Artificial Intelligence – Eine Einführung

Von
Dr. J. Retti, Universität Wien
Dr. W. Bibel, Technische Universität München
Prof. Dr. B. Buchberger, Universität Linz
Dipl.-Ing. E. Buchberger, Universität Wien
Dr. W. Horn, Universität Wien
Dipl.-Ing. A. Kobsa, Universität Wien
Dr. I. Steinacker, Universität Wien
Prof. Dr. R. Trappl, Technische Universität Wien
Dr. H. Trost, Universität Wien

B. G. Teubner Stuttgart 1984

Leitfäden der angewandten Informatik

Herausgegeben von

Prof. Dr. L. Richter, Dortmund
Prof. Dr. W. Stucky, Karlsruhe

Die Bände dieser Reihe sind allen Methoden und Ergebnissen der Informatik gewidmet, die für die praktische Anwendung von Bedeutung sind. Besonderer Wert wird dabei auf die Darstellung dieser Methoden und Ergebnisse in einer allgemein verständlichen, dennoch exakten und präzisen Form gelegt. Die Reihe soll einerseits dem Fachmann eines anderen Gebietes, der sich mit Problemen der Datenverarbeitung beschäftigen muß, selbst aber keine Fachinformatik-Ausbildung besitzt, das für seine Praxis relevante Informatikwissen vermitteln; andererseits soll dem Informatiker, der auf einem dieser Anwendungsgebiete tätig werden will, ein Überblick über die Anwendungen der Informatikmethoden in diesem Gebiet gegeben werden. Für Praktiker, wie Programmierer, Systemanalytiker, Organisatoren und andere, stellen die Bände Hilfsmittel zur Lösung von Problemen der täglichen Praxis bereit; darüber hinaus sind die Veröffentlichungen zur Weiterbildung gedacht.

Dipl.-Ing. Dr. techn. Johannes Retti

Geboren 1952 in Innsbruck. Von 1972 bis 1984 Studium der Informatik an der Technischen Universität Wien, 1984 Promotion. Seit 1979 am Institut für Medizinische Kybernetik der Universität Wien als Assistent tätig, seit 1981 Universitätslektor an der Universität Wien.

Dr. Wolfgang Bibel

Geboren 1938 in Nürnberg. 1964 Diplom für Mathematik, Universität München. 1968 Dr. rer. nat. (Mathematik, Physik, Philosophie) an der Universität München. Seit 1969 wiss. Assistent am Institut für Informatik der Technischen Universität München. Gastprofessuren an der Wayne State University, Detroit, USA, der Universität des Saarlandes, der Universität Karlsruhe und der Universität Rom.

o. Univ.-Prof. Dr. Bruno Buchberger

Geboren 1942 in Innsbruck. 1966 Dr. phil. (Mathematik, Nebenfach Physik), 1973 Habilitation (Mathematik) an der Universität Innsbruck. Seit 1974 o. Univ.-Prof. für Mathematik an der Universität Linz, Einrichtung des Studienschwerpunktes „Computerunterstütztes Mathematisches Problemlösen". Forschungsaufenthalte bzw. Gastprofessuren am Kernforschungsinstitut in Dubna (Moskau), Dpt. of Computer Science der University of Delaware und der University of Wisconsin-Madison (USA), Instituto di Matematica, Università di Genova (Italien).

Dipl.-Ing. Ernst Buchberger

Geboren 1957 in Wien. Von 1975 bis 1981 Studium der Informatik an der Technischen Universität Wien. Ab 1981 wissenschaftlicher Mitarbeiter am Institut für Medizinische Kybernetik der Universität Wien, seit 1982 Universitätslektor.

Dipl.-Ing. Dr. techn. Werner Horn

Geboren 1953 in Villach. Von 1971 bis 1977 Studium der Informatik an der Technischen Universität und Universität Wien. 1983 Promotion zum Doktor der Technischen Wissenschaften. Seit 1978 als Universitätsassistent am Institut für Medizinische Kybernetik der Universität Wien tätig. Seit 1980 Universitätslektor für die Gebiete „Artificial Intelligence und ihre Anwendung in der Medizin" und „Expertensysteme".

Dipl.-Ing. Mag. Alfred Kobsa

Geboren 1956 in Linz. Von 1975 bis 1980 Studium der Informatik und Betriebs- und Verwaltungsinformatik an der Universität Linz. Von 1980 bis 1982 Studium der Kognitiven Psychologie, Linguistik, Logik und Wissenschaftstheorie an der Universität Salzburg. Von 1980 bis 1981 Assistent am Institut für Informatik der Universität Linz. Derzeit Mitarbeiter am Projekt „Sprachverstehende Systeme" des Instituts für Medizinische Kybernetik der Universität Wien.

Dipl.-Ing. Dr. techn. Ingeborg Steinacker

Geboren 1953 in Innsbruck. Von 1972 bis 1974 Studium der Mathematik an der Universität Innsbruck. Von 1974 bis 1984 Studium der Informatik an der Technischen Universität Wien, 1984 Promotion. Seit 1980 wissenschaftliche Mitarbeit am Institut für Medizinische Kybernetik der Universität Wien, seit 1983 Lektor an der Universität Wien.

o. Univ.-Prof. Dr. Robert Trappl

Geboren am 16. Jänner 1939 in Wien. Ing. (Elektrotechnik), Dr. phil. (Hauptfach Psychologie, Nebenfach Astronomie), Diplom aus Soziologie des Instituts für Höhere Studien. 1971 Habilitation für Biokybernetik und Bioinformatik, seit 1977 ordentlicher Professor für Medizinische Kybernetik und Vorstand des gleichnamigen Instituts an der Universität Wien. Universitätslektor für Mathematik an der Technischen Universität Wien.

Dipl.-Ing. Dr. techn. Harald Trost

Geboren 1952 in Wien. Von 1970 bis 1983 Studium der Informatik an der TU Wien. 1983 Promotion. Seit 1978 Assistent am Institut für Medizinische Kybernetik der Universität Wien. Seit 1981 Universitätslektor.

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Artificial intelligence: e. Einf. / von J. Retti
... – Stuttgart: Teubner, 1984.
(Leitfäden der angewandten Informatik)
ISBN 3-519-02473-X

NE: Retti, Johannes [Mitverf.]

Printed in Germany
Gesamtherstellung: Zechnersche Buchdruckerei GmbH, Speyer
Umschlaggestaltung: W. Koch, Sindelfingen

8 Automatisches Programmieren

Bruno Buchberger

8.1 Problemstellung

Programmieren (algorithmisches Problemlösen) ist die Tätigke
die von einer Problembeschreibung (Problemspezifikation) zu ei
auf einer Maschine ausführbaren Algorithmus (Programm, Lösungsv
fahren) führt. Zu Beginn des Computerzeitalters vor nunmehr ca.
Jahren mußten alle Schritte beim Programmieren von der Probl
spezifikation bis zum Programm in der Internsprache
verwendeten Computers vom Menschen ausgeführt werden.
Entwicklung der Informatik seither kann wesentlich durch
Fortschritt charakterisiert werden, der bei der Unterstützung
Programmierens durch den Computer selbst erzielt wurde. Immer me
Teilschritte des Programmiervorganges werden als Routinevorgän
erkannt und dementsprechend als vom Computer durchführbare Au
gaben dem Menschen abgenommen, sodaß sich der menschliche Proble
löser immer mehr auf wesentliche, kreative, höhere, zentralere
universellere Aspekte des Problemlösens konzentrieren und be
schränken kann.

