Eidgenössische
Technische Hochschule
Zürich

Institut für Informatik

Friedrich L. Bauer
Edsger W. Dijkstra

Zwanzig Jahre Institut für Informatik

19. Oktober 1988

Authors' addresses:

Prof. Dr. Friedrich L. Bauer
Institut für Informatik
Technische Universität München
Arcisstrasse 21
8000 München 2 / Deutschland

Prof. Dr. Edsger W. Dijkstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712 - 1188 / USA

Einleitung

Der vorliegende Bericht enthält Vorträge, die anlässlich der Jubiläumsfeier vom 19. Oktober 1988 *"Zwanzig Jahre Institut für Informatik an der ETH Zürich"* von bekannten Pionieren dieses Fachgebiets gehalten wurden. Alle drei Redner (F. L. Bauer, E. W. Dijkstra und N. Wirth) sind von IEEE mit dem "Computer Pioneer Award", einer Auszeichnung im Range eines Nobelpreises der Naturwissenschaften, geehrt worden.

Dieser Bericht wird der letzte des Instituts für Informatik sein, denn es wird Ende Dezember aufgelöst und in vier neue Institute aufgeteilt werden. Diese vier Institute *für Computersysteme, für Informationssysteme, für Theoretische Informatik und für Wissenschaftliches Rechnen* werden neu einem Departement für Informatik angehören.

Im folgenden wird deshalb ein Blick auf die Vergangenheit, auf die Geschichte der Informatik, und nicht in die Zukunft geworfen, wie es sonst bei den gelben Forschungsberichten üblich ist. Das Institut für Informatik war 1968 als Fachgruppe für Computerwissenschaften von Prof. Heinz Rutishauser gegründet worden. Er löste sich damit vom Institut für Angewandte Mathematik, das seit 1948 unter der Leitung von Prof. Stiefel stand. Das Institut für Angewandte Mathematik hatte weltweite Anerkennung durch den Bau der ERMETH (Elektronische Rechenmaschine der ETH) erlangt und hätte 1988 sein 40-Jahr-Jubiläum feiern können. Es war Rutishauser leider nur während zwei Jahren vergönnt gewesen, seine neue Fachgruppe zu leiten. Weitere Mitglieder der Gründungszeit sind die Professoren P. Läuchli, N. Wirth und C. A. Zehnder. Auch das Institut für Informatik geniesst weltweite Anerkennung durch die von Prof. Wirth geschaffenen Programmiersprachen Pascal und Modula-2.

Prof. Wirth skizzierte am Jubiläumstag in seinem Referat *"Informatik: Vom Aschenputtel zum Dornröschen"* die Entwicklung der Informatik an der ETH Zürich. Der Informatik war um 1970 die Aufnahme in die Gesellschaft der Abteilungen verweigert worden. Heute ist das damals abgewiesene Aschenputtel eine begüterte Prinzessin und erwacht wie Dornröschen. Prof. Dijkstra beschrieb unter dem Titel *"A New Science, from Birth to Maturity"* die Entwicklungsschritte der Informatik in den vergangenen Jahrzehnten und unterstrich dabei vor allem die Unterschiede zwischen Europa und den USA. Prof. Bauer schliesslich wies in seinem Vortrag über *"Informatik und Algebra"* auf die gegenseitige Befruchtung der beiden Fachgebiete hin.

Am Nachmittag des Jubiläumstages präsentierten die verschiedenen Fachgruppen des Instituts (Computersysteme, Informationssysteme, Ingenieuranwendungen, Kommunikationssysteme, Wissenschaftliches Rechnen und Theorie und Didaktik) einem interessierten Publikum verschiedene Arbeiten. Der Tag wurde mit einem Festbankett für Institutsangehörige und Ehemalige auf dem Üetliberg beendet.

Als Organisator des Jubiläums möchte ich allen Institutsmitgliedern und vor allem den drei Rednern herzlich dafür danken, dass sie durch ihren Beitrag zum guten Gelingen des Festtages beigetragen haben. Der Schulleitung der ETH Zürich sei an dieser Stelle auch für die grosszügige finanzielle Unterstützung gedankt.

November 1988 Walter Gander

Die Referenten F. L. Bauer, E. W. Dijkstra und N. Wirth

Inhalt:

Informatik und Algebra

Prof. Friedrich L. Bauer

ÜBERBLICK:

0 Rutishauser

1 Al-Chwarizmi, Leibniz: *Schul-Algebra, Handkurbelmaschine*

2 Kummer, Emmi Noether, Artin, van der Waerden, Hasse: *Axiomatische Algebra, Von der Babbage-Zuse-Maschine zur von Neumann-Maschine*

3 Bourbaki, Birkhoff: *Algèbre universelle, Algebraische Spezifikation*

4 Dana Scott: *Topologische Algebra, Fixpunkttheorie*

5 Die Symbiose Informatik-Algebra

> "In der Algebra kommt man mit einem Funktionsbegriff, der die Zuordnung als das Primäre, die Art der Zuordnung, d.h. das Rechenverfahren als das Sekundäre hinstellt, nicht aus. Man muß vielmehr umgekehrt den Rechenausdruck als das Primäre, die durch ihn gelieferte Zuordnung als das Sekundäre ansehen."
>
> Helmut Hasse

0 Rutishauser

Das Thema *Informatik und Algebra* habe ich mit Bedacht gewählt. Das Jahr 1948 des einen zu feiernden Jubiläums markiert in etwa den Beginn meiner Berührung mit Stiefel und Rutishauser. Insbesondere mit Rutishauser, dem jüngeren der beiden, verband mich dann eine enge Freundschaft und eine fruchtbare Zusammenarbeit, die sich auf numerische Mathematik einerseits, auf Programmiersprachen andererseits erstreckte; wenn ich das mit heutigen Vokabeln umreißen sollte, wären Informatik und Algebra passend. Das Jahr 1968 des anderen zu feiernden Jubiläums liegt dann schon nahe am Todesjahr Rutishausers. 1968 war Rutishauser im Besitz eines Rufes nach München auf eine Professur, die mit der Leitung des Leibniz-Rechenzentrums verbunden war. Es gelang mir im Mai 1968 nicht, ihn zu überreden den Ruf anzunehmen; er nannte mir damals gesundheitliche Gründe und seine Befürchtungen waren, wie sich leider herausstellte, nicht grundlos. Daß er in Zürich verblieb, hat dann dem Aufbau der zunächst Computerwissenschaften genannten Informatik sehr geholfen. Rutishauser stand mir fast so nahe wie Samelson, sein früher Tod hat mich besonders schmerzlich betroffen.

Der Zweiklang *Informatik und Algebra* charakterisierte auch weiterhin mein Leben. Das war nicht zufällig, sondern beruht auf einer mich stark bewegenden inneren Verwandtschaft zwischen diesen Disziplinen, mit der Verbandstheorie als Bindeglied. Davon wird noch zu sprechen sein.

1 Al-Chwarizmi, Leibniz: *Schul-Algebra, Handkurbelmaschine*

Gestatten Sie mir zunächst einen historischen Exkurs. Das Wort Algorithmus, in der Informatik prominent, und das Wort Algebra gehen auf die selbe Person zurück, auf Al Chwarizmi, um 820. In seinem weit verbreiteten (und erst um 1150 ins Lateinische übersetzten) Buch mit dem Titel *Hisab aljabr w'almuqabalah* lehrt er das Rechnen mit den neuen indischen Ziffern; Algebra (wie auch Kabbalah) kommt davon her. *Dixit algoritmi* lautet die

stereotype Schlußformel der einzelnen Kapitel, und daraus entstand *Algorithmus* für jedwedes Verfahren des Arbeitens mit Ziffern. Leibniz beispielsweise gebraucht die Wendung *Algorithmus der Multiplikation*, ebenso gab es Algorithmen der Division und des Quadratwurzelziehens. Später wird der Terminus auch für das Arbeiten mit anderen Zeichen als Ziffernzeichen gebraucht. Die Kryptographie verwendet Algorithmen zur Chiffrierung und Dechiffrierung, die formale Logik arbeitet (auch) algorithmisch mit Wahrheitswerten und mit Formeln. In den fünfziger Jahren unseres Jahrhunderts kommt das Wort dann für das Arbeiten mit Dezimalzahlen (oder Dualzahlen) als Einheit wieder auf. So begegnete es mir zuerst in der Rutishauserschen Prägung vom *Q-D-Algorithmus* 1954. Sowohl in Householders *Principles of Numerical Analysis* von 1953 wie im Buch der Faddeevs kommt übrigens der Ausdruck *Algorithmus* noch nicht vor.

