

Informatik-Fachberichte 59, 141 -202, (Springer, Berlin)
Frühjahrsschule Künstliche Intelligenz
Teisendorf, März 1982

Computer-unterstützter Algorithmenentwurf *)

B. Buchberger
(Universität Linz)

Zusammenfassung:

Einige wesentliche Ideen, die in den letzten 15 Jahren für den Computer-unterstützten Algorithmenentwurf erarbeitet und z.T. in experimentellen Systemen verwirklicht wurden, werden an Hand von einfachen, aber ausführlichen Beispielen erklärt.

Inhalt:

Vom Problem zum Algorithmus
Die Rolle von mathematischem Wissen im Problemlöse-Prozeß
Computer-unterstützte Programmverifikation
Computer-unterstützte Programmtransformationen
Computer-unterstützte Strategien zur Programmsynthese
Computer-unterstützte Extraktion von Algorithmen aus Existenzbeweisen
Spezifikationen abstrakter Datentypen als Programme
Ausblick

Literatur

*) Unterstützt vom Österr. Fonds zur Förderung der wissenschaftlichen Forschung
(Projekt Nr. 4567),

Vom Problem zum Algorithmus

Die Beschreibung (Spezifikation) eines Problems ist

eine Formel (ein sprachlicher Ausdruck, eine Aussage), die gewünschte Eigenschaften einer (oder mehrerer) neuer Operationen (Funktionen oder Prädikate) relativ zu als gegeben (elementar) betrachteten Operationen angibt.

Eine Operation ist "gegeben",

wenn für sie ein Berechner (Berechnungsmechanismus, Rechner, Computer, Automat) vorhanden ist, der

zu jeder (sinnvollen) Eingabe (Angabe, Input) für die Operationen bei Aufruf der Operation eine zugehörige Ausgabe (Resultat, Output) liefert.

Die Lösung eines Problems relativ zu einem Berechner besteht in der Angabe

einer Formel (eines sprachlichen Ausdrucks, "Programmes", einer Anweisung), die den Aufbau neuer Operationen aus gegebenen Operationen beschreibt und bei deren Aufruf der Berechner Operationen durchführt, die die gewünschten Eigenschaften haben (die berechneten Operationen sind korrekt in Bezug auf die Problemspezifikation).

Beispiel einer Problembeschreibung:

Neue Operation: sort(ieren).

Beschreibung der gewünschten Eigenschaft:

sort(a) ist sortiert,

sort(a) ist eine permutierte Version von a.

b ist sortiert: $\langle \implies \rangle$ Für alle $1 \leq i < \text{Länge von } b$: $b_i < b_{i+1}$.

b ist eine permutierte Version von a : $\langle \implies \rangle$

Länge von a = Länge von b;

Es gibt eine Permutation p der Länge (Länge von a),

sodaß für alle $1 \leq i < \text{Länge von } a$: $b_i = a_p(i)$.

p ist ein Permutation der Länge n: $\langle \implies \rangle$

p ist eine bijektive Funktion von $\{1, \dots, n\}$ auf $\{1, \dots, n\}$.

(Typen der Variablen: a, b, p ... endliche Folgen natürlicher Zahlen,

i, n natürliche Zahlen.

sort(a) soll ebenfalls eine endliche Folge natürlicher Zahlen sein).

Elementare Operationen:

"Länge von", Indizieren, +, $\{1, \dots, n\}$ (Funktionen),
 \leftarrow , "ist bijektive Funktion von nach" (Prädikate).

Beispiel einer Lösungsbeschreibung:

sort(a):

Eingabe: a

Ausgabe: b.

$(b, n) := (a, \text{Länge von } a)$

for j := 1 to n-1 do

for k := 1 to n-j do .

if not $(b_k < b_{k+1})$ then $(b_k, b_{k+1}) := (b_{k+1}, b_k)$.

(sort ist die neue Operation; "Länge von", +, -, \leftarrow , Indizieren sind die benutzten elementaren Operationen).

Eine explizite Problembeschreibung ist eine Problembeschreibung der Gestalt

"Für alle $x : P(x, f(x))$ ",

wo f die neue Operation bezeichnet und

$P(x, y)$ eine Formel (der Prädikatenlogik) erster Stufe ist

(mit freien Variablen x, y),

in welcher f nirgends vorkommt.

(Wir schreiben hier " $P(x, f(x))$ " für das Ergebnis der Substitution von $f(x)$ für y an allen Stellen, wo y frei vorkommt).

Bei expliziten Problembeschreibungen gibt $P(x, y)$ an, in welcher Weise ein korrekter Output y für das Problem (der durch die neue Operation f berechnet werden soll) mit dem jeweiligen gegebenen Input x zusammenhängt.

Der allgemeine Fall einer Problembeschreibung heißt demgegenüber "implizit".

(Insbesondere nennen wir nicht-explizite Problemspezifikationen implizit).

Eine explizite Problemstellung kann man sprachlich immer in die Form "gegeben, gesucht" bringen:

Gegeben: x .

Gesucht: y ,

sodaß $P(x, y)$.

Oder im Normalfall (wo $P(x,y)$ die Gestalt $(E(x) \implies Q(x,y))$ hat):

Gegeben: x ,
 sodaß: $E(x)$ ("Eingabebedingung").
 Gesucht: y ,
 sodaß: $P(x,y)$ ("Ausgabebedingung").

Beispiel einer expliziten Problembeschreibung:

"b ist sortiert und

b ist eine permutierte Version von a"

ist eine Aussage der Gestalt $P(a,b)$, die die gewünschte Eigenschaft von "sort" explizit beschreibt: Zu jeder Eingabe a für sort beschreibt $P(a,b)$ wie eine korrekte Ausgabe $b = \text{sort}(a)$ beschaffen sein muß.