Das Gebiet des automatischen Programmierens (automatic pro
gramming) ist jener Teil der Informatik, der die Entwicklung imme
ausgefeilterer Methoden für diese Computer-Unterstützung des Pro
grammiervorganges zum Ziele hat. Sehr frühe Stadien de
Computer-Unterstützung des Programmiervorganges (z.B. Assemble
und Compiler für ALGOL-ähnliche Sprachen) zählt man allerdings
als heute selbstverständliche Bestandteile einer Programmierum
gebung - im heutigen Sprachgebrauch nicht mehr zum Gebiet des
automatischen Programmierens. (Vergleiche jedoch frühe Arbeiten
zum Compilerbau, in denen Compiler oft als "automatische
Programmiersysteme" bezeichnet wurden).

Wenn man künstliche Intelligenz als jenen Teil der Informatik be
trachtet, der sich mit der Computer-Realisierung von Verhaltens
weisen befaßt, die als "bisher dem Menschen vorbehalten" erschei
nen, dann muß man automatisches Programmieren als Teil der
künstlichen Intelligenz betrachten, weil das Programmieren (in dem
allgemeinen Sinne von "algorithmischem Problemlösen") sicher einer

der anspruchsvollsten menschlichen Aktivitäten ist, wenn nicht überhaupt geradezu das Paradigma intelligenten Problemlösens. (Die Angst, daß durch die Automatisierung, besser durch die "Computer-Unterstützung" des intelligenten Problemlösens der Mensch "automatisiert" wird, ist unbegründet. In Übereinstimmung mit alten Traditionen ist intelligentes Aktivsein nur ein Aspekt der menschlichen Existenz und bewußtes Ruhigsein der andere.) Im Sinne dieser Eingliederung des automatischen Programmierens in die künstliche Intelligenz bildet das automatische Programmieren zusammen mit den automatischen Beweisen, heuristischen Methoden des Problemslösens, Inferenz- und Induktionsmethoden etc. einen Bereich, der zu den anderen vier oder fünf Bereichen der künstlichen Intelligenz wie natürlichsprachliche Systeme, Expertensysteme, Erfassen von Bildern, Robotertechnik, Simulation natürlicher Intelligenz hinzutritt und mit diesen in vielfältiger Beziehung steht.

Freilich kann man organisch automatisches Programmieren auch als Teil der Softwaretechnologie betrachten, insbesondere des Gebiets, das man jetzt oft mit CAS (Computer-Assisted Software Design) bezeichnet, siehe zum Beispiel (Hesse, 1981). Zur Abgrenzung ist es in diesem Zusammenhang allerdings üblich, daß man mit "automatischem Programmieren" eher diejenigen Bereiche von CAS meint, die den formal und logisch tieferliegenden Teil der Entwicklung von korrekten Algorithmen zu Spezifikationen betreffen, während man andererseits im Bereich des CAS mehr an Fragen der Computer-Unterstützung der Entwicklung großer Software-Systeme durch organisatorische Maßnahmen (z.B. Computer-unterstützte Dokumentation, Menü-gesteuerte Programmstrukturierung etc.) interessiert ist.

Schließlich kann man automatisches Programmieren auch als Teil des symbolischen und algebraischen Rechnens (Symbolic and Algebraic Computation, Formula Manipulation, Symbolic Mathematics, Computer-Algebra) betrachten. Symbolisches und algebraisches Rechnen befaßt sich traditionsgemäß (d. h. seit ca. 20 Jahren) mit der algorithmischen Behandlung von Problemen bei symbolischen und

algebraischen (also nicht-numerischen) Objekten. Symbolische Objekte sind dabei

Terme,
Formeln und
Programme,

d.h. sprachliche Objekte, die sich durch ihre Semantik (Bedeutung) voneinander unterscheiden lassen:

Terme bezeichnen bei Belegung ihrer Variablen ein Objekt,
Formeln bezeichnen bei Belegung ihrer Variablen einen Sachverhalt,
Programme ergeben für jede Belegung ihrer Variablen eine andere Belegung.

Dementsprechend betrachtet man oft

Computer-Algebra (algorithmische Behandlung von Termen),
automatisches Beweisen (algorithmische Behandlung von Formeln),
automatisches Programmieren (algorithmische Behandlung von Programmen)

als die drei Hauptgebiete des symbolischen und algebraischen Rechnens. In letzter Zeit sieht man jedoch immer deutlicher, daß sich diese drei Gebiete nicht voneinander trennen lassen und in vielen Aspekten eine Einheit bilden.

Es gibt zwei grundsätzliche Wege, um den Programmiervorgang durch den Computer zu unterstützen, die sich durch Einführung der Ebene einer abstrakten Maschine zwischen die Ebene der Problemspezifikation und der Hardware-Maschine organisch ergeben:

Problemspezifikation
↓ T
Programm für abstrakte Maschine
↓ S
Programm für konkrete Hardware-Maschine

Durch Einführen der Ebene einer abstrakten Maschine wird der Weg von der Problemspezifikation zum Programm für die konkrete Hardware-Maschine in zwei große Teilschritte zerlegt:

T. Transformation der Problemspezifikation in ein Programm für die abstrakte Maschine,

S. Simulation der Exekution des Programms für die abstrakte Maschine auf der konkreten Hardware-Maschine (durch Compiler und Interpreter).

Der Programmiervorgang kann nun computer-unterstützt bzw. automatisiert werden:

entweder durch die Entwicklung von immer höheren abstrakten Maschinen, d.h. von immer höheren Programmiersprachen mit zugehörigen Compilern und Interpretern, die den Transformationsweg von der Problemspezifikation zum Programm für die abstrakte Maschine immer kürzer und leichter werden lassen;

oder durch Computer-Unterstützung des Transformationsvorganges von der Problemspezifikation zum Programm für die abstrakte Maschine.

Die Entwicklung immer höherer abstrakter Maschinen bzw. immer höherer Programmiersprachen hat mit den "logischen Programmiersprachen (Abschnitt 8.2) einen gewissen Abschluß gefunden: logische Programmiersprachen benutzen die Problemspezifikation als Programm. Der "Programmiervorgang" schrumpft im wesentlichen auf den "Spezifiziervorgang" bzw. das Ableiten von "algorithmisch brauchbarem Wissen". Die Exekution solcher Programme verlangt aber dementsprechend hochentwickelte Mittel, die im wesentlichen in der Anwendung eines automatischen Beweisers zur "Exekution" der Programme bestehen.

Für die Computer-Unterstützung des Transformationsvorganges von der Problemspezifikation zum Programm für die abstrakte Maschine gibt es im wesentlichen drei Pradigmen:

"Automatische" Programmsynthese (Abschnitt 8.3)

Gegeben: Grundwissen,
Problemspezifikation.