Algebra ist stets auch die Kunst vom Erfinden und Verstehen, also vom Beherrschen von Algorithmen. Dies trifft in der Neuzeit insbesondere auf das zu, was man auf dem Gymnasium "Buchstabenrechnen" nennt. Die Algorithmen, die der 12-jährige oft nur mechanisch ausführt, dienen der Vereinfachung von "Buchstabenausdrücken". Gegenstände der Algorithmen sind also nicht nur, wie in der niederen Arithmetik, bloß der Ring der ganzen Zahlen oder der Körper der rationalen Zahlen, sondern Terme, das heißt arithmetische Ausdrücke, die selbst eine Berechnungsvorschrift mit Zahlen beinhalten. Diese Doppelnatur der Terme wird uns noch beschäftigen müssen. Beim Buchstabenrechnen stehen die Buchstaben für beliebige Zahlen, sie bezeichnen *Variable*. Variable können durch Zahlen, aber auch durch ganze arithmetische Ausdrücke ersetzt werden. Auf die Wahl des Buchstabens, der eine Variable bezeichnet, kommt es nicht an: $(a+b)^2 = a^2 + 2ab + b^2$ und $(p+q)^2 = p^2 + 2pq + q^2$ meinen den gleichen Sachverhalt.

2 Kummer, Emmi Noether, Artin, van der Waerden, Hasse: *Axiomatische Algebra, Von der Babbage-Zuse-Maschine zur von Neumann-Maschine*

Die Algebra der Arithmetik arbeitet also mit Algorithmen für Polynome, für rationale Ausdrücke, für Ausdrücke mit Quadratwurzeln usw. Diese Algorithmen können der vereinfachenden Umformung dienen, etwa der Klammernbeseitigung, oder aber auch der Gewinnung neuer Terme als echte Ergebnisse, wie der Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Polynome. Sofern dabei das Ersetzen von Variablenzeichen ganz in den Hintergrund tritt, spricht man auch von *transzendenten Erweiterungen* der Zahlenbereiche durch *Unbestimmte* oder *freie Erzeugende* und hat mit diesen Termen dann *Zahlen höherer Art*, etwa Polynomringe oder Körper rationaler Ausdrücke, insbesondere aber Ringe von Wurzelausdrücken, die der zu Unrecht so vergessene Ernst Eduard Kummer – bezeichnenderweise im Zusammenhang mit Versuchen zum Beweis des großen Fermat – erstmals auf Teilbarkeitsalgorithmen hin untersucht hat. Von Kummers "Idealen" geht ein gerader Weg zu Emmy Noether und zu der neuen Auffassung von Algebra, die unter ihrem Einfluß van der Waerden, Artin, Hasse in den zwanziger Jahren dieses Jahrhunderts propagiert haben, eine Auffassung, die in der Mathematik ebenso revolutionär war wie in der Physik die Einführung der Quantentheorie. (Bis in die Schule hat sich der Fortschritt jedoch nur teilweise verbreitet; so findet man leider Abiturienten, die nicht wissen, daß man mit Polynomen ebenso rechnen kann wie mit Zahlen).

Evidenterweise besteht ein großer Teil dieser Algebra aus Normalformentheorie; das Auswerten eines instantiierten arithmetischen Ausdrucks, das *Rechnen* im eigentlichen Sinn, fällt als Trivialfall darunter.

Eine weitere Revolution, die die Mathematik schon um die Jahrhundertwende überkam, war die Zulassung anderer Gebilde als Ringe und Körper als gleichberechtigter "Reiche", nämlich Gruppen und Halbgruppen, Verbände, Boolesche Verbände und Boolesche Ringe; auch heterogener Gebilde, etwa Vektorräume

und Vektorverbände, Matrixalgebren und derlei. Sogar Gebilde mit ungewohnten, um nicht zu sagen exotischen Gesetzen, wie Lie-Algebren, wurden hoffähig. Der feine Unterschied zwischen der Spezifikation, dem Typ des Gebildes und dem einzelnen Modell kam mit dem Wort *axiomatische Methode* zum Vorschein. Die Mengenalgebra fügte sich als Theorie gewisser Modelle eines Booleschen Verbandes ein, die Aussagenlogik entpuppte sich als Theorie der Terme im Gebilde Boolescher Verband, mit dem zweielementigen Booleschen Verband der Wahrheitswerte als Grundmodell. Hier treten auch Hierarchien auf: die Grundmenge eines Mengenmodells ist ein Parameter, so wie für Körpererweiterungen der Grundkörper ein Parameter ist.

Als in der Mitte dieses Jahrhunderts ein Zusammentreffen von technologischem Fortschritt (elektronische Schaltkreise) und kriegsbedingten Anreizen (Neutronendiffusion, Überschallströmungen) aus Geräten zur Berechnung und zum Druck von Schußtafeln endlich Universalrechner werden ließ, stand die Algebra nicht an der Wiege. Eher war es die Analysis, die die mathematischen Probleme prägte, die nun praktisch anzupacken waren. Vor allem aber haftete den Universalrechnern ihre Abstammung an, entstanden sie doch als vollautomatisierte Handkurbelmaschinen, Tabelliermaschinen oder Buchungsmaschinen. Gegenstand des Rechnens waren auch nicht mehr einzelne Ziffern, sondern bereits ganze Ziffernblöcke für Dezimalzahlen oder (schon revolutionär!) Dualzahlen bestimmter fester Stellenzahl – die Herkunft aus den Berechnungsformularen haftete diesen Zahlen an. Die Automatisierung folgte im übrigen, auch nicht einfallsreich, zunächst dem seit Jahrhunderten im Schlagwerk von Uhren und im Spielwerk von Drehorgeln angewandten Prinzip der Repetition, allenfalls mit Schleifen innerhalb Schleifen, wie es auch schon die automatische Multiplikationssteuerung von Tschebischeff 1896 und die automatische Divisionssteuerung von Rechnitzer 1902 taten. Dies alles führte dann durch Konrad Zuse zur Verwirklichung der Absichten von Babbage, Ludgate und Torres y Quevedo, die es Jahrzehnte zu früh versucht hatten. Mit Zuse gleichlaufend, jedoch zeitlich später waren die Bemühungen von Stibitz, Aiken und Atanasoff. Die starren Schleifen à la Zuse (Ki-

nofilm!) boten jedoch keinerlei Möglichkeit zur Indexberechnung, d.h. konkret zur Adressenänderung.