Beispiel einer impliziten Problembeschreibung:

Neue Operationen: lesen, speichern.

Beschreibung der gewünschten Eigenschaften:

"Für alle s,i,j,c :

lesen(speichern(s,i,c), i) = c ,

lesen(speichern(s,i,c), j) = lesen(s,i), wenn $i \neq j$."

(Für "speichern(s,i,c)" lies: "Der Speicher, der aus s entsteht, wenn man an der Stelle i den Inhalt c hinspeichert". Für "lesen(s,i)" lies: "Der Inhalt, der im Speicher s an der Stelle i steht").

Eine (implizite oder explizite) Problembeschreibung bestimmt die betreffenden neuen Operationen im allgemeinen nicht eindeutig. (Für manche Operationen, z.B. die Nachfolgerfunktion auf den natürlichen Zahlen, ist eine eindeutige Charakterisierung durch Spezifikationen in erster Stufe, d.h. ohne Quantoren über die Operationssymbole, sogar grundsätzlich unmöglich). Nicht jedes implizite Problem läßt sich in ein äquivalentes explizites verwandeln. Es ist algorithmisch, nämlich syntaktisch, entscheidbar, ob eine vorgegebene Problemspezifikation explizit ist. Typische Beispiele von implizit definierten Problemen sind durch rekursive Gleichungen "definierte" Operationen oder z.B. algebraisch spezifizierte abstrakte "Datentypen".

Die Schwierigkeit des Auffindens einer Problemlösung hängt ab
 vom Problem und
 vom zur Verfügung stehenden Berechnungsmechanismus.

Schlußregelsysteme für (verschiedene deskriptive Sprachen z.B. für) Prädikatenlogik erster Stufe sind sehr allgemeine (nicht-deterministische) Berechnungssysteme. Jede Formel einer (solchen) Sprache kann deshalb

einerseits als Beschreibung einer (erwünschten) Eigenschaft von Operationen und andererseits als Manipulationsvorschrift ("Programm") mit den Operationen aufgefaßt werden.

Alles, was mit den Schlußregeln aus der Problemspezifikation abgeleitet werden kann, ist "korrekt" bezüglich der Problemspezifikation. (Es ist jedoch möglich, daß die so gewonnene Problemlösung unvollständig ist in dem Sinne, daß nicht für alle Eingaben mit den Schlußregeln aus der Problemspezifikation die "Berechnung" einer zugehörigen Ausgabe durchgeführt werden kann).

Beispiel der Auffassung einer Problemspezifikation als Programm:

Aus der Problemspezifikation (dem "Programm") für sort und dem nicht angeführten Grundwissen (den "Programmen") für die elementaren Funktionen können wir z.B. schließen ("Rechnen"):

- (1) Länge von $\text{sort}((2,1)) = \text{Länge von } (2,1) = 2$.
- (2) $\text{sort}((2,1))_1 = (2,1)_{p(1)}$ und $\text{sort}((2,1))_2 = (2,1)_{p(2)}$ oder
 $\text{sort}((2,1))_1 = (2,1)_{q(1)}$ und $\text{sort}((2,1))_2 = (2,1)_{q(2)}$,
 wobei p und q die beiden Permutationen (1,2) bzw. (2,1) sind (als Dupel geschrieben).
- (3) $\text{sort}((2,1))_1 < \text{sort}((2,1))_2$.
- (4) $\text{sort}((2,1))_1 = (2,1)_{q(1)} = 1$,
 $\text{sort}((2,1))_2 = (2,1)_{q(2)} = 2$
 (die andere Alternative in (2) scheidet wegen (3) aus).
- (5) $\text{sort}((2,1)) = (1,2)$ (aus (4)).

Wir haben also für die Operation "sort" zum "Input" (2,1) den "Output" (1,2) bei sort "berechnet".

Beispiel der Auffassung einer Problemspezifikation als Programm:

Aus der Problemspezifikation (dem "Programm") für "lesen" und "speichern" und Grundwissen über natürliche Zahlen kann man z.B. schließen ("rechnen"):

$$\begin{aligned} & \text{lesen}(\text{speichern}(\text{speichern}(\text{leer}, 1, 5), 2, 3), 1) = \\ & = \text{lesen}(\text{speichern}(\text{leer}, 1, 5), 1) = 5. \end{aligned}$$

Wir haben also für die Operation "lesen" zu den "Inputs" $\text{speichern}(\text{speichern}(\text{leer}, 1, 5), 2, 3)$ und 1 den "Output" 5 "berechnet".

Zusammenfassend:

Eine Problemspezifikation ist immer eine korrekte (allenfalls nicht vollständige) Problemlösung, wenn als "Rechner" ein "Beweiser" zugelassen wird.

Für die Praxis hat diese Einsicht in dieser Form wenig Bedeutung, weil die möglichen Beweise, die von einer Problemspezifikation als "Programm" ausgehen, zunächst völlig ungezielt und richtungslos verlaufen können und nur mehr oder weniger "zufällig" für einen gegebenen "Input" zu einem erstrebenswerten Resultat führen. Man beachte jedoch den Unterschied zwischen den beiden Beispielen: Im ersten Beispiel ist eine große Fülle verschiedenartigster Schlußschritte notwendig (von denen wir nur sehr grobe Zwischenschritte angegeben haben), um zum Ziel zu kommen. Wenn überhaupt, dann ist eine Zielgerichtetheit im Beweis "automatisch" durch das Auflösen des Existenzquantors "es existiert eine Permutation p " nahegelegt, was einem "systematischen Durchprobieren und Austesten" aller möglichen Permutationen entspricht.

Beim zweiten Beispiel kommt man im wesentlichen mit den sehr einfachen Schlußregeln für die Gleichheit (Ersetzen, Einsetzen, Identität, Symmetrie und Transitivität) aus und es ist auch ziemlich klar, in welcher Weise diese Schlußregeln "zielgerichtet" anzuwenden sind, um zum "Resultat" zu kommen.

Auf der anderen Seite steht die in der Praxis heute fast ausschließlich vorhandene Situation: Die vorhandenen virtuellen oder realen Rechner verlangen die Angabe von Lösungsverfahren (relativ zu den elementaren Operationen der Rechner), durch welche ihr Verhalten vollständig determiniert wird, wodurch ein zielstrebiges Lösungsverhalten ermöglicht wird, allerdings auch die Frage nach der Korrektheit des Verfahrens ein zentrales Problem wird. Also:

Rechner	Lösungsverfahren	Korrektheit	Effizienz
allgemeiner "Schlüssezieher"	gleich der Problem- spezifikation	"automatisch" garantiert	keine Zielstrebigkeit, ineffizient
universeller deterministischer Rechner	nichttrivial, anders als Problem- spezifikation	Korrektheit ist ein "Problem"	vollkommen zielgerichtet, effizient

Sowohl Korrektheit automatisch zu garantieren, als auch effizient zu sein, erscheint als das Ideal für das Lösen von Problemen. Diesem Ideal kann man sich von zwei Seiten her nähern:

- †. Die Rechner, wie sie sich in ihrer virtuellen Gestalt (Hardware + aufgesetzter Software) dem Problemlöser zeigen, für problemnähere Lösungsverfahren brauchbar zu machen ("intelligenter" zu machen ohne Effizienzverlust).
- †. Den Vorgang des Transformierens von Problemspezifikationen in rechnernahe Problemlösungen besser abzusichern und technisch zu erleichtern.

Beide Wege haben eine "lange" Geschichte:

- †. Höhere Programmiersprachen mit ihren Compilern; Zurverfügungstellung des Sprachmittels der Rekursion; Rewrite-Sprachen; Horn-Clausen Sprachen; Fifth-Generation Computer.
- †. Programmgeneratoren; Ingenieurmäßige Methoden der Software-Entwicklung; computer-unterstützte Programmverifikation; Kalküle der korrekten Programmtransformation; Strategien für automatische Programmsynthese.

In dieser Vorlesung sollen die wichtigsten Denkansätze in beiden Wegen, soweit sie noch nicht standardmäßig in industriellen Software-Systemen realisiert sind, an Beispielen erläutert werden.

Die Rolle von mathematischem Wissen im Problemlöseprozeß

Zwischen mathematischem Wissen über die in einem Problem involvierten Operationen und der Effizienz von Problemlösungen besteht ein Identitätszusammenhang:

Mehr mathematisches Wissen ---> besserer Lösungsalgorithmus

Beispiel: Wissen über das Sortierproblem.

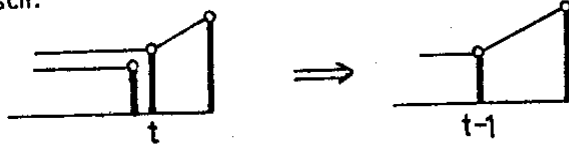
Für die im Sortierproblem involvierten Operationen gilt z.B. Folgendes:

b ist sortiert ab der Stelle t und
 der Wert von b an der Stelle t ist größer gleich allen davor liegenden Werten

==>

b ist sortiert ab der Stelle t-1.

Graphisch:



(b ist sortiert ab der Stelle t : $\Leftarrow \Rightarrow$)

für alle i mit $t < i < \text{Länge von } b : b_i < b_{i+1}$.

Der Wert von b an der Stelle t ist größer gleich allendavor liegenden

Werte : $\Leftarrow \Rightarrow$

für alle i mit $1 < i < t : b_i < b_t$.

Dieses Wissen kann als korrekte Ergänzung der Problemspezifikation für einen Beweiser als Rechner Beweise der Art

$\text{sort}((3,2,1,4)) = (1,2,3,4)$

verkürzen. Andererseits kann es die Idee und die Grundlage für den Korrektheitsbeweis für folgenden Sortieralgorithmus liefern:

$\text{sort}(a)$:

Eingabe: a.

Ausgabe: b.

$(b, n) := (a, \text{Länge von } a)$

for j := 1 to n-1 do

 b := Maximum(b, j),

wo Maximum(b, j) eine Operation ist, die das Maximum von b_1, \dots, b_{n-j+1} an die Stelle $n-j+1$ bringt (siehe früheres Beispiel eines Programmes; "Bubble-Sort Algorithmus").

Beispiel Wissen über ein Nimspiel:

Zu lösen sei folgendes Problem (Rechenberg [44]):

Zwei Personen spielen miteinander, indem sie von einem Haufen Streichhölzer abwechselnd nach folgender Regel Hölzer wegnehmen:

Der erste nimmt eine beliebige Anzahl i von Hölzern weg, jedoch nicht alle.

Der zweite darf vom verbleibenden Rest wieder eine beliebige Anzahl i' von Hölzern wegnehmen, die jedoch kleiner oder gleich der "Schranke" $2i$ sein muß.

Der erste darf nun $i'' < 2i'$ Hölzer wegnehmen. Usw.

Wer bei seinem Zug alle verbleibenden Hölzer wegnehmen kann, hat gewonnen.

Hier bedeutet eine "1" bzw. eine "0" an der Stelle (n,i) , daß $G(n,i)$ gilt bzw. nicht gilt. Es mag sein, daß bei Beobachtung dieser Tabelle zwei Dinge auffallen:

- (N1) Für $n =$ eine Fibonacci-Zahl ist die gesamte Spalte im wesentlichen 0.
 (N2) Zwischen der k -ten Fibonacci-Zahl F_k und der $(k+1)$ -ten F_{k+1} wiederholt sich "im wesentlichen" der Werteverlauf zwischen 1 und F_{k-1} .