Gesucht: Programm,
sodaß das Programm die in der Problemspezifikation spezifizierten Eigenschaften hat (unter Voraussetzung des Grundwissens)

"Automatische" Programmtransformation (Abschnitt 8.4):

Gegeben: Grundwissen,
Programm'.

Gesucht: Programm",
sodaß das Programm" mit dem Programm' äquivalent ist, das Programm" jedoch (relativ zu einem bestimmten Kriterium von "Güte") besser ist als das Programm'.

"Automatische" Programmverifikation (Abschnitt 8.5):

Gegeben: Grundwissen,
Problemspezifikation,
Programm für abstrakte Maschine.

Frage: Hat das Programm die in der Problemspezifikation spezifizierten Eigenschaften (unter Voraussetzung des Grundwissens)?

"Automatisch" ist hier und im folgenden immer im Sinne von "automatisch oder computer-unterstützt in Interaktion mit dem menschlichen Problemlöser" zu verstehen. Programmieren in sehr hohen Sprachen, automatische Programmsynthese, -transformation und -verifikation sind für die Zukunft als in Programmierumgebungen zusammenspielende Alternativen, nicht als sich gegenseitig ausschließenden Konkurrenten zu betrachten (Abschnitt 8.6).

Literaturhinweise: Zum Gesamtgebiet des automatischen Programmierens findet sich in (Barr, Feigenbaum 1982), Kapitel X, eine Einführung, die sich hauptsächlich auf die Beschreibung exis-

tierender Software-Systeme konzentriert, während wir hier versuchen, die wesentlichen Basisideen an Beispielen zu demonstrieren. Für eine Übersicht über die Computer-Algebra siehe (Buchberger, Collins und Loos, 1982). Über automatisches Beweisen siehe das Kapitel über automatische Inferenzmethoden in diesem Buch.

8.2 Logisches Programmieren

Zu finden sei ein Programm zur Lösung des folgenden Problems:
    Gegeben: M (eine "Zuordnung").
    Gesucht: V,
    sodaß   V ein "Vertretersystem" für M ist.
(Verwendete Definitionen:
    M ist eine "Zuordnung" genau dann, wenn
        M bildet A in die Potenzmenge von B ab.
    V ist ein "Vertretersystem" für M genau dann, wenn
        V bildet A in B injektiv ab, sodaß
        für alle $a \in A$: $V(a) \in M(a)$.
Hier sind A und B "beliebige, aber fixe endliche Mengen" (A und B sind "global")).

(Eine mögliche Interpretation als "Heiratsproblem":
    A ..... Menge von Burschen,
    B ..... Menge von Mädchen,
    M(a) ... die mit a befreundeten Mädchen,
    V(a) ... die von a "Auserwählte".
Also:
    $V(a) \in M(a)$ ... die Auserwählte von a ist eine der Befreundeten von a,
    V injektiv: zwei verschiedene Burschen können nicht dieselbe Auserwählte haben).

Das "Programmieren" (im Sinne von "algorithmisches Lösen") eines solchen Problems zerfällt nun in zwei Teile:

Der kreative Teil: Finden von "algorithmisch brauchbarem Wissen".

Der Routineteil: Transformieren dieses Wissens in ein Programm für die zur Verfügung stehende Hardware-Maschine.

Die Transformation in ein Programm für die Hardware-Maschine sollte als Routinearbeit wohl möglichst selbst einem Computer überlassen werden können. Um dieses Ziel zu erreichen, sollte algorithmisch brauchbares Wissen möglichst in der Form, wie es als mathematisches Wissen abgeleitet (bewiesen) wird, auch bereits als Programm verwendet werden können. Die dazu notwendige abstrakte Maschine muß dazu einige Fähigkeiten haben, die weit über die Fähigkeiten von Hardware-Maschinen in ihrer "nackten" Form hinausgehen. Wir zeigen das Wesentliche an obigem Beispiel.

Um eine Idee für algorithmisch brauchbares Wissen zu bekommen, betrachtet man, wie die Objekte, die als Parameter in das Problem eingehen, aus kleineren Objekten zusammengebaut werden können:

aus (a,b) und V kann man die Abbildung (a,b).V bilden und

aus (a,C) und M kann man die Zuordnung (a,C).M bilden

(falls a nicht im Definitionsbereich von V vorkommt),

(wobei wir (a,b).V für die Vereinigung von {(a,b)} mit V schreiben). Man kann dann beweisen:

(W1) (a,b).V ist ein Vertretersystem für (a,C).M falls

V ist ein Vertretersystem für M und

b∉C und

b "kommt nicht vor in" V.

(W2) Die leere Abbildung ist ein Vertretersystem für die leere Zuordnung.

(Definition:

b "kommt nicht vor in" V genau dann, wenn

für alle (a,b')∈V: b'≠b.)

Das Problem, ein Vertretersystem für (a,C).M zu bestimmen, ist damit zurückgeführt auf

dasselbe Problem für die "kleinere" Eingabe M und

andere Probleme, nämlich

b∈C und

b "kommt nicht vor in" V zu entscheiden.

Setzen wir nun einmal voraus, daß sich die Unterprobleme "∈" und "kommt nicht vor in" algorithmisch lösen lassen! (In der Tat kann man das in weiteren Verfeinerungsstufen genauso machen wie das hier für das Hauptproblem gezeigt wurde). Dann läßt sich unter Verwendung des Wissens (W1), (W2) das Problem, ein Vertretersystem V für die konkrete Eingabe

M:= { (1,{1,2}), (2,{1,3}) }

zu finden, systematisch (algorithmisch, mechanisch) wie folgt lösen:

Wegen(W1):

(1) (1,b).V ist ein Vertretersystem für {1,{1,2}) . { (2,{1,3}) }

falls V ein Vertretersystem für { (2,{1,3}) } und

b ∈ {1,2} und

b kommt nicht vor in V.

(2) (2,b).V ist ein Vertretersystem für (2,{1,3}) . ∅

falls V ein Vertretersystem für ∅ und

b ∈ {1,3} und

b kommt nicht vor in ∅.

Wegen (W2):

(3)∅ ist ein Vertretersystem für ∅.

Einsetzen von (3) in (2) und Produzieren von

1 ∈ {1,3},

1 kommt nicht vor in ∅

durch "Aufruf der Unterprogramme" für "∈" und "kommt nicht vor in" liefert:

(4) (2,1).∅ ist ein Vertretersystem für (2,{1,3}) . ∅.

(4) kann in (1) eingesetzt werden, Produzieren von

1 ∈ {1,2}

führt aber in eine Sackgasse, denn

1 kommt nicht vor in (2,1).∅

ist falsch. Durch "Backtracking" muß man zurückgehen zur nächsten Produktionsmöglichkeit

$$2 \in \{1,2\},$$

die zusammen mit

$$2 \text{ kommt nicht vor in } (2,1).\emptyset$$

und (4) die rechte Seite von (1) erfüllt und damit zu

(5) $\{1,2\}.\{\{2,1\}\}$ ist ein Vertretersystem für

$$\{1,\{1,2\}\} \;.\; \{\,\{2,\{1,3\}\}\,\}$$

bzw. zu

(6) $\{\,\{1,2\},\ \{2,1\}\,\}$ ist ein Vertretersystem für

$$\{\,\{1,\{1,2\}\},\ \{2,\{1,3\}\}\,\}$$

führt.