Die Algebra war noch nicht gefragt. Es bestand (und besteht) eine fast unüberbrückbare Kluft zwischen solchen drehorgelmäßig automatisierten Handkurbelmaschinen und der Turing-Maschine, diesem gespenstischen Destillat der Gedanken des exzentrischen Alan Mathison Turing: Die Turing-Maschine ist nämlich eine Maschine, die bewußt – zu Beweiszwecken – als Universalrechner konzipiert worden war und die, im vollen Einklang mit der *splendid isolation* der reinen Mathematik, nicht der physikalischen Verwirklichung bedurfte, ja einer solchen Profanierung sich geradezu widersetzte. Kein Algebraiker wird jedoch dem ad-hoc-Aufbau der Turing-Maschine Geschmack abgewinnen können. Daß trotzdem nur einige Jahre später der Schritt von der nicht-universellen Babbage-Zuse-Maschine mit einer starren Struktur endlich vieler, allenfalls geschachtelter Schleifen (wie gesagt, ohne jegliche Möglichkeit zur Adressenänderung) hin zu einem praktisch einsatzfähigen und dennoch universellen Rechner gelang, mutet auf den ersten Blick als schierer Zufall an. Eckert und Mauchly nämlich hatten aus den Erfahrungen mit dem Bau des ENIAC nicht nur gelernt, daß eine flexiblere Ablaufsteuerung notwendig war, sondern auch, daß es wichtig war, die Ablaufsteuerung ebenso schnell zu machen wie die Ausführung der Rechenoperationen. Sie mußte also elektronisch erfolgen, und dazu mußte beim damaligen Stand der Technik das Programm auf einem wiederverwendbaren Speicher festgehalten werden. Dieser Speicher konnte mit dem ohnehin notwendigen Ergebnisspeicher zusammengelegt werden. Nicht mehr war verlangt, und die Schleifenstrukturen bildete man mittels Sprungbefehlen nach. Das war die Situation Mitte 1945.

Ebenso wunderlich mag es heute erscheinen, daß man sogleich zu indizierten Feldern, einer sehr mächtigen Datenstruktur – wir werden darauf noch zu sprechen kommen – griff. Wieder waren eher technologische Gründe als tiefere Einsicht maßgebend – sowohl Quecksilber-Verzögerungsspeicher wie Kathodenstrahlröhrenspeicher legten eine Mitzählung der gespeicherten Impulse und somit einen adressierten Speicher (*random access store*) nahe.

Magnetbänder hätten stattdessen eine schlangenartige Speicherorganisation (*first in - first out*) suggeriert.

Jedenfalls konnte man nun, ob man es wollte oder nicht, das Programm während des Laufs der Rechnung abändern – insbesondere konnten Adressen neu berechnet werden. Es stellte sich erst später heraus, daß dadurch eine universelle Maschine entstanden war. Es ist schwer vorstellbar, daß Eckert und Mauchly von der Sehnsucht nach einer universellen Maschine getrieben waren; vermutlich waren ihnen die entsprechenden Begriffsbildungen der Logiker gar nicht geläufig. Von Neumann hingegen ist zuzutrauen, daß ihn – zumindest unbewußt – diese Absicht bewegte, er war ja mit Turing sowohl persönlich wie in wissenschaftlicher Hinsicht bekannt; mit der Logik war von Neumann überdies durch seine Arbeiten über Beweistheorie in tiefere Berührung gekommen. Jeder Logiker, der Ackermanns Beispiel einer Funktion, die nicht mit primitiver Rekursion berechenbar ist, verstanden hatte, hatte auch ein Gefühl dafür, daß eine endliche Zahl von starren Schleifen à la Zuse oder Aiken zu wenig war.

Dank der Desinformation, die einige Beteiligte: von Neumann, Goldstine, Burks betrieben, werden wir nie genau wissen, ob Eckert und Mauchly oder von Neumann zuerst die Idee des abänderungsfähigen Programms hatten. Dokumentiert ist allerdings, daß von Neumann auf die Adressenberechnung und Adressensubstitution erstmals geführt wurde bei dem Versuch, eine Aufgabe zu programmieren, die eine laufende Änderung des Index einer Reihe von Größen involvierte [Knuth 1970]; es ist aber schwer vorstellbar, daß Eckert und Mauchly nicht auch an solche Aufgaben gedacht hatten. Eine systematische Adressenänderung hatte im übrigen auch Seeber 1945 im ansonsten nicht universellen SSEC (Selective Sequence Electronic Calculator) der IBM (fertig 1947) realisiert.

Jedenfalls, unabhängig von der Prioritätsfrage: nur *peu à peu* wurde ausgesprochen, daß man mit Hilfe der Adressensubstitution eine Universalrechenmaschine gewonnen hatte [Goldstine, von Neumann 1946; Mauchly 1947]. Aiken aber hielt starrköpfig an der logischen Trennung von Befehls- und Ergebnisspeicher fest und sah deshalb für die Adressenberechnung Indexregister vor,

was allenfalls als Mittel zur Geschwindigkeitssteigerung zweckdienlich war. Auch Zuse gibt an, er habe absichtlich den "fehlenden Draht" nicht gezogen – möglicherweise eine nachträgliche Reflektion.

Bei all dem hätte die Algebra – wäre man nur auf den Gedanken gekommen, sich von ihr leiten zu lassen – den richtigen Weg gewiesen. Es ist die Doppelnatur der Terme, einerseits Objekte der Berechnung, andererseits selbst Rechenvorschriften zu sein, die den Schlüssel liefert – abänderungsfähige Programme sind, weit über Indexlauf hinausgehend, Programme, die erst Terme berechnen, die dann zur Ausführung kommen. Wie Samelson und ich einige Jahre später entdeckten, hätte auch bei einer strikten logischen Trennung von Befehls- und Ergebnisspeichern die Ablaufsteuerung nur mit Kellerspeichern versehen werden müssen, wie es für die (Zwischen-)Ergebnisspeicherung ohnehin zweckmäßig war – nach dem bei unserem Logikrechner STANISLAUS angewandten Prinzip. Damit bereits war die Maschine universal, ermöglichte insbesondere allgemeine Rekursion. Wir entwickelten damals eine solche Maschine und bekamen sie auch patentiert [Bauer, Samelson 1957], Anmeldetag war der 30. März 1957. Die Maschine wurde aber nicht gebaut. Später fanden sich Teilzüge unserer Patente sogar in Taschenrechnern.

Die fünfziger Jahre waren also zunächst beherrscht von der sogenannten *von Neumann-Maschine*. Die fehlende Trennung zwischen Befehlsspeicher und Ergebnisspeicher führte in den aufkeimenden Programmiersprachen dazu, daß Elemente der Ablaufstruktur und der Datenstruktur vermengt blieben. Bei FORTRAN war das nicht verwunderlich, aber Sprünge mit berechenbarem Sprungziel (`switch`) waren noch in ALGOL 58 als Eierschalen des Maschinendenkens vorhanden. In ALGOL 60 wurden manche dieser Reste eliminiert, aber nicht alle Flecken waren leicht zu entfernen. Die von Rutishauser propagierte Laufanweisung vermengte in, wie es schien, natürlicher Weise natürliche Zahlen mit der Ablaufstruktur. Dabei traten semantische Ungereimtheiten auf, und hauptsächlich das Bestreben nach semantischer Präzisierung – wenn man will, der Zwang des algebraischen Denkens – führten dazu, daß man zusehends die Trennung

von Ablaufstrukturen und eigentlichen Datenstrukturen vorantrieb. Wirth ging dabei mit PASCAL mit gutem Beispiel voran, hauptsächlich aber kämpfte Dijkstra an dieser Front, mit langsamem, jedoch unaufhaltsamem Erfolg. Die Algebra stand immer noch in der Ecke, aber sie konnte beginnen ihre Tränen zu trocknen.

3 Bourbaki, Birkhoff: *Algèbre universelle, Algebraische Spezifikation*

Typisch für die Programmiersprachen der sechziger Jahre war auch, daß sie alle nur über ein festes Repertoire von Objektgebilden verfügten. Wo dieses nicht ausreichte, setzte man auf vorhandene Programmiersprachen weitere Objektgebilde auf. So entstanden zum Beispiel LOGALGOL und eine Matrixversion auf der Basis von ALGOL 60. Innerhalb solcher vorgegebener Objektgebilde waren allerdings Erweiterungen zu neuen Operationen verfügbar – durch Unterprogramme in der maschinennahen Programmierung ebenso wie durch Prozeduren in den algorithmischen Sprachen.

Im übrigen stand höchstens Tupelbildung (direktes Produkt) zur Bildung neuer Objektmengen zur Verfügung, erstmals in COBOL in Reinkultur (in ALGOL 60 mußte man ersatzweise zu Feldern greifen). In ALGOL 68 wurde dann auch die Union (direkte Summe) zweier Objektgebilde einbezogen. Man hatte sich damit auf zwei Standardkonstruktionen abgestützt, die der Mathematik theoretisch ausreichen mochten. Praktisch genügten sie für problemorientierte Situationen nicht.

In den fünfziger Jahren fand jedoch die dritte Revolution in der Algebra statt: die Hinwendung zur *algèbre universelle*. Die Entwicklung der Kategorientheorie legte es nahe, wichtige allgemeine Begriffe wie Homomorphismus unabhängig vom speziellen algebraischen Gebilde zu definieren und zu untersuchen. Zunächst klebte man zwar an homogenen Gebilden, aber 1970 nahmen Birkhoff und Lipson die Erweiterung auch auf heterogene Gebilde vor. Für die auch praktisch immer dringlicher werdende Notwendigkeit, den Programmiersprachen eine saubere seman-

tische Basis zu geben, kam diese Entwicklung gerade zur rechten Zeit. Sie führte in der Informatik zum Aufkommen eines neuen Stils, der algebraischen Spezifikation eines Objektgebildes. Die Algebra begann aufzuleuchten.

Hier mag ein Beispiel dienen:
Signatur ("Stellenverteilung") $\Sigma = (\underline{m};\, o,\, p(.),\, n(.))$;
homogen, eine nullstellige und zwei einstellige Operationen.

Terme: $o,\, p(o),\, n(p(o)),\, \ldots,\, x,\, p(x),\, n(x),\, p(n(n(p(x)))),\, \ldots$

Gesetze: $p(n(a)) = a$,
$n(p(a)) = a$,
$p(a) \neq a$,
$n(a) \neq a$.

(Aus den ersten beiden Gesetzen folgt übrigens Injektivität von p und von n.) Danach sind beispielsweise die Terme o und $n(p(o))$ zu identifizieren, hingegen dürfen die Terme o und $p(o)$ nicht identifiziert werden. Man geht nun in der Informatik typischerweise über den Blickwinkel der *algèbre universelle* hinaus:

Neben der Festlegung der Eigenschaften, die alle Modelle des Gebildes haben müssen, faßt man ein Erzeugungsverfahren für Modelle ins Auge. Es baut die Menge $W_\Sigma[x]$ der Terme der gegebenen Signatur auf. Relevant für die Informatik ist nämlich nur, was sich prinzipiell durch einen Term berechnen lässt – hierin steckt der tief algebraische Ansatz der Informatik. Die Gesetze der Spezifikation erzwingen nun einerseits, verbieten andrerseits gewisse Klassenbildungen innerhalb der Menge der Terme. Als konkretes Modell dient dann jede Menge von zulässigen Äquivalenzklassen der Terme. Der Einfachheit beschränke ich mich in unserem Beispiel auf Grundterme (ohne Unbestimmte). Unter Benutzung folgender (informell definierter) Termmengen

$$
\begin{array}{lcl}
[o] & : & \{o,\ p(n(o)),\ n(p(o)),\ p(n(p(n(o)))),\ n(p(n(p(o)))),\\
 & & \qquad n(p(p(n(o)))),\ \ldots\}\\
{[p(o)]} & : & \{p(o),\ p(p(n(o))),\ p(n(p(o)))),\ n(p(p(o)))),\\
 & & \qquad p(p(p(n(n(o))))),\ \ldots\}\\
{[n(o)]} & : & \{n(o),\ n(n(p(o))),\ n(p(n(o)))),\ p(n(n(o)))),\\
 & & \qquad n(n(n(p(p(o))))),\ \ldots\}\\
{[p^2(o)]} & : & \{p(p(o)),\ p(p(p(n(o)))),\ p(p(n(p(o)))),\ \ldots\}\\
{[n^2(o)]} & : & \{n(n(o)),\ n(n(n(p(o)))),\ n(n(p(n(o)))),\ \ldots\}\\
{[p^3(o)]} & : & \{p(p(p(o))),\ \ldots\}\\
 & & \vdots
\end{array}
$$

ergeben sich als zulässige Klasseneinteilungen unter anderem

$$[o],\ [p(o)],\ [n(o)],\ [p^2(o)],\ [n^2(o)],\ \ldots \qquad \mathbb{Z}/_{\Sigma} \qquad (1)$$

$$\begin{array}{l}
[o] \cup [p^2(o)] \cup [n^2(o) \cup [p^4(o)] \cup [n^4(o)] \cup \ldots,\\
\quad [p(o)] \cup [n(o)] \cup [p^3(o)] \cup [n^3(o) \cup \ldots \qquad GF_2/_{\Sigma}
\end{array} \qquad (2)$$

Das Modell (1) ("initiales Modell") ist isomorph dem Redukt von $\mathbb{Z}$, dem Ring der ganzen Zahlen, auf Null, Nachfolger- und Vorgängeroperation; das Modell (2) ist isomorph dem Redukt von GF_2, dem zweielementigen Körper – wobei die einstelligen Operationen zusammenfallen. Das oben definierte Gebilde hat offensichtlich mehrere, nichtisomorphe termerzeugbare Modelle, es ist polymorph. Andere Modelle sind isomorph den Restklassenringen von $\mathbb{Z}$ modulo $n, n > 2$.

Alle Modelle, die als Objektstrukturen interessant sind, sind also epimorphe Bilder der Termmenge $W_{\Sigma}[x]$. Wiederum ist als prominentes Instrumentarium der Modellierung eine Normalformtheorie erforderlich. Dieses *Erzeugungsprinzip der Informatik* führt auch dazu, daß die betrachteten Modelle sämtlich abzählbar sind. Damit ist z.B. die Menge aller Drehungen einer euklidschen Ebene kein für die Informatik handhabbares Modell des Gebildes Gruppe – so einfach es auch scheinen mag. Als Analoginstrument ist eine kontinuierlich verstellbare Schraube geeignet – das Erzeugungsprinzip der Informatik führt aber auf eine gewisse Diskretheit der Modelle, die für die Digitalisierung auch erheb-

lich ist. Der Körper $\mathbb{R}$ der reellen Zahlen ist eben dem direkten Wirkungsfeld der Informatik entzogen – mit gutem Grund, da fast alle reellen Zahlen nicht berechenbar sind. Die Menge der berechenbaren reellen Zahlen liegt dagegen im Wirkungsfeld der Informatik, und es sind ihrer immer noch genügend viele.

Typisch für die Informatik ist auch die Begriffsbildung der Kryptäquivalenz von Gebilden, in die Algebra von Garrett Birkhoff eingeführt. Es gibt eine Reihe von Gebilden mit nicht nur verschiedenen Gesetzen, sondern sogar verschiedenen Signaturen, deren Modelle ineinander überführt, "umcodiert" werden können. Solche Äquivalenzklassen von Spezifikationen sollen Archetypen heißen. Der Archetyp Gruppe ist recht geläufig. Im einfachsten Fall kann die Umcodierung, bei Gruppen etwa $a/b \stackrel{\text{def}}{=} a * b^{-1}$ bzw. $a * b \stackrel{\text{def}}{=} a/(e/b)$, $a^{-1} = e/a$, termweise erfolgen, in komplizierteren Fällen muß sie rekursiv definiert werden. Hier stieß die Informatik in eine Lücke, die die *algèbre universelle* gelassen hatte. Klarerweise ist der Informatiker an der Umcodierung selbst viel konkreter interessiert als der Algebraiker, da sie seine algorithmische Argumentation direkt berührt.