(Diese Beobachtung muß noch etwas verfeinert werden, siehe die Unstimmigkeit bei $n = 12, i = 4$, die sich tatsächlich erst bei einem so hohen Wert von n zeigt).

(N1) und (N2) ist zunächst nur eine Vermutung (Rechenberg [45]), die sich aus der Betrachtung eines endlichen Stückes der Tabelle ergibt. Die Vermutung läßt sich in der Tat beweisen (siehe Buchberger [12] bzw. Buchberger/Lichtenberger [14]). Der Beweis braucht keine besonders tiefgründigen mathematischen Resultate, nur einige Grundtatsachen über Fibonacci-Zahlen, jedoch einige technische Mühe, eine Reihe von Fallunterscheidungen und verschachtelten Induktionen und benötigt ca. 20 Seiten Papier.

Das mathematische Wissen (N1), (N2) kann nun zur Konstruktion eines besseren Algorithmus verwendet werden:

Algorithmus:

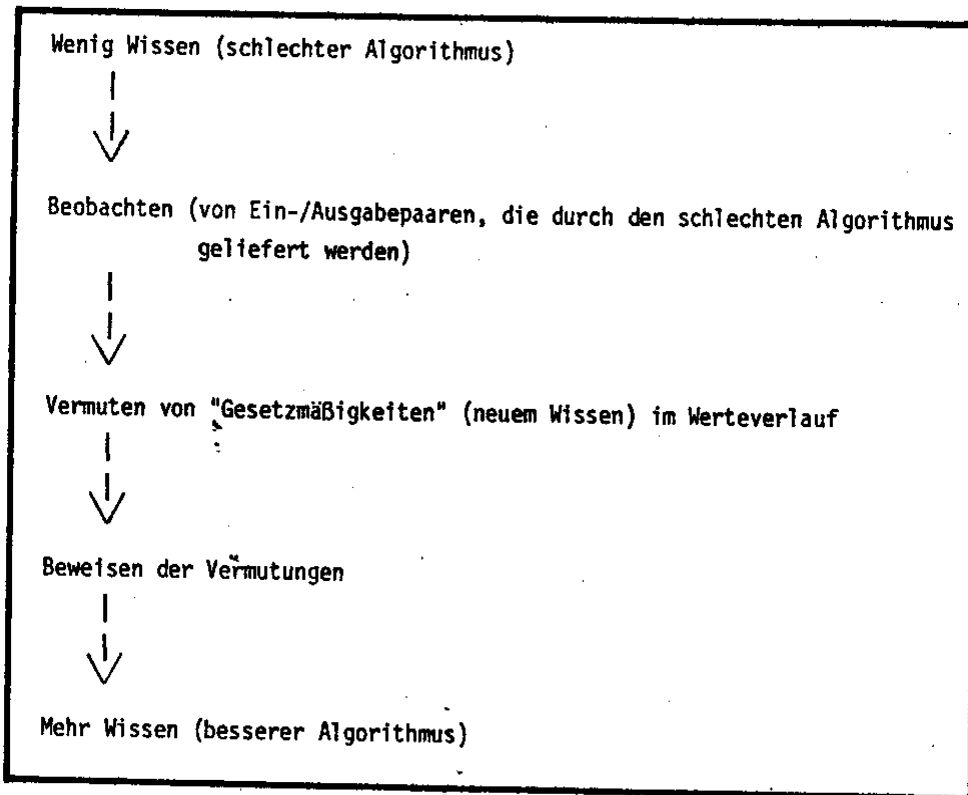
```

if n < m
then i := n
else
  if  $\frac{n}{3} < m$  then  $m := \lceil \frac{n}{3} \rceil - 1$ 
  while  $K_n < n \wedge m < n - K_n$  do
    n := n -  $K_n$ 
    if  $\frac{n}{3} < m$  then  $m := \lceil \frac{n}{3} \rceil - 1$ 
  if n =  $K_n$  then i := 1 else i := n -  $K_n$ 

```

($K_n :=$ die größte Fibonacci-Zahl, die kleiner oder gleich n ist).

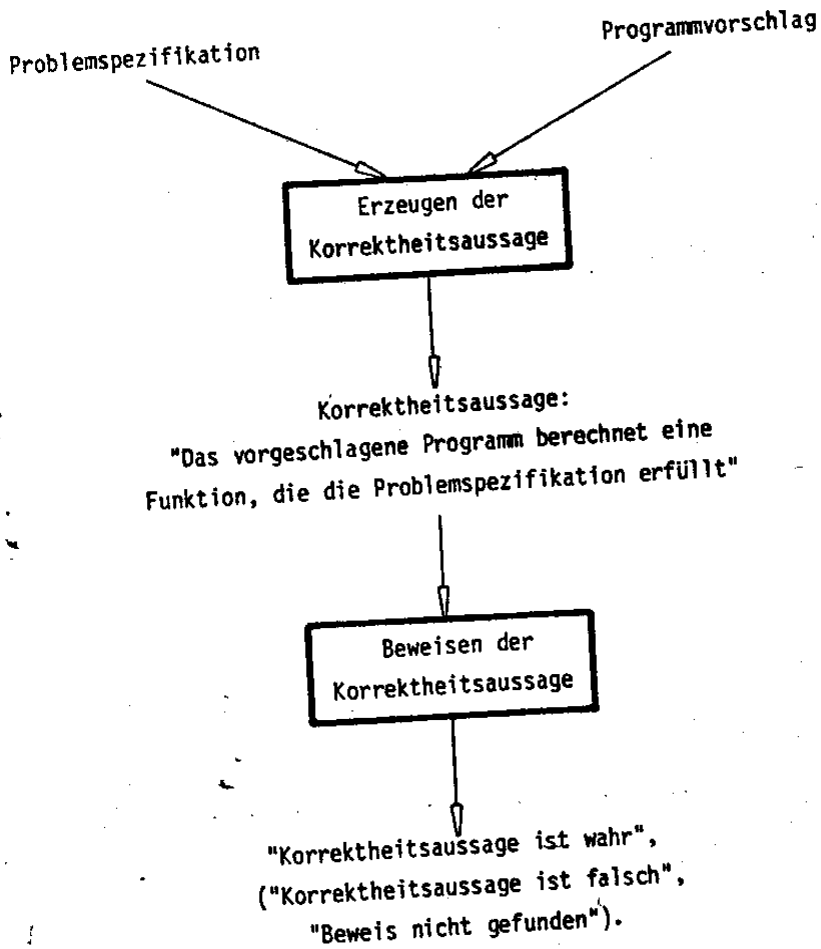
Aus diesem Beispiel kann man auch eine wichtige Grundstrategie für die Gewinnung von neuem Wissen (---> besseren Algorithmen) ablesen:



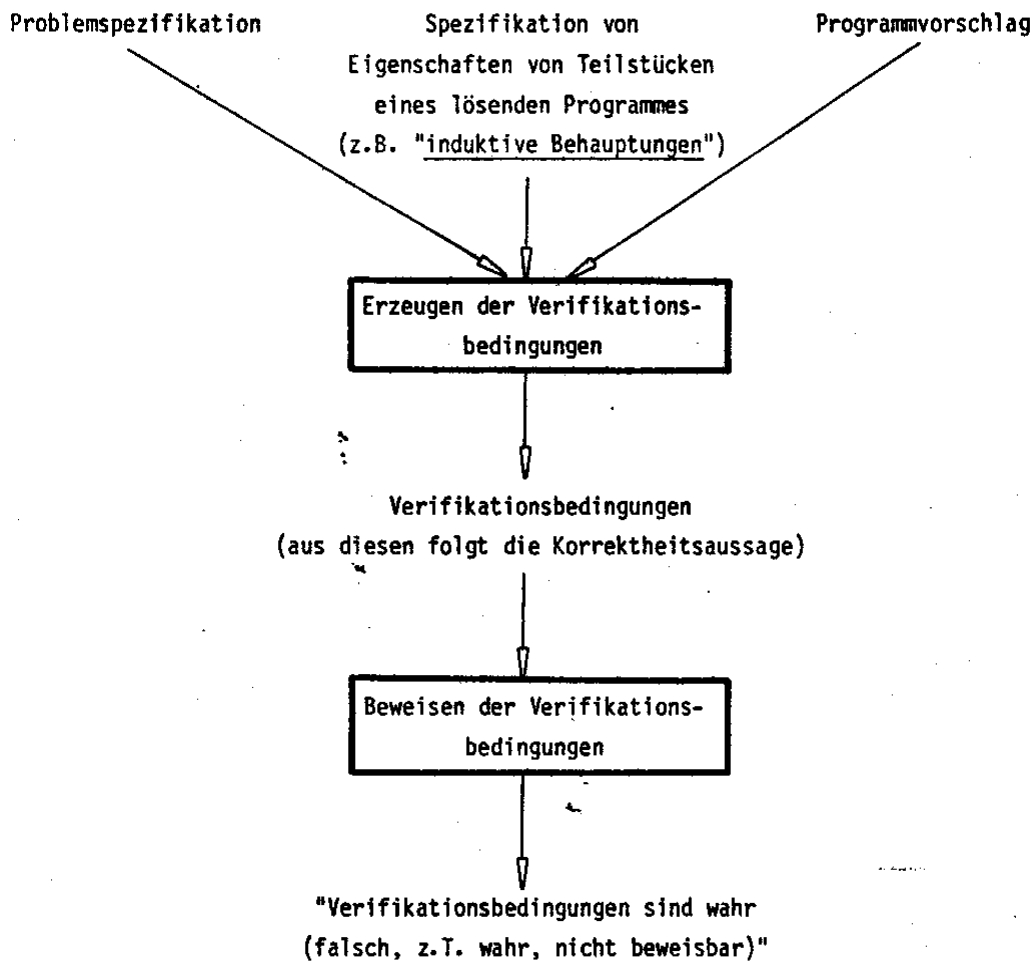
Übung: Beispiele von: Mehr mathematisches Wissen ---> besserer Algorithmus.
 (Hinweis: Fast jedes mathematische Wissen kann in einem sehr weiten Sinne so verstanden werden, umgekehrt basiert jede Algorithmenverbesserung auf zusätzlichem Wissen).

Computer-unterstützte Programmverifikation

Computer-unterstützte Programmverifikation ist ein erstes Paradigma für computer-unterstützten Algorithmenentwurf. In einer ersten Grobbeschreibung besteht dieses Paradigma in folgenden Schritten:



In dieser Form des Paradigmas wäre der Block "Erzeugen der Korrektheitsaussage" trivial, der Block "Beweisen der Korrektheitsaussage" müßte ein "Wunderding" sein. Es gibt nun Regelsysteme (am meisten benutzt die sogenannte "Methode der induktiven Behauptungen"), mit denen der Beweis der Korrektheitsaussage durch den Beweis einer Reihe von einfacheren Aussagen ("Verifikationsbedingungen") zerlegt werden kann, aus denen die Korrektheitsaussage folgt. Die einfacheren Aussagen sind hierbei Beschreibungen von Eigenschaften von Teilstücken eines lösenden Programmes:



Das Erzeugen der Verifikationsbedingungen ist immer noch ein sehr einfacher Prozeß, das Beweisen der einzelnen Verifikationsbedingungen kann schwierig sein, jedoch ist der Beweis der Korrektheit wenigstens in entkoppelte Teile zerlegt.