Sobald also "algorithmisch brauchbares Wissen" für das betrachtete Problem in der Form von "Klausen" der obigen Art (W1), (W2) ("Horn-Klausen") vorhanden ist, kann man dieses Wissen in ganz mechanischer Art verwenden, um für konkrete Eingaben die Lösung des Problems zu berechnen.

Logisches Programmieren besteht nun im Anschreiben von Wissen über die in der Problemspezifikation beschriebene Funktion in der Form von Hornklausen. Ein logisches Programm ist demnach einfach eine Menge von Hornklausen. Hornklausen sind pradikatenlogische Formeln (siehe Kapitel 7 über automatisches Beweisen) der speziellen Gestalt:

Literal falls (Literal und Literal und ... und Literal),

wobei Literale negierte oder unnegierte atomare Formeln sind. Die bekannteste Programmiersprache mit diesem Programmbegriff ist PROLOG.

Die Möglichkeit, logische Programme automatisch zu exekutieren, d.h. durch entsprechende Compiler und Interpreter eine abstrakte Maschine zu realisieren, auf welcher Rechenvorgänge der in obigem Beispiel beschriebenen Art automatisch ablaufen können, ist ein wesentlicher Fortschritt der letzten Jahre. Der Interpreter für solche Programme ist im wesentlichen ein automatischer Beweiser (Resolutionsbeweiser, siehe Kapitel 7 über automatisches Beweisen).

Grob gesprochen muß ein Interpreter für eine logische Programmiersprache zusätzlich zu den Fähigkeiten der abstrakten Maschinen für

- Assemblersprachen (Fähigkeit: symbolische Adressen),
- ALGOL-ähnliche Sprachen (Fähigkeit: Blockkonzept),
- rekursive Sprachen (Fähigkeit: Stackmechanismus),

noch die Fähigkeit haben,

- Terme (und nicht nur Variable) als formale Parameter zu behandeln (was die Anpassung der Struktur aktueller Parameter an die Termstruktur der formalen Parameter erfordert: "Matchen", "Unifizieren"; siehe Kapitel 7: Automatisches Beweisen)
- und Suchräume selbständig zu durchwandern (Backtracking, Depth-first-Suche u.ä.; siehe Kapitel 2: Suchstrategien).

Zusammenfassend besteht der Vorteil des logischen Programmierens darin, daß Problemspezifikationen oft überhaupt nicht oder nur geringfügig transformiert beziehungsweise mit zusätzlichem Wissen angereichert werden müssen, um ein lauffähiges Programm zu erhalten. Logische Programme sind immer korrekt bezüglich der Problemspezifikation, solange das zusätzliche Wissen, das in das logische Programm eingebracht wird, aus dem Grundwissen folgt, das man über die Grundfunktionen hat, die in der Problemspezifikation vorkommen. Die Termination und die Komplexität der Berechnungen ist allerdings genauso ein Problem wie bei jedem andern Typ von Programmiersprache.

Literaturhinweise zum logischen Programmieren: Der Gedanke des logischen Programmierens ist ca. 1974 entstanden. Wie viele andere Ideen, "lag er in der Luft" und wurde von verschiedenen Autoren explizit formuliert. Für die grundlegenden Ideen siehe das Lehrbuch (Kowalski 1979). Von aktuellen Forschungsthemen gibt (Clark, Tärnlund 1982) einen Eindruck. PROLOG-Einführungen sind (Clocksin, Mellish 1981) und (Clark, McCabe 1984). Dieses letzte Buch ist besonders leicht lesbar. - Eine gewisse Zwischenstufe zwischen dem rekursiven, funktionalen Programmieren (wie z.B. LISP) und dem logische Programmieren stellt das Rewrite-Rule-Programmieren (bzw. Programmieren durch Spezifikation abstrakter

Datentypen) dar. Dieser Ansatz ist allerdings noch nicht so weit als praktische Programmiersprache ausgebaut wie PROLOG. Es wird aber an vielen Stellen daran gearbeitet, siehe z. B. (Lescanne 1983). In diesem Zusammenhang ist unter anderem das Erzwingen der Church-Rosser-Eigenschaft von Term-Reduktionssystemen durch "Critical-Pair/Completion" Algorithmen eine grundlegende Technik, die in speziellem Kontext in (Buchberger 1965) und in allgemeinerer Form in (Knuth-Bendix 1967) eingeführt wurde, siehe auch (Buchberger 84).

8.3 Automatische Programmsynthese

Logisches Programmieren verkürzt zwar den Weg zwischen Problemspezifikation und exekutierbarem Programm, das Erzeugen von algorithmisch brauchbarem Wissen (durch Beweisen) in Form von Hornklausen (verallgemeinerten Rewrite-Regeln) liegt aber vollkommen in der Hand des menschlichen Programmierers. Bei der automatischen Programmsynthese geht es um die Computer-Unterstützung des Übergangs von der Problemspezifikation zum Programm (für irgendeine abstrakte Maschine; natürlich wird auch die Automatisierung dieses Übergangs umso leichter sein, je höhere Programmiersprachen man als Zielsprache betrachtet).

Eine der Ideen für die Computer-Unterstützung dieses Übergangs ist die Extraktion von Algorithmen aus automatischen oder halbautomatischen Existenzbeweisen.

In spezieller Weise liegt diese Idee bereits dem logischen Programmieren zugrunde. Z. B. kann das Berechnen eines Vertretersystems V zur konkreten Eingabe $M_0 := \{ (1,\{1,2\}), (2,\{1,3\}) \}$ auch als automatischer Beweis der Aussage

"Es existiert ein V, sodaß V ist ein Vertretersystem für M_0"

unter Verwendung des Hornklausen-Wissens (W1), (W2) betrachtet werden, wobei im Laufe des Beweises die auftretenden Substitutionen von konkreten Werten für Variable "gesammelt" werden und schließlich in ihrer Kombination die "Ausgabe" (die Antwort, den

lösenden Ausdruck) liefern. Allgemein kann die Exekution von logischen Programmen auf einer "PROLOG-Maschine" als Beweis von Existenzaussagen der folgenden Art betrachtet werden:

"Es existiert ein y, sodaß $P(x_0,y)$".

wobei P(x,y) die Aussage ist, die den gewünschten Zusammenhang zwischen möglichen Eingaben und den zulässigen Ausgaben des Problems beschreibt. P ist also die Problemspezifikation. (Über die Praxis des formalen Spezifizierens siehe (Buchberger, Lichtenberger 81), S. 28-72. x_0 bezeichnet hier einen fixen Eingabewert (sprachlich: ein Term ohne freie Variable, z.B. eine Konstante).

Eine automatische Programmsynthese kann erfolgen, wenn es gelingt, allgemeine Existenzaussagen mit variabler Eingabe zu beweisen. Genauer sind die zwei Schritte einer automatischen Programmsynthese mit dem Gedanken "Extraktion von Algorithmen aus Existenzbeweisen":

1. Beweise (automatisch oder halbautomatisch):
   "Für alle x existiert ein y, sodaß P(x,y)",
   wobei P die Problemspezifikation ist.
2. Extrahiere aus dem Existenzbeweis den "lösenden Term", das ist ein Term t(x) für welchen gilt:
   "Für alle x P(x,t(x))".