Es gibt nun Archetypen, die mächtiger sind als andere, in denen sie nämlich nicht codiert werden können. Schlangen ('queues') und Keller ('stacks') sind recht schwache Archetypen, indizierte Felder sind ziemlich stark. Für diesbezügliche Einzelheiten muß ich auf eine kürzlich in Acta Informatica erschienene Arbeit über kryptäquivalente Spezifikationen verweisen [Bauer, Wirsing 1988].

4 **Dana Scott:** *Topologische Algebra, Fixpunkttheorie*

Terme haben, wie gesagt, eine Doppelnatur: einerseits sind sie nach dem Erzeugungsprinzip Elemente der Objektgebilde (die Unbestimmten sind dabei frei erzeugende Elemente), andrerseits legen sie Berechnungsvorschriften fest (die Unbestimmten sind dabei ersetzungsfähige Variable). In der letzteren Auffassung dienen Terme der Festlegung einer Ablaufstruktur, allerdings nur in trivialer Weise: Sie beschreiben in einer festen endlichen Zahl

von Schritten ohne Wiederholung ablaufende Berechnungen. Aber bereits hier sind subtile Verfeinerungen angebracht, für die die traditionelle Algebra kein Gespür hatte. Beispielsweise ist man gewohnt, einen Term von innen nach außen zu berechnen, weil einem bei einem arithmetischen Term, sofern in ihm die Null nicht vorkommt, nichts anderes übrigbleibt. Aber die Auswertung von Termen über der boolschen Algebra der Wahrheitswerte erfolgt typischerweise am besten von außen nach innen. Es muß also festgelegt werden, welcher Auswertungsmechanismus mit einem Term verbunden werden soll – im Jargon mit Stichworten wie *left innermost, parallel innermost, left outermost, parallel outermost* und anderen bezeichnet. Mehrstellige Operationen, die nicht die Auswertung aller Operanden verlangen, um das Ergebnis festzustellen, heißen nicht-strikt; für sie bietet eine Auswertung von außen nach innen Vorteile. Bekanntestes Beispiel ist die if-then-else-Operation. Interessanterweise hat in diesem Punkt in theoretischer Hinsicht die Algebra der Informatik den Rang abgelaufen: Als Diskriminator-Gebilde bezeichnet die Algebra Gebilde, die die vierstellige Operation d [Grätzer 1964]

$$d(x,y,a,b) \stackrel{\text{def}}{=} \begin{cases} a & \text{falls } x = y \\ b & \text{falls } x \neq y \end{cases}$$

direkt oder durch einen Term ausdrückbar enthalten. d ist eine typische nicht-strikte Operation. Gleichwertigerweise kann man auch die dreistellige Operation t [Werner 1970]

$$t(x,y,z) \stackrel{\text{def}}{=} \begin{cases} x & \text{falls } x \neq y \\ z & \text{falls } x = y \end{cases}$$

verwenden, die, im Unterschied zum heterogenen if-then-else, homogen ist.

Mit Termen allein kommt man aber nicht aus. Man geht nun zweistufig vor und sieht Algorithmen an als Vorschriften zur Erzeugung immer neuer Terme, die man dann entweder sofort (eager evaluation), oder irgendwann, oder so spät als möglich (lazy evaluation) auswertet. Dies schließt ausdrücklich Parallelismus ein und erfordert damit den Übergang von der klassischen Logik zu einer geeigneten Modallogik.

Das gängigste dabei verwendete Erzeugungsschema für Terme ist das der Rekursion, oder in der Sprache der Logiker das des Y-Operators im Lambda-Kalkül. Die Fixpunkttheorie, die dazugehört, ist seit 1970 dank Dana Scott fester Bestandteil der Informatik. Hier spielt die Verbandstheorie als Bindeglied zwischen topologischer Algebra und Informatik ihre besondere Rolle. Es gibt auch Varianten der Rekursionstheorie, die in anderem Gewand auftreten und womöglich weniger beschwerlich sind; auf der maschinellen Seite Kellermaschinen, die direkt die Kellerstruktur der Abläufe bewirken. Es gibt ferner Fragmente der Rekursion, die für viele Zwecke ausreichen, wie etwa die Technik der Invarianten für schwächste Vorbedingungen, die sich in Ansätzen schon in der Flußdiagrammdiskussion von von Neumann und Goldstine findet, aber erst von Dijkstra ausgearbeitet wurde; sie steht wiederum in enger Verbindung mit geeigneten modalen Logiken.

Für die Rekursion ist nun entscheidend wichtig, daß sie nur dann funktionieren, nämlich terminieren kann, wenn durch geeignete nicht-strikte Operationen ein Ausstieg möglich ist. Klassischerweise dient dazu die Fallunterscheidung, sofern `boolean` als Objektgebilde und ein universelles Gleichheitsprädikat verfügbar sind, allgemeiner eine Diskriminatoroperation.

Es mag einleuchten, daß in dieser Auffassung eine klare syntaktische und semantische Trennung in Objektgebilde und ihre algebraische Spezifikation einerseits, in Ablaufstruktur (*control structure*) und ihre Rekursionstheorie andrerseits notwendig ist. Die Algebra beherrscht die erstere Seite. Was über die andere?

Wenn man will, kann man auch Ablaufstrukturen algebraisch spezifizieren. Für Fragmente der Rekursion wurde das öfter getan, u.a. von Broy, Wirsing 1980. Ob es sich praktisch lohnt, ist eine noch offene Frage.

Im Prinzip obsiegt also die Algebra auf der ganzen Linie; sie steckt die (theoretische) Informatik in die Tasche, könnten wir sagen, oder weniger salopp:

Die Algebra steht im Dienst der Informatik, die Informatik wird dabei selbst Gegenstand der Algebra.

5 Die Symbiose Informatik – Algebra

Das ist jedoch nicht die volle Wahrheit. Wie sieht es anders herum aus? Seit Computer verfügbar sind, hat man sich ihrer bedient, um komplizierte algebraische Fragen zu beantworten. Die Fortschritte der Erkennungsalgorithmen (irreführenderweise *Künstliche Intelligenz* genannt) erlauben, mehr und mehr Untersuchungen der Algebra zu mechanisieren. Mehr und mehr steht die Informatik auch im Dienst der Algebra, und die Algebra wird selbst Gegenstand der Informatik.

Wie weit das führt, wohin das geht, weiß heute noch niemand; die praktischen und theoretischen Grenzen liegen im Dunkel. Unübersehbar aber ist schon jetzt, daß eine Symbiose zwischen Algebra und Informatik heraufzieht. Die letztere als Ingenieurwissenschaft, die erstere als Geisteswissenschaft haben eigentümliche Verschiedenheiten; sie tun gerade deshalb gut daran, sich gegenseitig zu unterstützen, zu befruchten und zu achten.

Die Verschiedenheiten gelten natürlich auch gegenüber der gesamten Mathematik. Für den Mathematiker bezeichnend ist die Fähigkeit, ein rein gedankliches, ein abstraktes Gebäude aufzubauen. Die wahre Realität liegt für ihn einzig im Geistigen: Bilder und (Gips-)Modelle werden 'nur' zur Anschauung zugelassen, als didaktische Krücken; aber vor ihrer Verselbständigung wird sofort gewarnt: sie sind eigentlich entbehrlich und insofern unanständig.

Für den Informatiker kennzeichnend ist die Fähigkeit, einen überraschenden Einfall zur effizienten Realisierung eines schwierigen, oft sogar außerhalb seines Bereiches auftretenden mathematisierbaren Problems zu haben. Er ist mit *ingenium*, mit Scharfsinn und Erfindergeist ausgestattet und schöpferisch auf ein konkretes Ziel gerichtet, er will am Ende eine nützliche Tätigkeit einer irgendwie gearteten Maschinerie sehen. Theorie wird als Hilfsmittel zugelassen, aber vor reinen gedanklichen Spekulationen (wie beispielsweise transfinite Induktion) wird sofort gewarnt; bloße Existenzsätze sind nutzlos, weil unproduktiv, und insofern verpönt.