Der zentrale Punkt der Methode liegt im Erfinden der zusätzlichen Eigenschaften des lösenden Programmes. Hier sind zwei Grundsituationen denkbar:

A. "Verifikation von fertigen Programmen":

Zu einer gegebenen Problemspezifikation liegt ein Programmvorschlag "fertig" vor. Es werden vermutete Eigenschaften dieses Programmes spezifiziert. Die daraus entstehenden Verifikationsbedingungen (das sind Aussagen der Art "die Teilstücke des Programmes haben die vermuteten Eigenschaften") werden bewiesen.

B. "Entwickeln und Verifizieren":

Es liegt zunächst nur eine Problemspezifikation vor.

Man hat Ideen für zusätzliches Wissen über die in der Problemstellung involvierten Operationen.

Man erfindet eine Programmstruktur und Teilstücke von Programmen und gewünschte Eigenschaften dieser Programme,

sodaß die daraus entstehenden Verifikationsbedingungen ("die Programmstücke haben die gewünschten Eigenschaften") im wesentlichen mit dem zusätzlichen Wissen identisch sind oder leicht daraus folgen.

Die Betrachtungs- (und Vorgangs)weise B. ("Algorithmenentwicklung und Verifikation gehen Hand in Hand") ist die vernünftige (siehe Henke, Luckham [25]), aufgrund derer das Verifizieren von Programmen ein "natürlicher" und praktisch durchführbarer Vorgang ist. Sie entspricht der Einsicht, daß Programmieren "Umsetzen von Wissen in Verfahren" ist. Die Betrachtungsweise A. hat die Programmverifikation als praktische Technik in Mißkredit gebracht, weil man bei A. den Eindruck bekommt, daß man das Programm "noch einmal" erfinden muß, um geeignete induktive Behauptungen zu finden.

Beispiel für ein Regelsystem, mit welchem Verifikationsbedingungen mit Programmteilen und Spezifikationen von gewünschten Eigenschaften von Programmteilen verbunden werden:

Methode von Floyd-Naur-Hoare (Methode der induktiven Behauptungen) für ALGOL-ähnliche Programme (Floyd [19], Hoare [27]):

(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 ". $x \dots$ ein Vektoren).

Regel für Folgen von Wertzuweisungen:

Um $\{E\} S \{A\}$ zu beweisen, wo S eine Folge von Wertzuweisungen ist, genügt es, die Aussage $E \implies A'$ zu beweisen,

wo A' aus A dadurch entsteht, daß man die in A vorkommenden Variablen "so ersetzt, wie es durch die Hintereinanderausführung der einzelnen Wertzuweisungen in S bewerkstelligt wird".

(Z.B.: Um

{

(b,n) := (a,L(a)); j := 1

{b ist sortiert ab der Stelle n-j+2,

der Wert von b an der Stelle n-j+2 ist größer gleich allen davor-

liegenden Werten (falls $2 < j$), $j < n$, $n=L(b)$, b ist eine permutierte

Version von a}

} A

zu zeigen, genügt es,

\implies a ist sortiert ab der Stelle $L(a)+1$,

der Wert von a an der Stelle $L(a)+1$ ist größer gleich allen davor liegenden Werten falls $2 < l$, $1 < L(a)$, $L(a)=L(a)$, a ist eine permutierte Version von a

} A'

zu beweisen (Ersetzung: $b \rightarrow a$, $n \rightarrow L(a)$, $j \rightarrow 1$).

A' gilt "trivialerweise".

Regel für for-Schleifen:

Um $\{E\} P; \text{for } x := t_1 \text{ to } t_2 \text{ do } Q \text{ endfor}; R \{A\}$ zu beweisen ($P, Q, R \dots$ Programme, $t_1, t_2 \dots$ Terme; die Belegung von x und alle freien Variablen in t_2 mögen in Q nicht geändert werden), genügt es, für eine Aussage I ("Schleifeninvariante") folgendes zu beweisen:

(F1) $\{E\} P \{I'\}$

(wo I' aus I durch Ersetzen von x durch t_1 entsteht),

(F2) $\{I, x < t_2\} Q \{I''\}$

(wo I'' aus I durch Ersetzen von x durch $(x+1)$ entsteht),

(F3) $\{I''\} R \{A\}$

(wo I'' aus I durch Ersetzen von x durch (t_2+1) entsteht).

(Z.B.: Um

{

$(b, n) := (n, L(a))$

for j := 1 to n-1 do Q

{b ist sortiert, b ist eine permutierte Version von a}

zu zeigen, genügt es, wenn wir z.B. zeigen:

(F1') $\{ \} (b, n) := (n, L(a)) \{I(a, b, j, 1)\}$,

(F2') $\{I(a, b, n, j), j < n-1\} Q \{I(a, b, n, j+1)\}$,

(F3') $\{I(a, b, n, n)\} \{b \text{ ist sortiert, } b \text{ ist eine permutierte Version von } a\}$.

(Hier ist $I(a, b, n, j) : \iff b$ ist sortiert ab $n-j+2$,

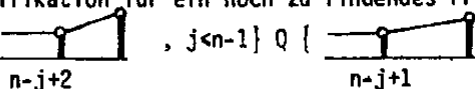
der Wert von b an der Stelle $n-j+2$ ist größer gleich allen davorliegenden Werten (falls $j > 2$),

$j < n$, $n=L(b)$, b ist eine permutierte Version von a.)

(F1') gilt (vgl. Beispiel bei der Regel für Wertzuweisungen).

(F3') gilt ebenfalls:  \implies 

(F2') ist eine Problemspezifikation für ein noch zu findendes Programm Q (das im wesentlichen also $\{ \text{---} \bullet \text{---} \}$, $j < n-1\} Q \{ \text{---} \bullet \text{---} \}$ erfüllen



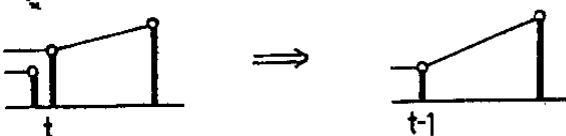
soll).