Dieser Gedanke zur (halb)automatischen Programmsynthese wurde Anfang 1970 stark verfolgt. Zwischenzeitlich war er in den Hintergrund gerückt, weil in den üblichen universellen Beweisern das Instrument der Induktion, das für Beweise in algorithmischen Strukturen von grundlegender Bedeutung ist, nur über Umwege eingeführt werden kann. In der Zwischenzeit wurde die Technik des automatischen Beweisens stark verbessert. Der Gedanke der Extraktion von Algorithmen aus Existenzbeweisen lebt deshalb wieder auf. Hier sei ein auf diesem Gedanken aufbauendes neueres System von (Manna, Waldinger 1980) skizziert: Ein Beweis ist dabei eine "Sequenz" der folgenden Art

Behauptungen	Ziele	lösende Terme
$A_1(a,x)$		$s_1(a,x)$
$A_2(a,x)$		$s_2(a,x)$
	$G_1(a,x)$	$t_1(a,x)$
$A_3(a,x)$		$s_3(a,x)$
	$G_2(a,x)$	$t_2(a,x)$
	$G_3(a,x)$	$t_3(a,x)$

(a ... ein Konstantenvektor, x ... ein Variablenvektor, A_i, G_i ... Aussagen, s_i, t_i ... Terme). Eine solche Sequenz hat die Bedeutung:

Wenn für alle x $A_1(a,x)$ und
für alle x $A_2(a,x)$ und
für alle x $A_3(a,x)$
dann für ein x $G_1(a,x)$ oder
für ein x $G_2(a,x)$ oder
für ein x $G_3(a,x)$.

Wenn eine Instanz eines Ziels $G_i(a,x)$ wahr ist (oder eine Instanz einer Behauptung $A_i(a,x)$ falsch), dann ist die entsprechende Instanz von $t_i(a,x)$ (bzw. $s_i(a,x)$) ein Beispiel ("lösender Term") für das ursprüngliche Problem. Ein Beweis geschieht dadurch, daß zu einer bestehenden Sequenz durch Anwenden bestimmter Schlußregeln neue Zeilen hinzugefügt werden. Es sind vier Gruppen von Schlußregeln vorgesehen: Splitting Regeln, Transformationsregeln, verallgemeinerte Resolutionsregeln und strukturelle Induktion. Es kann hier alles nur an einem Beispiel gezeigt werden.

Beispiel (Manna, Waldinger 1980): Quotient und Rest bei ganzzahliger Division. Als Startzeilen einer Sequenz stellt man dieses Problem wie folgt dar:

Behauptungen	Ziele	lösende Terme	
		div(i,j)	rem(i,j)
B1: 0<i und 0≤j (Eingabebedingung)	Z2: i=y.j+z und 0<z≤j (Ausgabebedingung)	T: y;	z;

Als Grundwissen über =, < fügt man hinzu:

B3. u=u
B4. (u<v ==> nicht(v≤u))

Durch Anwenden der "Splitting Regel" erhält man

B5. 0<i
B6. 0≤j

Wissen über die Multiplikation verwendet man als "Transformations regeln", und zwar:

0.v → 0
(u+1).v → u.v+v.

Anwendung der ersten dieser Transformationsregeln auf Ziel Z2 führt zu:

Z7. i=0+z und 0<z≤j	T: 0;	z;

Ähnlich führt die Anwendung der Transformationsregel 0+v → v zu

Z8. i=z und 0<z≤j	T: 0;	z;

Resolution angewandt auf B3 und Z8 führt zu

Z9. 0<i≤j	T: 0;	i;

Nochmals Resolution angewandt auf B5 und Z9 führt zu

Z10. i≤j	T: 0;	i;

(In diesem Stadium kann man aus dem Beweis ablesen, daß im Fall $i<j$, 0 und i die Werte von Quotient und Rest sind). Auf das Ziel Z2 kann man jetzt die zweite Transformationsregel für die Multiplikation anwenden und erhält

$$Z11.\ i=y_1.j+j+z \text{ und } 0\leq z<j \quad T:\ y_1+1; \qquad z;$$

Die Transformationsregel $u=v+w \rightarrow u-v=w$, angewandt auf Z11, ergibt:

$$Z12.\ i-j=y_1.j+z \text{ und } 0\leq z<j \quad T:\ y_1+1; \qquad z;$$

Ziel Z12 hat nun wieder genau die Gestalt des ursprünglichen Zieles Z2. An dieser Stelle wird die entsprechende Induktionshypothese eingeführt:

B13. Wenn $(u_1,u_2)<(i,j)$, dann
(wenn $0<u_1$ und $0<u_2$,
dann $u_1=\mathrm{div}(u_1,u_2).u_2+\mathrm{rem}(u_1,u_2)$
und $0\leq\mathrm{rem}(u_1,u_2)<u_2$)

Durch Resolution zwischen Z12 und B13 erhält man

$$Z14.\ (i-j,j)<(i,j) \text{ und } 0\leq i-j \text{ und } 0<j \quad T:\ \mathrm{div}(i-j,j)+1; \qquad \mathrm{rem}(i-j,j);$$

Jetzt muß eine geeignete noethersche Ordnung zur Verfügung stehen, z.B. die durch die folgende Transformationsregel definierte Ordnung $<'$:

$(u_1,u_2)<'(v_1,v_2) \rightarrow \underline{\text{true}}$, wenn $u_1<v_1$.

Dadurch entsteht das neue Ziel

Z15.	i-j<1 und 0<i-j und 0<j	T: div(i-j,j)+1; rem(i-j,j);

Daraus folgt im wesentlichen durch Resolution mit den Behauptungen B6 (0<j) und B4 (u<v ===> nicht(v<u))

Z16.	nicht(i<j)	T: div(i-j,j)+1; rem(i-j,j);

(D.h. man weiß in diesem Stadium des Beweises, daß im Fall j<i Quotient und Rest von i und j durch div(i-j,j)+1 und rem(i-j,j) "rekursiv" bestimmt werden können). Aus Z10 und Z16 kann man schließlich durch einen (etwas modifizierten) Resolutionsschritt

	T: div(i,j):	rem(i,j):
Z19. true	if i<j then 0 else div(i-j,j)+1	if i<j then i else rem(i-j,j)

erhalten. true in der Zielspalte zeigt an, daß man aus den Spalten für die lösenden Terme die endgültigen (rekursiven) Programme für div und rem entnehmen kann.

Freilich ist der obige Beweis zunächst "händisch". Die Schwierigkeiten der Sequenz der Beweisschritte ist aber auf weite Strecken nicht von einer höheren Stufe als z.B. bei Resolutionsbeweisern für das universelle Beweisen in der Pradikatenlogik. An manchen Stellen scheint ein Einbringen einer "Idee" in den Beweis jedoch notwendig (und wahrscheinlich auch "wünschenswert") zu sein. Auf jeden Fall gibt dieses Beweissystem jedoch eine Vorstellung, inwieweit eine Automatisierung bzw. Computer-Unterstützung der Programmsynthese möglich erscheint. (Manna, Waldinger 1983) enthält ein nicht-triviales Beispiel einer ("händischen") Programmsynthese mit diesem Deduktionssystem, in welchem die (wenigen) Stellen der Deduktion, die für eine Automatisierung schwierig erscheinen, genau analysiert sind.