Die Mathematik dient der Erbauung des Menschen an den Früchten seines Verstands. Die Informatik dient der Befreiung des Menschen von der Last der eintönigen geistigen Tätigkeit.

Selbstverständlich gibt es keine reinen Mathematiker in diesem Sinn, wie es auch keine reinen Informatiker gibt: es gibt einen *homo faber* auch in fast jedem Mathematiker, so wie es einen *homo contemplativus* auch in fast jedem Informatiker gibt.

Innerhalb der gesamten Geisteswissenschaft ist eben die Mathematik die einzige *exakt* zu nennende Spielart; sie steht deshalb von den Ingenieurwissenschaften am nächsten der Informatik, der einzigen, die sich mit immateriellem, mit 'physikfreiem' *ingenium* befaßt. Das verbindet Mathematik und Informatik, macht sie zu Geschwistern.

Literatur

[Bauer, Samelson 1957] Bauer, F.L. und Samelson, K.: *Verfahren zur automatischen Verarbeitung von kodierten Dateun und Rechenmaschine zur Ausübung des Verfahrens.* Auslegeschrift 1094019, Deutsches Patentamt

[Bauer, Wirsing 1988] Bauer, F.L. and Wirsing, M.: *Crypt-Equivalent Algebraic Specifications.* Acta Informatica 25, 111-153 (1988)

[Broy, Wirsing 1980] Broy, M. and Wirsing, M.: *Programming Languages as Abstract Data Types.* In: Dauchet, M. (*ed.*): 5ème Colloque sur les Arbres en Algèbre et en Programmation, Lille 1980, p. 160-177

[Goldstine, von Neumann 1946] Goldstine, H.H. and von Neumann, J.: *On the Principles of Large Scale Computing Machines.* In: John von Neumann Collected Works, Vol. 5, Oxford 1963 (lecture, given by von Neumann on 15 May 1946)

[Mauchly 1947] Mauchly, J.W.: *Preparation of Problems for EDVAC-type Machines.* In: Annals of the Computation Laboratory of Harvard University Vol. 16, Cambridge, Mass. 1948 (lecture, given at the Symposion on Large Scale Digital Calculating Machinery, 7-10 January 1947)

[Grätzer 1964] Grätzer, G.: *On the class of subdirect powers of a finite algebra.* Acta Sci. Math. (Szeged) 25, 160-168 (1964)

[Knuth 1970] Knuth, D.E.: *von Neumann's First Computer Program.* Comp. Surveys 2, 247-260 (1970)

[Werner 1970] Werner, H.: *Eine Charakterisierung funktional vollständiger Algebren.* Archiv d. Math. 21, 383-385 (1970)

A new science, from birth to maturity

Prof. Edsger W. Dijkstra

If we put the birth of Computing Science at the advent of the program-controlled computer, the topic is about four decades old. These have been exciting decades and I am grateful for my good fortune of having been involved for most of that period, grateful because this involvement enabled me to observe that whole process of growth from close quarters. My observations cover the whole gamut, ranging – as they do – from the stage that Computing Science hardly existed to the current stage in which Computing Science is a vigorous and flourishing discipline whose academic viability is no longer in doubt. An additional reason for personal gratitude is that, during most of my involvement, my travels frequently took me to the other side of the Atlantic Ocean. Such mobility has been essential for observation "from close quarters" because American and European Computing Science evolved quite differently. We shall return to that difference extensively.

Of course I realize that reducing the globe to just two continents is a rather strong simplification of geography, and that, in a comprehensive study of the history of computing science, much finer distinctions would be needed. But I am not a professional historian and hope that you will allow me to make this geograph-

ical simplification. In the same vein I hope that you forgive me when, for the sake of simplicity, I cut the forty-year period up into four decades, each of them with a rough characterization: we could characterize the fifties as the decade of hardware, the sixties as the decade of syntax, the seventies as the decade of semantics, and the eighties – tentatively, for the decade is still young – as the decade of synthesis.

0 The fifties

My characterization of the fifties as the decade of hardware is not surprising and easily justified. The whole field began with the conception of the program-controlled computer for which von Neumann got the credit that may be due to Turing. Building such a machine so that it was sufficiently reliable was, at the time, at the limits of technology – if not beyond! – and it is only too understandable that, at the beginning, the physical equipment attracted all the attention.

Accordingly, most of the inventions of the fifties were aimed at higher speeds, larger stores, and greater reliability. I mention as examples: transistors and ferrite cores, the parity check, the B-line – later better known as the index register –, the real-time interrupt, automatically managed multilevel store, and the timid beginnings of multiprocessing. Those were the days when the goals could easily be quantified: speeds in number of operations per second, store size in bits, and reliability in MTBF. (The last term, standing for "Mean Time between Failures", strongly suggests that machines continued to be expected to fail frequently.) The significance of this decade is best appreciated by realizing how many of these innovations are still with us, in one form or the other.

In all three technical respects, progress was impressive indeed. To give you one example: I remember that we accepted, almost as a Law of Nature, that machine speeds doubled each year, and all of you know that that amounts to a factor of one thousand over the decade. But there is no such thing as a free lunch, and the price we paid was heavy: programming remained a haphazard activity.

Firstly, programming was hardly seen as a problem; most people felt that programming required no more than accuracy and enough computer access and sufficiently short turn-around time to debug your programs. Secondly, in those equipment-oriented days, programming was viewed as intrinsically tied to a specific machine; consequently, the greatest expert in the cunning exploitation of specific machine features was regarded as the best programmer. To put it bluntly, the distinction between a programmer and a hacker had not yet been made.

For the sake of completeness: it was not a decade of hardware progress only. The fifties saw a number of efforts to ease the coding problem, such as autocoders and FORTRAN, but, true to the spirit of the decade, each of these coding aids was aimed at a very specific machine. Consequently, it remained very hard to see programming as something to which science could contribute. Finally, the fifties gave us the professional societies such as the ACM and the BCS; true to aforementioned spirit of the decade, their names refer explicitly to the equipment. They were *not* scientific societies, but professional ones, in which practitioners could exchange their experience.

1 The sixties

Let us turn our attention to the next decade, the sixties. Its first month witnessed an amazing landmark in the international history of computing, viz. the publication of the ALGOL 60 Report, edited by Peter Naur. Three facts stand out: firstly, ALGOL 60 was truly the outcome of international cooperation in the best sense of the word, secondly, it has never been doubted that it was an achievement of the highest scientific standard, and thirdly, at either side of the Atlantic Ocean, its fate was totally different. At both sides it was instrumental in giving shape to a young science, but the shapes were very different.

ALGOL 60 was a breakthrough by virtue of

0 its block structure

1 its parameter mechanism

2 its inclusion of recursion

3 its inclusion of the type Boolean

4 its "cleanness" (e.g. the absence of special constraints on index expressions)

5 the formal definition of its syntax

6 the nonoperational definition of its semantics (to the extent that it was not immediately clear how to implement it)

7 its stability (due to very few trouble spots)

8 its machine-independence.

Compared to the culture of the day, this is a truly impressive list! Allow me to highlight three seemingly minor points. With its unambiguous identity and scope of local variables, the block structure incorporated a radical improvement over standard mathematical practice in which both identity and scope of dummies are often not defined beyond "but every mathematician knows what is meant". As to recursion, the mathematical definition of a continued fraction was "a fraction whose numerator is an integer and whose denominator is an integer plus a fraction whose numerator is an integer and whose denominator is an integer plus a fraction and so on". The proper definition of a continued fraction is of course "a fraction whose numerator is an integer and whose denominator is an integer plus a continued fraction". [For heaven's sake, don't underestimate the quantum leap of this improvement. A few years ago, I quoted these two definitions of a continued fraction as an example of progress. I later heard that from this example a famous scientist in my audience, Fellow of the Royal

Society and all that, had concluded that I was mathematically incompetent because everyone knew that circular definitions did not make sense.] The introduction of the type Boolean as a first class citizen is a similar milestone: to this very day, the standard mathematician cannot read a Boolean expression without interpreting it as a statement of fact, and for him $0 = 1$ is "wrong", rather than one of the many expressions that has the value false.