(Für entsprechende Regeln für die anderen Basiskonstrukte von ALGOL-ähnlichen Sprachen und eine didaktische Einführung in die Programm-Verifikation siehe z.B. Manna [32], Buchberger/Lichtenberger [14]).

Beispiel einer Programmverifikation (im Stile "Entwickeln und Verifizieren):

Problemspezifikation: $\{ \} P \{ b \text{ ist sortiert, } b \text{ ist permutierte Version von } a \}$
 (d.h. grob: $\{ \} P \{ \text{Diagramm} \}$)).

Wissen:



Erster Programmentwurf für P mit Vorschlag für induktive Behauptung:

$(b,n) := (a,L(a))$

①

for j := 1 to n-1 do Q,

wo die induktive Behauptung naheliegend $I(a,b,n,j)$ sein wird (man beachte, daß die induktive Behauptung meistens außer dem wichtigen Wissen noch eine Reihe "kleinerer" Teilbehauptungen enthält, deren Notwendigkeit sich meistens erst bei Durchführung des Beweises "aus technischen Gründen" ergibt).

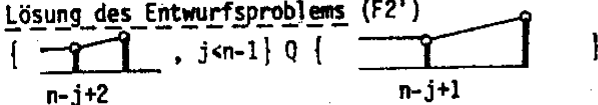
Die Problemspezifikation und die induktive Behauptung führt mit diesem Programmentwurf gemäß den Regeln zu den Teilproblemen: (F1'), (F2'), (F3') (wobei (F1') und (F3') Beweisprobleme sind, während (F2') ein Entwurfsproblem ist).

(F1') führt gemäß den Regeln zur Verifikationsbedingung: $\implies I(a,a,L(a),1)$.

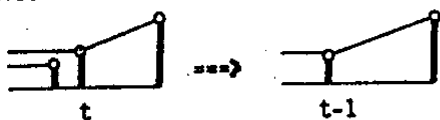
Beweis der Verifikationsbedingung: Sehr leicht (siehe voriges Beispiel).

(F3') gilt (siehe voriges Beispiel).

Zur Lösung des Entwurfsproblems (F2')



gibt das Wissen



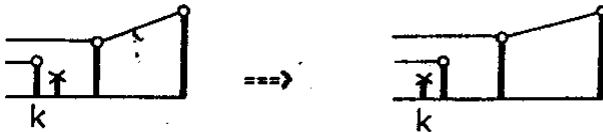
eine Idee, nämlich: Es genügt der Entwurf eines Programmes Q , welches folgende Spezifikation erfüllt

$$\left\{ \begin{array}{c} \text{---} \circ \text{---} \\ | \quad | \\ \text{---} \end{array} \right\}, j < n-1 \quad Q \quad \left\{ \begin{array}{c} \text{---} \circ \text{---} \\ | \quad | \\ \text{---} \end{array} \right\}$$

$n-j+2$
 $n-j+2$

(d.h. Q muß im wesentlichen das Maximum von b_1, \dots, b_{n-j+1} an die Stelle $n-j+1$ bringen).

Brauchbares Wissen, um dieses Problem zu lösen:



Dieses Wissen kann z.B. zu folgendem Programmvorschlag für Q führen:

```

for k := 1 to n-j do
  if not (bk < bk+1) then (bk, bk+1) := (bk+1, bk).

```

Die Behandlung dieses Vorschlags mit den Regeln liefert insgesamt vier Verifikationsbedingungen, die sämtlich mit dem skizzierten Wissen leicht zu beweisen sind (siehe Buchberger/Lichtenberger [14]).

An welchen Stellen in einem derartigen Verifikationsprozeß ist nun eine Computer-Unterstützung sinnvoll und möglich?

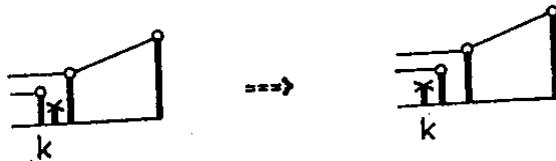
1. Wünschenswert wäre natürlich eine Unterstützung im Beweis von "algorithmisch brauchbarem" Wissen über die involvierten Begriffe (Operationen). Solches Wissen ist grundlegend. Beweise in diesem Bereich können beliebig schwierig sein. Computer-Unterstützung in diesem Bereich ist zwar wünschenswert, um die Beweismühen zu verringern, hier wird aber eine Interaktion mit dem menschlichen Problemlöser notwendig und wünschenswert sein, um Richtungen in der Problemlösung festzulegen.
2. Der Prozeß des Generierens von Teilverifikationsproblemen bzw. Verifikationsbedingungen bei vorgegebenen induktiven Behauptungen und vorhandenem Programm-vorschlag ist ein vollständig algorithmisierbarer Vorgang und es ist auch in hohem Maße wünschenswert, diesen Routinevorgang dem Menschen abzunehmen.

3. Der Beweis der Verifikationsbedingungen unter Verwendung des vorhandenen Grundwissens ist zwar als ein Beweisvorgang eine für die Computer-Unterstützung nicht triviale Aufgabenstellung. Die Beweise in diesem Bereich werden aber in der Regel viel weniger komplex sein. Relativ einfache Beweiser ("Simplifier") werden hier im Normalfall ausreichen. Es ist auch außerordentlich wünschenswert, diese Routinebeweise dem menschlichen Problemlöser abzunehmen.

(4. Der Versuch, induktive Behauptungen bei gegebenem Programm automatisch zu finden, hat nur bei der Vorgangsweise A. einen Sinn. Siehe dazu z.B. Wegbreit [49]).

Beispiel eines Beweises von algorithmisch brauchbarem Wissen:

Wir beweisen:



d.h.