Literaturhinweise zur automatischen Programmsynthese: Beispiele anderer Grundgedanken zur automatischen Programmsynthese sind: Programmsynthese aus Beispielen von Ein-/Ausgabe-Paaren, siehe z.B. (Jouannaud, Kodratoff 1983); "Computer Aided Intuition Guided Programming", siehe (Bauer et al. 1983); "Falten und Entfalten", siehe z. B. (Darlington 1983); "Syntax-Directed, Semantics-Supported Program Synthesis", eine Kombination von automatischem Beweisen und heuristischen Problemlösestrategien, siehe (Bibel 1980); Umwandlung von "Zweiparameter-Algorithmen in ein Spektrum von Einparameter-Algorithmen", siehe (Goad 1982). Einen Einblick in aktuelle Forschungsthemen gibt (Biermann, Guiho 1983).

8.4 Automatische Programmtransformation

Praktisch sind Software-Systeme und Methoden für die automatische Programmsynthese eng mit der automatischen Programmtransformation verwoben. Dies umso mehr, da die Programmiersprachen, in welchen die synthetisierten Programme formuliert sind, meist sehr hoch sind. Trotzdem gibt es typische Techniken, die für die automatische Transformation von Programmen entwickelt wurden, deren Korrektheit relativ zu einer Problemspezifikation bereits als gegeben vorausgesetzt werden kann. Eine solche Technik ist die in (Darlington, Burstall 1976) und (Burstall, Darlington 1977) beschriebene. Sie besteht im wesentlichen in folgendem Dreischritt:

1. Entfalten
2. Anwenden von algebraischen Gesetzen
3. Falten.

"Entfalten" besteht dabei im wiederholten Ersetzen und Einsetzen unter Verwendung der Definitionen bzw. rekursiven Beziehungen zwischen den beteiligten Funktionen.

Den entstehenden Ausdruck kann man mit Hilfe der für die beteiligten Funktionen gültigen Gesetze (z.B. Assoziativität, Kommutativität) in eine andere Gestalt bringen.

In dieser neuen Form kann man versuchen, eine Instanz des definierenden Terms der interessierenden Funktion wiederzufinden und durch einen Aufruf der interessierenden Funktion (für ein "kleineres" Argument) zu ersetzen ("Falten"). Wir beschreiben nur die Faltungsoperation genauer. Alles andere wird wieder nur an einem Beispiel gezeigt.

Faltungsregel: Wenn E ← E' und F ← F' Gleichungen sind und in F' eine Instanz $E'_X[t]$ von F' vorkommt, dann darf man

F ← F"

als neue Gleichung einführen, wo F" aus F' dadurch entsteht, daß man $E'_X[t]$ durch $E_X[t]$ ersetzt.

Diese und die anderen harmlos aussehenden Regeln haben, im Sinne der Strategie "Entfalten, Umwandeln, Falten" angewandt, eine sehr große effizienzverbessernde Kraft. (Dies ist andererseits nicht verwunderlich, da die Regeln "Einsetzen" und "Ersetzen" im wesentlichen bereits einen universellen Conputer konstituieren).

Beispiel der Transformation eines rekursiven Prgramms in ein effizienteres (Burstall, Darlington 1977): Man betrachte folgende rekursive "Definition" der Fibonacci-Zahlen:

(1) f(0) ← 1
(2) f(1) ← 1
(3) f(x+2) ← f(x+1) + f(x)

Man transformiert in folgenden Schritten:

* (4) g(x) ← (f(x+1),f(x)) (mit "Definitionsregel")
(5) g(0) ← (f(1),f(0)) (mit "Substitutionsregel" aus (4))
(6) g(0) ← (1,1) (mit "Entfaltungsregel" aus (5) und (1),(2))
(7) g(x+1) ← (f(x+2),f(x+1)) (mit "Substitutionsregel" aus (4))
(8) g(x+1) ← (f(x+1)+f(x),f(x+1)) (mit "Entfaltungsregel" aus (7),(3))
(9) g(x+1) ← (u+v,u), wobei (u,v) = (f(x+1),f(x)) (mit "Wobei-Regel" aus (8))
(10) g(x+1) ← (u+v,u), wobei (u,v) = g(x)

(mit Faltungsregel aus (9),(4))

(11) f(x+2) ← u+v, wobei (u,v) = (f(x+1),f(x))
(mit "Wobei-Regel" aus (3))

(12) f(x+2) ← u+v, wobei (u,v) = g(x)
(mit Faltungsregel aus (11),(4)).

Zusammenfassend erhält man folgendes rekursive Programm für f:

f(0) ← 1
f(1) ← 1
f(x+2) ← u+v, wobei (u,v) = g(x)
g(0) ← (1,1)
g(x+1) ← (u+v,u), wobei (u,v) = g(x).

Die Berechnung von f(n) nach dem ursprünglichen Programm braucht exponentiell viele Schritte (Additionen), nach dem zweiten nur linear viele Schritte. Die einzige Stelle, wo eine "Idee" notwendig ist, ist die mit (*) gekennzeichnete. Dort muß man eine Idee für eine "günstige" Definition der neuen Funktion g haben. Alles andere verläuft mechanisch.

Literaturhinweise zur automatischen Programmtransformation: Das Projekt, in welchem der Gedanke der computer-unterstützten (aber durch die "Intuition geführten") Programmtransformationen am konsequentesten durchgeführt wird, ist CIP, siehe (Bauer et al. 1983). In engem Zusammenhang mit dem Themenkreis der automatischen Programmtransformationen stehen natürlich die seit langem studierten Techniken der Compiler-Optimierung, siehe z. B. (Zima 1983), Kapitel 7. Viel Material zu Programmtransformationen wurde auch im Zusammenhang mit dem Studium der "Programmschemata" zusammengetragen, siehe z. B. (Greibach 1975). Ebenso gehören hierher die an vielen Stellen studierten Möglichkeiten, spezielle Typen rekursiver Programme in effizientere iterative Programme zu transformieren, siehe z. B. den Algorithmus für ein Nimmspiel in (Buchberger, Lichtenberger 80), S. 224 ff.

8.5 Automatische Programmverifikation

Methoden zur computer-unterstützten Programmverifikation zerlegen das Problem der Verifikation in zwei Teile:

1. Generierung eines oder mehrerer Lemmata aus der Problemspezifikation und dem Programm, sodaß die Korrektheit des Programms relativ zur Problemspezifikation garantiert ist, falls die Lemmata bewiesen sind.
2. Computer-unterstützter Beweis der Lemmata.

Meist muß man in den Generator für die Lemmata zusätzlich zur Problemspezifikation und zum Programm noch einige weitere Information über das Programm einbringen. Die Generierung der Lemmata ("Verifikationsbedingungen") ist aber dann ein völlig automatisierbarer Vorgang.