ALGOL 60 was conceived at a time that business administration and numerical computations were viewed as the main computer applications, but the mere presence of the ALGOL 60 Report made it immediately obvious that there was more to programming than coding for these two types of applications: some people would have to write the ALGOL compilers! Besides being viewed as data processor and as calculator, the computer became more widely viewed as symbol manipulator as well. The formal definition of the syntax was an invitation to approach the compilation problem in a less ad-hoc manner. Moreover, the combination of block structure, parameter mechanism, and recursion offered the possibility of a clean presentation of algorithms of a novel degree of sophistication. The topic was no longer "trivial" and computing science became conceivable as an academically respectable discipline. ALGOL 60 has been more than a scientific landmark: politically, it has been an indispensable stepping stone for Computing Science on its way to academic respectability.

As said, the two sides of the Atlantic Ocean received ALGOL 60 very differently. The superficial story is that ALGOL 60 was widely received in Europe but rejected in America, where it failed to dislodge FORTRAN. IBM had penetrated the American system of higher education by giving the more prestigious universities its machines almost for free, thereby hooking those universities, while European academia had had the good fortune of having been spared that largesse and had remained free, etc. Though not without a germ of truth, this story is really too superficial, for even American universities can unhook themselves if they really wish to do so. The story of ALGOL 60's different fates needs refinement, and so does its explanation.

In America, where, within a few years, Perlis coined the term

"ALGOL-like languages", the study of formal languages quickly emerged as a scientific topic in its own right and ALGOL 60's main role was to provide the initial paradigms for that scientific inquiry. It was hardly viewed as a vehicle for serious programming. And it is precisely in that last capacity that ALGOL 60 drew most of the European attention. Europe quickly implemented ALGOL 60 in its full glory and used it; Tony Hoare's "quicksort" convinced the last doubters that the inclusion of recursion was more than a whim, "Jensen's Device" showed the full power of the parameter mechanism, and, thanks to ALGOL 60 , programming could, and did, become serious business.

The next decade would drive that difference in attitude home to me. At conferences in the USA, the following conversation became a standard ingredient.

Question:	"And, Dr. Dijkstra, what are you currently working on?"
Answer:	"Programming. "
Reaction:	"Ah, I see: Programming Languages. How interesting."

But I am rushing ahead. Both camps took ALGOL 60 seriously, both viewed it as an opportunity and incentive for scientific investment, but they stressed different aspects, viz. formal languages versus programming. How come?

There was an obvious difference in timing. World War II had left the USA with a thriving economy and industry, while those were a shambles in Europe. By 1960, Europe's recovery was well under way, but powerful computers were still much less common than in America. It is understandable that the need for something that would deserve the name of Computing Science was felt more urgently in America than in Europe. Moreover, the fence around the campus that separates the academic world from the rest of society is traditionally much lower than in Europe. Finally, the legal status of most European universities is such that establishing a new academic discipline usually requires a few decades of lawmaking. Finding out how the techniques of scientific thought could be applied to meet the programming challenge was a less tangible and more distant goal than developing formal language theory:

the latter topic was much more in line with the mathematical tradition of the day and thus a target within reach. Urgency and opportunity thus made it in the sixties a cornerstone of America's budding computing science. So much for the difference in timing.

Another explanation for the different reactions to ALGOL 60 can be found in a traditional difference in technological priorities. Right from the start, American computing has been much more concerned with attaining speed than with reducing equipment, be it circuitry or storage size. There is a very simple economic/technological explanation for this: after World War II, none of the European laboratories had the resources needed for the development of the fastest machines conceivable at the time. But that is only part of the explanation, for there is also a cultural difference, as mentioned in passing by Alice S. Rossi[1] in 1964: "Americans are easily impressed by large numbers.". By the time ALGOL 60 came around, this aspect had already created two completely different computing cultures. I remember a conversation in 1962, in Rome. We were sitting around a coffee table. One American boasted that he had made an "algebraic translator" of 50'000 instructions, only to be immediately outdone by one of his compatriots, whose algebraic translator comprised no less than 80'000 instructions. Peter Naur broke the subsequent silence of awe by remarking that he had written an ALGOL translator of 5'500 instructions, upon which I could outdo him with a compiler of only 2'700 instructions. In short: our yardsticks for achievement measured in opposite directions!

The American priority to speed had two direct consequences. Firstly, American computing all but ignored, as too time-consuming, the closed subroutine, which, right from the start, was the cornerstone of European computing. This attitude would culminate in the design of the IBM/360 whose large number of explicitly named registers actually defied a reasonable implementation of closed subroutines. Secondly, American computing ignored ALGOL 60 as a serious programming vehicle because – rightly or

[1] In fact in a footnote to her article "Equality between the Sexes: An Immodest Proposal".

wrongly – it was felt that ALGOL implementations were too slow.

The sixties were the decade in which automatic computing began to gain academic respectability, and academic departments started to emerge. The American departments were, on the whole, sooner to be institutionalized. They were also more ready to incorporate application areas as part of the departmental responsibility, and to this very day you can find in American universities areas such as numerical analysis and artificial intelligence being a part of the CS Department.

Further American contributions from the sixties were symbolic processing – I mention LISP – and complexity theory. They were gratefully incorporated in the American CS curriculum. Were the American departments premature? Many people felt that way and the question whether Computing Science deserved the name of a science was raised over and over again. Newell, Perlis, and Simon have tried to settle that discussion at the end of the decade by "Facts breed science. Computers exist. Ergo.". Needless to say, this did not end the discussion.

European universities too started to think about computing science. They, too, thought deep and hard about forging an academic discipline worthy of the name. But they were less in a hurry to institutionalize and eventually they embarked with a probably somewhat more conscious design. For an academic discipline to be viable, its areas should be coherent and should mutually reinforce each other; moreover, the material taught should have a staying power of, say, fifty years. For the sake of coherence, and in recognition of the fact that the automatic computer really deserves the name "general purpose", it was generally decided that the application areas had better not be included in computing science. For the sake of staying power, it was generally decided that computing science should dissociate itself from the fickle market place: teaching how to make do with the equipment currently on the market was not the calling of European computing science, and all material with a half-life of five years was banned. In passing I mention that COBOL and FORTRAN were viewed as industrial products, and therefore not taught.

Europe's contribution to computing in the sixties consisted

mainly in improving the art of design, such as design of operating systems, of programming languages and of their implementations. Quite a few of such designs were engineered with a novel delicacy. Furthermore, thinking removed itself from the physical equipment, and programming languages were no longer understood in terms of their implementation. (When Perlis called ALGOL 60 "a very inefficient language", the Europeans in his audience were shocked by his patent lack of separation of concerns.)

To complete the picture of the sixties, in 1968, the existence of the software crisis was admitted at the NATO Conference on Software Engineering in Garmisch-Partenkirchen, and Wirth started the implementation of PASCAL. In 1969, C.A.R. Hoare published his "An axiomatic approach to computer programming" and I circulated my "Notes on structured programming". Let us move on to the next decade.

2 The seventies

I called the seventies the decade of semantics: the interest in syntax and parsing tapered off, Dana Scott's denotational semantics provided the logical foundation for recursion in all its glory, and formal proofs of program correctness entered the picture.

In hindsight, the reduction in interest in syntax and parsing was more than justified. Programs that, on account of complexity of the syntax, are hard to parse mechanically are correspondingly hard to compose correctly; consequently, a straightforward syntax that simplifies the parsing eases the programming task (something the designers of Ada overlooked).

