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: $0 \leq i$ und $0 < j$ (Eingabebedingung)	Z2: $i = y.j+z$ und $0 \leq z < j$ (Ausgabebedingung)	T: y;	z;

Als Grundwissen über $=$, $\leq$ fügt man hinzu:

B3. $u = u$

B4. $(u \leq v \Rightarrow \text{nicht}(v < u))$

Durch Anwenden der "Splitting Regel" erhält man

B5. $0 \leq i$

B6. $0 < j$

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

$0.v \rightarrow 0$

$(u+1).v \rightarrow u.v+v.$

Anwendung der ersten dieser Transformationsregeln auf Ziel Z2 führt zu:

Z7. $i = 0+z$ und $0 \leq z < j$	T: 0;	z;

Ähnlich führt die Anwendung der Transformationsregel $0+v \rightarrow v$ zu

Z8. $i = z$ und $0 \leq z < j$	T: 0;	z;

Resolution angewandt auf B3 und Z8 führt zu

Z9. $0 \leq i < j$	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

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

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

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

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 $0\leq u_1$ und $0<u_2$,
dann $u_1=\text{div}(u_1,u_2).u_2+\text{rem}(u_1,u_2)$
und $0\leq\text{rem}(u_1,u_2)<u_2$)

Durch Resolution zwischen Z12 und B13 erhält man

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

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<i 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 Prädikatenlogik. 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 E' 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 Computer 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) $f(x+2) \leftarrow u+v$, wobei $(u,v) = (f(x+1),f(x))$

(mit "Wobei-Regel" aus (3))

(12) $f(x+2) \leftarrow u+v$, wobei $(u,v) = g(x)$

(mit Faltungsregel aus (11),(4)).

Zusammenfassend erhält man folgendes rekursive Programm für f:

$f(0) \leftarrow 1$
$f(1) \leftarrow 1$
$f(x+2) \leftarrow u+v$, wobei $(u,v) = g(x)$
$g(0) \leftarrow (1,1)$
$g(x+1) \leftarrow (u+v,u)$, 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: