarbij verschoven in de richting van de te ondersteunen programmeertaal. Uitgangspunten hierbij zijn:

1. Compacte codering van programmacode is belangrijk.
2. Compilatie van een programma naar een CISC architectuur is eenvoudig.
3. Een complexe instructie executeert sneller dan een equivalente reeks eenvoudiger instructies.
4. Toevoegen van complexe instructies is vrijwel gratis.

Recent onderzoek naar het instructiegebruik van gecompileerde Lispcode op CISC computers heeft aangetoond dat de z.g. 80/20 regel van kracht is. Dit houdt in dat 80 % van de geëxecuteerde instructies gevormd worden door 20 % van de instructieset, ofwel ingewikkelde instructies worden nauwelijks gebruikt. Het blijkt zeer moeilijk voor

compilers om efficiënt gebruik te maken van complexe instructies. Statistisch onderzoek binnen de Lispgroep heeft bovenstaand resultaat bevestigd en tevens aangetoond dat de 80/20 regel ook voor de Lisp source code zelf geldt. Ook op hardware-gebied zijn veel ontwikkelingen gaande die de CISC benadering in een ander daglicht stellen. Van belang zijn o.a. de onderzoeken naar instruction-pipelining, zeer snelle en grote cache-geheugens en gescheiden programma- en databussen. Tezamen heeft dit geleid tot de z.g. RISC (reduced instruction set computer) filosofie. Als belangrijkste kenmerken hiervan zijn te noemen:

1. Alleen eenvoudige veelgebruikte instructies worden ondersteund door hardware. Toe te voegen instructies mogen de cyclustijd niet verlengen.
2. M.b.v. pipelining wordt getracht om een throughput van 1 instructie per cycle te krijgen.
3. Alle instructies hebben een vast formaat. Hardwired decoding van instructies maakt dan een korte cyclustijd mogelijk.
4. Grote bandbreedte tussen caches en cpu zowel voor data als programma.
5. Intensief gebruik van veel registers vermindert het aantal geheugenoperaties.
6. Een eenvoudig ontwerp maakt een snelle realisatie mogelijk, zodat nieuwe technologieën direct kunnen worden toegepast.

Metingen hebben aangetoond dat RISC processoren de talen Lisp en Prolog efficiënt kunnen executeren. Deze efficiëntie kan nog verhoogd worden door het toevoegen van adequate hardware ondersteuning voor deze talen hetgeen één van de doelstellingen van het Scarce project is.

De RISC filosofie kan worden samengevat met de zin: ondersteun alleen het hoogst noodzakelijke, maar doe dat zo goed mogelijk. Deze benaderingswijze kan ook op Lisp implementaties worden toegepast. Zoals reeds vermeld, kan in Lisp m.b.v. krachtige functie- en macromechanismen de taal eenvoudig worden uitgebreid, zowel wat betreft syntax als semantiek. Omgekeerd betekent dit dat we het Lisp runtime systeem zodanig kunnen opbouwen dat alleen de veelgebruikte operaties, de z.g. basic subset van Lisp, efficiënt ondersteund worden. Minder frequente operaties worden m.b.v. deze basic subset geïmplementeerd. Zo onstaat een laagsgewijze opbouw. Voor nieuwe architecturen behoeft alleen de basic subset opnieuw te worden geïmplementeerd, terwijl door het toevoegen van nieuwe lagen wijzigingen in de Lisptaal eenvoudig kunnen worden overbrugd. In het Scarce project wordt gewerkt met de taal T, een op de universiteit van Yale ontwikkelde Lisp variant. De bijbehorende compiler vertaalt de Lisp code door te beginnen met een source naar source transformatie. Het zo ontstane Lisp programma bevat nog slechts 4 taalconstructies! De compiler kan daarna de programma-analyse beperken tot deze wel zeer kleine subset van de taal Lisp en is daardoor in staat deze analyse zeer grondig uit te voeren. Op deze wijze weet men code te genereren die 1. niet onder doet voor code gegenereerd door "state of the art" Pascal en C vertalers, en 2. de programmeur niet laat betalen voor exclusive Lisp eigenschappen zolang hij ze niet gebruikt. Het moge duidelijk zijn dat zowel op architectuur- als op taalniveau de filosofie van de eenvoud, welke voortvloeit uit de 80/20 regel, met zeer goed resultaat in praktijk wordt gebracht.

Het Scarce project nu en in de toekomst.

Uit het voorgaande blijkt dat het Scarce project sterk is beïnvloed door de RISC benadering. De doelstellingen van SCARCE kunnen als volgt worden samengevat:

1. Ontwerp van een architectuur waarbij de RISC beginselen zeer strict zijn toegepast. In het bijzonder zal bij iedere toe te voegen instructie moeten worden nagegaan of deze resulteert in een snellere programmaverwerking. Het potentiële executievoordeel moet worden afgewogen tegen de extra benodigde hardware.
2. De hardware implementatie van de architectuur dient "scalable" te zijn. Dit houdt in dat bij gelijk blijvende functionaliteit extra hardware een snellere executie mogelijk maakt op een voor de programmeur transparante manier.
3. De functionaliteit kan d.m.v. coprocessoren worden uitgebreid.
4. Voor Lisp en Prolog is de meest noodzakelijke ondersteuning aanwezig.
5. Conventionele talen mogen in hun executie niet gehinderd worden door de extra ondersteuning voor Lisp en Prolog.

De processor-architectuur is reeds ontwikkeld en wordt momenteel in het kader van een SPIN samenwerkingsproject met de firma Pijnenburg (te St. Michielsgestel) in VLSI geïmplementeerd. De gerealiseerde ondersteuning voor de talen Lisp en Prolog blijkt onder meer uit:

1. Frequent voorkomende datatypen worden met tagbits ondersteund. Tag checks worden parallel uitgevoerd.
2. Function calling is zeer snel.
3. Een snel trap mechanisme zorgt voor snelle afhandeling van niet in hardware geïmplementeerde functies.
4. Voor Prolog wordt de zeer veel voorkomende shallow backtrack operatie ondersteund m.b.v. een shadow register set. Kenmerkend voor de gekozen benadering is dat deze shadow registers voor andere talen als gewone registers adresseerbaar zijn.

De extra ondersteuning voor deze 2 talen kan zowel met software als hardware gerealiseerd worden, waarbij natuurlijk alleen in het laatste geval snelheidswinst geboekt wordt.

Naast processorimplementatie wordt gewerkt aan programmeeromgevingen voor Lisp en Prolog, inclusief efficiënte compilers. Wat betreft Lisp is het reeds genoemde T-systeem als uitgangspunt genomen.

Ondertussen heeft het Scarce project een spin-off in een aantal andere projecten gehad, te weten:

1. Het onderzoeksproject naar parallelle logische architecturen.
2. Het garbage collection project.
3. Het "Non Conventional Architecture" project.

Wat betreft de toekomst van het Scarce project wordt gedacht aan de volgende uitbreidingen:

1. Ontwerp van een Memory Management Unit (MMU).

2. Ontwerp van een compleet Scarce systeem inclusief MMU, welke als coprocessor board in een RT-PC geplaatst kan worden.
3. Combineren van meerdere Scarce CPUs tot een VLIW architectuur. VLIW staat hier voor: Very Long Instruction Word. Hiermee kan low level parallellisme worden uitgebuit.
4. Combinatie van meerdere Scarce systemen in een multiprocessor systeem. Uitbreiding van de Scarce architectuur met communicatiefaciliteiten zal hierbij nodig zijn.

Technische Universiteit Delft