As in the mean time can be expected, the American and European interests in proofs of program correctness differed greatly: the American interest focussed on the mechanization of a posteriori verification of given programs, written in given languages; the European interest was more in a constructive approach to the problem of program correctness, and turned to design methodologies or calculi for the derivation of programs that would be correct by construction. From the European perspective, a posteriori program verification amounted to putting the cart before the horse;

from the American perspective, the constructive approach of the Europeans was hopelessly idealistic, because it was mathematical. This was the decade in which I began to stress that as scientists we should clearly distinguish between the intrinsic problems of automatic computing and the problems caused by shortcomings of the American educational system, which traditionally does not consider intellectual advancement its primary concern.

On the American academic scene, complexity theory continued to flourish; automatic theorem proving attracted a lot of attention and got off the ground. A serious concern during the seventies was how to prevent a sizeable part of Computing Science from deteriorating into the dishonourable art of "How to live with the IBM/360"; in winning this battle, the less infected and intellectually more autonomous universities in Europe played a considerable role. Finally, the advent of the personal computer revived most of the mistakes of the fifties – but now on a more grandiose scale –. Today, the ubiquitous personal computer has created an equally ubiquitous misunderstanding of what computing science is about. (At a party, two years ago, Tony Hoare and I were approached by a mathematician who expressed his delight in meeting two such outstanding computing scientists for he had never been able to understand what he had to do in order to save a file on his personal computer, model so-and-so. We could not help him and Tony asked him, whether he could recommend a book on category theory; he could not help Tony either.)

The seventies were a decade of frantic and expanding activity; at the same time it revealed symptoms of decay and showed that in the academic community some intellectual rot was setting in. Also in this respect, America was leading: it became highly productive in rather uninspiring papers, ranging from boring to utterly foolish. Soon, Europe would follow this example.

In the seventies, universities all over the Western world had a hard time. Firstly, lecturing style deteriorated when chalk and blackboard were ousted in favour of the overhead projector with prepared foils: instead of lectures, students got presentations. Secondly, life on campus had been totally disrupted by the student revolts of the late sixties, which breathed an anti-intellectualism

as vigorous as Chairman Mao's Cultural Revolution of which they were the echo. All disciplines have suffered from these two calamities; we must bear in mind that they hit Computing Science at a very vulnerable stage, and there are reasons to suspect that American computing suffered even more than its European counterpart.

The American universities are generally praised for the fact that they are much less a world apart and that the barriers to communication across the campus boundary are much lower than in Europe. It has its charms, but the price can be heavy. For a university to be leading, it must offer what society needs, rather than what society demands. The lower barriers make that the direct demands of society are heard much more loudly and it is not surprising that American computing science, on the whole, became more led than leading.

I am not referring to the local banker's pressure for more "Advanced COBOL" in the curriculum: all but the very smallest institutions can resist such pressure. I am referring to the conflicting pressures from a society that is traditionally ambivalent about technology, that welcomes gadgets almost without restriction but equates technological expertise with the loss of innocence that is an essential ingredient of The American Dream. In the case of Computing Science I am referring to the pressures from a society that is structurally unable to include in its vision a view of programming as a branch of formal mathematics and applied logic, and that therefore is forced to ask for snakeoil. If, then, the campus is insufficiently shielded from those pressures and the market forces are given too free a reign, you can draw your conclusion: a flourishing snakeoil business, be it in "programming by example", "object-oriented programming", "natural language programming", "automatic program generation", "expert systems", "specification animation", or "computer-supported co-operative work". (I am not inventing these slogans: they are all honest quotations.) I used the term "decay" and am forced to conclude that I did not exaggerate.

A more universal problem is presented by the negative effects of academic institutionalization: in the seventies, they became quite

visible for computing science. Firstly, the ridiculous fragmentation: these days we see special professors in "local-area networks", "compiler construction" , or "support environments". Secondly, the diversion into elaborate trivia, caused by the Ph.D. mechanism that requires a steady stream of well-delineated and obviously solvable problems (if you care for them). Thirdly, the need for an outlet for your less than gifted students (which has introduced such topics as "human factors", "software metrics" and "experimental computing science"). As computing scientists, we don't need to feel particularly guilty: no academic discipline has found a truly satisfactory solution for the education of the unavoidable second- and third-rate researchers. Hence, we usually ignore the dilemma, or deny its existence, but it is a negative effect of academic institutionalization.

3 The eighties

Let us now switch to the current decade. I shall be less elaborate on that one for two obvious reasons: firstly, it is not over yet, and, secondly, it is still too close to have established its significance.

European computing science suffered from more than the traditional problems caused by institutionalization, because many departments were founded in such a hurry that quite a few vacancies were filled by so-called "instant professors". Moreover, many a government forced its universities into co-operative efforts with industry (a trend from which Computing Science suffered more than, say, Assyrian Linguistics). As a result, even genuine research potential has been diverted to the ephemeral, shallow, or foolish. Confining myself to Computing Science proper, I shall ignore these aberations.

Tentatively, I called the eighties the decade of synthesis. I did so because Computing Science and Formal Mathematics got more and more intertwined, and did so in a variety of ways.

Firstly, mechanical theorem proving or proof verification came of age. It has definitely left the youthful stage of toy problems: successful applications have ranged from deep theorems such as Gödel's Theorem, to practical challenges such as proving the cor-

rectness of circuitry and special-purpose operating systems. The appearance of a journal dedicated to Automated Reasoning is a clear signal.

Secondly, the fundamentally close connection between proving and programming, as revealed by Constructive Type Theory, is beginning to have its impact: programs are deduced from constructive existence proofs and vice versa.

Thirdly, logic programming has established a visible link between proving and computing, a link that also emerges in the application of complexity theory to the notion of provability. The discovery of the existence of correct theorems that shall never be proved because the price of the proof is prohibitive – and that is putting it mildly – caused some rethinking about the scope and purpose of mathematical reasoning.

Fourthly, it is beginning to be realized, mainly on campus but even in some pockets in industry, that the programming task presents a fertile field for the application of the techniques of scientific thought. The emergence of the journals "Science of Computer Programming" at this decade's beginning and "Structured Programming" at its end are clear symptoms. It is fascinating to observe how this development works both ways and how, in the wake of Programming Methodology, the somewhat wider topic of Mathematical Methodology is emerging. [Since this was written in first draft, Springer announced yet another journal: "Formal Aspects of Computing", with the telling subtitle "The International Journal of Formal Methods".]

It is fascinating to observe how a breakthrough in our manipulative abilities has created the opportunity of realizing an increasing portion of Leibniz's Dream of presenting calculation, i.e. the manipulation of uninterpreted formulae, as an alternative to traditional mathematical reasoning. Because the Dream of Leibniz aims at providing an alternative for traditional mathematical reasoning, traditional mathematics did not provide the most hospitable environment for its realization; this, therefore, had to take place in a separate discipline, which is now known as Computing Science.

It is not surprising that the computing scientist picked up this

gauntlett. By virtue of its mechanical interpretability, each programming language presents a formal systems of some sort, and so the notion of a formal system is something he grew up with. He is very familiar with formal methods because they provide the only sufficiently reliable way of designing programs. Knowing, for instance, how compilers work, he is familiar and totally at ease with the idea of manipulating uninterpreted formulae. Finally, he has good reason for not sharing the common fear of symbol manipulation because he has the tools for mechanization at his disposal.

It is my conjecture that meeting the challenge, embodied in the Dream of Leibniz, will provide the ultimate justification for the existence of Computing Science as a scientific discipline in its own right. We have to show that it is possible to be precise and complete without making a mess of it.

Let me quote Leibniz to wish us well for the next forty years and beyond: "Calculemus ! " .

I thank you for your attention.