Der Computer-unterstützte Beweis geschieht dann entweder mit einem universellen automatischen Beweiser oder mit speziellen Beweisern, die für den bestimmten Datentyp, über welchem das Programm arbeitet, sehr viel effizientere Beweise ausführen kann als ein universeller Beweiser.

Eine bekannte Methode der Programmverifikation, die als Grundlage für die computer-unterstützte Programmverifikation dienen kann, ist die Methode von Floyd-Naur-Hoare (Methode der induktiven Behauptungen) für ALGOL-ähnliche Programme. (Man schreibt "{E} S {A}" für die Korrektheitsaussage "Für alle x: wenn E(x) vor Ausführung des Programms S für die vorliegenden Werte der Programmvariablen x gilt, dann gilt A(x) nach Ausführung von S für die dann aktuellen Werte der Programmvariablen x". x ... ein Variablenvektor). Für jedes zusammengesetzte Sprachkonstrukt gibt es dann eine Regel, wie der Beweis der Korrektheitsaussage für dieses Sprachkonstrukt zurückgeführt werden kann auf den Beweis der Korrektheitsaussagen für die einzelnen Teilsprachkonstrukte. Wir geben nur ein Beispiel einer solchen Regel und zeigen alles andere wieder an einem Beispiel:

Regel für while-Schleifen:
Um

{E} P; while B do Q; R {A}

zu zeigen, genügt es, eine Aussage I ("Schleifeninvariante"), eine noethersche Relation $\prec$ auf einer Menge M und einen Term t ("Terminationsterm") zu suchen, für die man folgendes beweisen kann:

{E} P {I},
aus (I und B) folgt t ∈ M,
{I und B und t=T} Q {I und t$\prec$T} (T .. eine neue Variable),
{I und nicht B} R {A}.

Beispiel für eine Programmverifikation: Betrachte den Euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers (GGT) von zwei natürlichen Zahlen m, n zusammen mit seiner Spezifikation:

{m,n ∈ N}
(z,r):= (m,n)
(*)
while r≠0 do (z,r):= (r,Rest(z,r))
{z = GGT(m,n)}.

Hier steht "Rest(z,r)" für den Rest bei der ganzzahligen Division von z durch r. Als induktive Behauptung an der Stelle (*) wählen wir

GGT(z,r) = GGT(m,n), z≠0 oder r≠0,

als Menge M mit noetherscher Relation wählen wir N mit der Kleinerbeziehung und als Terminationsterm r. Gemäß der Regel für while-Schleifen muß man dann zeigen:

(1) { m,n∈N } (z,r):= (m,n) { GGT(z,r)=GGT(m,n), z≠0 oder r≠0 },
(2) aus GGT(z,r)=GGT(m,n), z≠0 oder r≠0, r≠0 folgt r∈N,

(3) { GGT(z,r)=GGT(m,n), z≠0 oder r≠0, r≠0, r=T }
(z,r):= (r,Rest(z,r))
{ GGT(z,r)=GGT(m,n), z≠0 oder r≠0, r¢T },
(4) { GGT(z,r)=GGT(m,n), z≠0 oder r≠0, r=0 } { z=GGT(m,n) }.

(1), (2) und (4) sind leicht zu zeigen. (3) wird durch Anwenden der "Zuweisungsregel" aufgelöst in

(3') Aus GGT(z,r)=GGT(m,n), z≠0 oder r≠0, r≠0, r=T
folgt GGT(r,Rest(z,r))=GGT(m,n), r≠0 oder Rest(z,r)≠0,
Rest(z,r)¢T.

(3') folgt aber unmittelbar aus dem folgenden Wissen über den GGT:

(W) aus r≠0 folgt GGT(z,r)=GGT(r,Rest(z,r)).

Wir beobachten an dem Beispiel einige für die automatische Programmverifikation grundsätzlich wichtige Dinge:

1. Die Erstellung der zu beweisenden Aussagen der Art (3') etc. ("Verifikationsbedingungen") aus der Problemspezifikation, dem Programm und den induktiven Behauptungen ist ein vollkommen mechanischer Vorgang, der leicht automatisiert werden kann.

2. Der Beweis der Verifikationsbedingungen zerfällt in einen wesentlichen Teil, der im wesentlichen das algorithmisch brauchbare Wissen benutzt, das auch zum Entwurf des Algorithmus (z.B. in Form eines logischen Programms) zentral ist (siehe (W) im Beispiel), und in triviale Teile, wie z.B. der Beweis von (1), der im wesentlichen mit leichten Substitutionen etc. auskommt.

3. Der "triviale" Teil ist leicht automatisierbar und es ist auch hilfreich, diese Routineteile des Beweises zu automatisieren. Der "wesentliche Teil" kann auch leicht automatisiert werden, sobald das nötige algorithmische Wissen zur Verfügung gestellt wird. Der Beweis dieses Wissens kann allerdings beliebig

schwierig sein, in ihm steckt der "kreative" Teil des Algorithmenentwurfs.

4. Programmverifikation ist nur sinnvoll, wenn sie in den Entwurfsprozeß eingebettet ist. Das algorithmische Wissen, das den Entwurfsprozeß leitet, ist eben auch dasjenige, welches über die zentralen Stellen des Korrektheitsbeweises führt. Verifikation von "fertigen" Programmen bezüglich vorgegebener Spezifikationen ist "unnatürlich", weil man beim Verifizieren das Programm noch einmal entwickeln muß.

Literaturhinweise zur automatischen Programmverifikation: Das bisher am weitesten entwickelte Verifikationssystem ist der Stanford-PASCAL-Verifier, siehe (Luckham et al., 1979), (Polak 1981). Dieses System baut auf der Methode der induktiven Behauptungen auf und kann den Entwicklungsprozeß von logisch durchaus anspruchsvollen Algorithmen zu einem sehr großen Teil automatisch unterstützen. Andere fortgeschrittene Programmverifikationssysteme sind z.B.: das System von (Boyer, Moore 1979), das als Programmiersprache und als Beweissprache eine LISP-ähnliche Sprache verwendet; das AFFIRM-System, siehe (Gerhart et al., 1980), das auf dem Konzept der abstrakten Datentypen aufbaut (siehe die Bibliographie über abstrakte Datentypen (Kutzler, Lichtenberger 1983)); das LCF-System, das auf dem Lambda-Kalkül basiert, siehe (Gordon, Milner, Wadsworth 1979). Für eine detaillierte Einführung in die Praxis der "händischen" Programmverifikation in Verbindung mit einer praktischen Einführung in die Technik des Beweisens siehe (Buchberger, Lichtenberger 1980). Leider gibt es noch sehr wenig zusammenfassende Literatur zu speziellen automatischen Beweisern, die im Rahmen der Programmverifikation sicher eine mindestens ebenso wichtige Rolle wie universelle Beweiser spielen. Ein Beispiel eines speziellen Beweiser ist (Nelson, Oppen 1980).

