8.1 Problemstellung

Programmieren (algorithmisches Problemlösen) ist die Tätigkeit, die von einer Problembeschreibung (Problemspezifikation) zu einem auf einer Maschine ausführbaren Algorithmus (Programm, Lösungsverfahren) führt. Zu Beginn des Computerzeitalters vor nunmehr ca. 40 Jahren mußten alle Schritte beim Programmieren von der Problemspezifikation bis zum Programm in der Internsprache des verwendeten Computers vom Menschen ausgeführt werden. Die Entwicklung der Informatik seither kann wesentlich durch den Fortschritt charakterisiert werden, der bei der Unterstützung des Programmierens durch den Computer selbst erzielt wurde. Immer mehr Teilschritte des Programmiervorganges werden als Routinevorgänge erkannt und dementsprechend als vom Computer durchführbare Aufgaben dem Menschen abgenommen, sodaß sich der menschliche Problemlöser immer mehr auf wesentliche, kreative, höhere, zentralere, universellere Aspekte des Problemlösens konzentrieren und beschränken kann.

Das Gebiet des automatischen Programmierens (automatic programming) ist jener Teil der Informatik, der die Entwicklung immer ausgefeilterer Methoden für diese Computer-Unterstützung des Programmiervorganges zum Ziele hat. Sehr frühe Stadien der Computer-Unterstützung des Programmiervorganges (z.B. Assembler und Compiler für ALGOL-ähnliche Sprachen) zählt man allerdings - als heute selbstverständliche Bestandteile einer Programmierumgebung - 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 betrachtet, der sich mit der Computer-Realisierung von Verhaltensweisen befaßt, die als "bisher dem Menschen vorbehalten" erscheinen, 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 dem 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, Menu-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 demonstieren. 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 $a \varepsilon A$: $V(a) \varepsilon M(a)$.
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:
$V(a) \varepsilon M(a)$ ... 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-Maschin sollte als Routinearbeit wohl möglichst selbst einem Compute überlassen werden können. Um dieses Ziel zu erreichen, sollt algorithmisch brauchbares Wissen möglichst in der Form, wie es al mathematisches Wissen abgeleitet (bewiesen) wird, auch bereits al Programm verwendet werden können. Die dazu notwendige abstrakt Maschine muß dazu einige Fähigkeiten haben, die weit über di Fähgigkeiten 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
$b \in C$ 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 $(a,b') \in V$: $b' \neq b$.)

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 ε C und

b "kommt nicht vor in" V zu entscheiden.

Setzen wir nun einmal voraus, daß sich die Unterprobleme "ε" 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 ε {1,2} und

b kommt nicht vor in V.

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

falls V ein Vertretersystem für Ø und

b ε {1,3} und

b kommt nicht vor in Ø.

Wegen (W2):

(3)Ø ist ein Vertretersystem für Ø.

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

1 ε {1,3},

1 kommt nicht vor in Ø

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

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

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

1 ε {1,2}

führt aber in eine Sackgasse, denn

1 kommt nicht vor in (2,1).Ø

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 prädikatenlogische 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 Problespezifikationen 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 zwischem dem rekursiven, funktionalen Programmieren (wie z.B. LISP) und dem logische Programmieren stellt das Rewrite-Rule-Programmieren (bzw. Programmieren durch Spezifikation abstrakter