8.6 Ein integrierter Software-Arbeitsplatz

Die verschiedenen Ansätze zum automatischen Programmieren sind derzeit in verschiedenen experimentellen Pilotsystemen implementiert. In Zusammenhang mit den vielfältigen Bemühungen, das Management des Software-Entwicklungsprozesses durch den Computer
zu unterstützen, die Computer-Ein-/Ausgabe durch graphische und natürlichsprachliche Schnittstellen zu erleichtern, Expertenwissen in Expertensystemen und Methodenbanken zur Verfügung zu stellen, die Potenz zur Erarbeitung von algorithmisch brauchbarem Wissen durch universelle und spezielle automatische Beweiser zu vergrößern, gewisse Transformationsprozesse auf mathematischen Objekten zu automatisieren ("Computer-Algebra"), sollten in naher Zukunft modular aufgebaute Systeme entstehen, die den Problemlöseprozeß von der vagen Problemformulierung über die exakte Problemspezifikation im Rahmen einer formalen, deskriptiven Sprache bis hin zum korrekten, effizienten und auf einer abstrakten Maschine lauffähigen Programm unterstützen. Dabei sollte der Mensch weder bezüglich seines Problemlösestils eingeschränkt werden, sondern unter verschiedenen "Philosophien" angepaßt an das Problem und den Stand der Problemlösung wählen können, noch erscheint es sinnvoll, ihm alle kreativen Aktionen während des Problemlöseprozesses abzunehmen. Flexible Systeme, die die verschiedenen Ansätze und das damit gewonnene Know-How integrieren, erscheinen für die nächste Zukunft realisierbar und erstrebenswert. In (Buchberger 82) wird ein Studienschwerpunkt im Rahmen der Informatik bzw. Mathematik beschrieben, der versucht, die Studenten in die formalen und praktischen Aspekte solcher integrierter Systeme einzuführen.

Dank: Diese Arbeit wurde durch den Österreichischen Fonds zur Förderung der wissenschaftlichen Forschung unterstützt (Projekt Nr. 4567).

Literatur

- Barr A., Feigenbaum E. A.: The Handbook of Artificial Intelligence. Vol. II, Heuristech Press, Stanford; 1982.

- Bauer F. L. und CIP Language Group: The Munich Project CIP, Vol.I: The Wide Spectrum Language 85. Bericht, Technische Universität München, Institut für Informatik; Dezember 1983.

- Bibel W.: Syntax-Directed, Semantics-Supported Program Synthesis. Artificial Intelligence 14, 243-261; 1980.

- Biermann A. W., Guiho G. (Hsg.): Computer Program Synthesis Methodologies. Proc. of the NATO Advanced Study Institute, Bonas, September 1981, Reidel Publ. Comp., Dodrecht, Boston, London; 1983.

- Boyer R. S., Moore J. S.: A Computational Logic. Academic Press, New York - London; 1979.

- Buchberger B.: Ein algorithmisches Kriterium für die Lösbarkeit algebraischer Gleichungssysteme. Aequationes mathematicae 4(3), 374-383; 1970. (Publikation der Dissertation, Univ. Innsbruck, 1965).

- Buchberger B.: Studienschwerpunkt CAMP (Computer-Aided Mathematical Problem Solving) an der Universität Linz. Bericht Nr. CAMP 82-4.1, Institut für Mathematik, Universität Linz; 1982.

- Buchberger B.: Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory. In: Recent Trends in Multidimensional Systems Theory (N. K. Bose Hsg.), D. Reidel Publ. Comp., Dodrecht, Boston, London; erscheint 1984.

- Buchberger B., Collins G. E., Loos R.: Computer-Algebra (Symbolic and Algebraic Computation). Springer-Verlag, Wien - New York; 1982 (2. Auflage 1983).

- Buchberger B., Lichtenberger F.: Mathematik für Informatiker I (Die Methode der Mathematik). Springer-Verlag, Berlin - Heidelberg - New York; 1980 (2. Auflage 1981).

- Burstall R. M., Darlington J.: A Transformation System for Developing Recursive Programs. J. ACM 24(1), 1977.

- Clark K. L., McCabe F. G.: Micro-Prolog: Programming in Logic. Prentice-Hall, Engelwood Cliffs, N.J.; 1984.

- Clark K. L., Tarnlund S.-A. (Hsg.): Logic Programming. Academic Press, London; 1982.

- Clocksin W. F., Mellish C. S.: Programming in Prolog. Springer, Berlin - Heidelberg - New York; 1981.

- Darlington J.: The Synthesis of Implementations for Abstract Data Types, A Program Transformation Tactic. In (Biermann, Guiho 1983), 309-334.

- Darlington J., Burstall R. M.: A System which Automatically Improves Programs. Acta Informatica 6, 41-60; 1976.

- Gerhart S. L. and AFFIRM group: An Overview of AFFIRM: A Specification and Verification System. Proc. of the IFIP Congress 1980 (Lavington S. H. Hsg.), 343-347; 1980.

- Goad C. A.: Automatic Construction of Special Purpose Programs. Proc. 6th Conference on Automated Deduction, Springer Lecture Notes in Computer Science 138 (Loveland D. W. Hsg.), Berlin - Heidelberg - New York - Tokyo; 1982.

- Gordon M. J., Milner A. J., Wadsworth C. P.: Edinburgh LCF. Lecture Notes in Computer Science 78, Springer, Berlin; 1979.

- Greibach S. A.: Theory of Program Structures: Schemes, Semantics, Verification. Lecture Notes in Computer Science 36, Springer, Berlin - Heidelberg - New York; 1975.

- Hesse W.: Methoden und Werkzeuge für Software-Entwicklung: Ein Marsch durch die Technologie-Landschaft. Informatik-Spektrum 4(4), 229-245; 1981.

- Jouannaud J.-P., Kodratoff Y.: Program Synthesis from Examples of Behavior. In: (Biermann, Guiho 1983), 213 - 250.

- Knuth D. E., Bendix P. B.: Simple Word Problems in Universal Algebras. Proc. of the Conf. on Computational Problems in Abstract Algebra, Oxford 1967 (Leech J., Hsg.), 263-298, Pergamon Press, Oxford; 1970.

- Kowalski R.: Logic for Problem Solving. North-Holland, New York - Oxford; 1979.

- Kutzler B., Lichtenberger F.: Bibliography on Abstract Data Types. Informatik Fachberichte 68, Springer, Berlin - Heidelberg - New York - Tokyo; 1983.

- Lescanne P.: Computer Experiments With the REVE Term Rewriting System Generator. Proc. of the Principles of Programming Languages Conference; 1983.

- Luckham D. C. and PASCAL Verifier Group: Stanford PASCAL Verifier User Manual. Stanford, Computer Science Department, Report No. STAN-CS-79-731; 1979.

- Manna Z., Waldinger R.: A Deductive Approach to Program Synthesis. ACM TOPLAS 2/1, 92-121; 1980.

- Manna Z., Waldinger R.: Deductive Synthesis of the Unification Algorithm. In: (Biermann, Guiho 1983), 251-308.

- Nelson C. G., Oppen D. C.: Fast Decision Procedures Based on Congruence Closure. J. ACM 27(2), 356-364; 1980.

- Polak W.: Program Verification at Stanford: Past, Present, Future. Report, Stanford University, Computer Systems Laboratory; 1981.

- Zima H.: Compilerbau II (Synthese und Optimierung). Reihe Informatik 37, Bibliographisches Institut, Mannheim, Wien, Zürich; 1983.