$k+1 < t$,

b ist eine permutierte Version von a ,

b ist sortiert ab der Stelle t ,

der Wert von b an der Stelle t ist größer gleich allen davorliegenden Werten,

der Wert von b an der Stelle k ist größer gleich allen davorliegenden Werten,

$b_k > b_{k+1}$

====>

b' ist sortiert ab der Stelle t ,

der Wert von b' an der Stelle t ist größer gleich allen davorliegenden Werten,

der Wert von b' an der Stelle $k+1$ ist größer gleich allen davorliegenden Werten,

b' ist eine permutierte Version von a ,

wobei b' = "die aus b durch Vertauschen von b_k und b_{k+1} entstehende Folge"

(d.h. $b' = \text{speichern}(\text{speichern}(b, k, b_{k+1}), k+1, b_k)$, wobei b_k eine Abkürzung für $\text{lesen}(b, k)$ ist; vgl. Beispiel einer impliziten Problembeschreibung).

Wir zeigen nur einen Teil der Behauptung, nämlich

b' ist eine permutierte Version von a , d.h.

Länge von b' = Länge von b und

es existiert eine Permutation p' der Länge (Länge von a), sodaß für alle i mit

$1 < i < \text{Länge von } a$:

$$b'_i = a_{p'(i)}.$$

Wegen der Annahme, daß b eine permutierte Version von a ist, gibt es ein p , sodaß

$$b_i = a_p(i) \text{ (für alle } 1 < i < \text{Länge von } a).$$

Wir definieren: $p'(k) := p(k+1)$,
 $p'(k+1) := p(k)$,
 $p'(i) := p(i)$ für alle i mit $1 < i < \text{Länge von } a$, $i \neq k$, $k+1$.

Sei nun $i \neq k$, $k+1$ fix, aber beliebig. Dann gilt:

$$b'_i = \text{speichern}(\text{speichern}(b, k, b_{k+1}), k+1, b_k)_i = *$$

$$= \text{speichern}(b, k, b_{k+1})_i = * \quad b_i = a_{p(i)} = a_{p'(i)}.$$

(An den Stellen $*$) wurden die Axiome für "speichern, lesen" verwendet);

Sei nun $i = k$. Dann gilt:

$$b'_i = b'_k = \text{speichern}(\text{speichern}(b, k, b_{k+1}), k+1, b_k)_k = *$$

$$= \text{speichern}(b, k, b_{k+1})_k = * \quad b_{k+1} = a_{p(k+1)} = a_{p'(k)} = a_{p'(i)}.$$

Ähnlich behandelt man den Fall $i = k+1$. Außerdem müßte man noch zeigen, daß p' eine Permutation ist.

Beispiel eines Beweises von Verifikationsbedingungen (unter Verwendung von bereits bewiesenem, algorithmisch brauchbarem Wissen):

Algorithmisch brauchbares Wissen:

sort(b,t),
 größer(b,t)
 größer(b,t-1)
 ==>
 sort(b,t-1)

(Hier haben wir abgekürzt: "b ist ab der Stelle t sortiert" durch "sort(b,t)",
 "der Wert von b an der Stelle t ist größer gleich
 allen davorliegenden Werten" durch "größer(b,t)").

Eine der Verifikationsbedingungen, die bei der Verifikation des Teilprogramms Q im Sortierprogramm entsteht, ist:

sort(b,n-j+2),
 größer(b,n-j+2) (falls $2 < j$),
 $j < n$, $n = L(b)$, permut(a,b)
 $j < n-1$,
 größer(b,k),
 $k < n-j+1$,
 $k > n-j$
 ==>
 sort(b,n-j+1),
 größer(b,n-j+1) (falls $2 < j+1$)
 $j+1 < n$, $n = L(b)$, permut(a,b)

(permut(a,b) steht als Abkürzung für "b ist eine permutierte Version von a").
 größer(b,n-j+1) folgt aus größer(b,k) und $n-j < k < n-j+1$, also $k = n-j+1$. sort(b,n-j+1)
 folgt dann aus sort(b,n-j+2), größer(b,n-j+2) und größer(b,n-j+1) durch "Anwenden"
 des obigen Wissens.

Man beachte den phänomenologischen Unterschied zwischen den beiden Beweistypen: Der
 letzte Beweis war mehr oder weniger ein reiner "Rewrite"-Beweis (Ersetzen und Einset-
 zen in "Horn-Klausen"), während man im ersten Beweis an wesentlichen Stellen mit dem
 Existenzquantor (es existiert eine Permutation p) hantieren mußte bzw. Konzepte aus
 der Mengenlehre (Quantifizieren über Funktionen hier durch Verwenden von
 "speichern"/"lesen" umgangen) verwenden mußte.

Die Notwendigkeit, Beweistechniken anzuwenden, die die Zielstrebigkeit von
 "Rewrite"-Beweisen zerstören, kann man umgehen, wenn man bei der Problemdefinition
 sich auf eine implizite "Definition" der zugrunde liegenden Operationen, d.h. Angabe
 von als gültig vorausgesetztem, algorithmisch brauchbarem Wissen im Stile von
 Rewrite-Formeln konzentriert. Dies ist eine der wesentlichen Grundentscheidungen im
 Stanford PASCAL Verifikationssystem (Luckham et al. [31], Suzuki [48]).

Beispiel einer Definition des Sortierbegriffes im Rewrite-Stil:

sort(b,1), permut(a,b) \implies b ist sortierte version von a.

sort(b,L(b)+1).

sort(b,t), größer(b,t), größer(b,t-1) \implies sort(b,t-1).

sort(b,t), $i < t$, $k < t$, größer(b,t) \implies sort(vertausche(b,i,k),t).

größer(b,1).

größer(b,t), $b_t < b_{t+1}$ \implies größer(b,t+1).

größer(b,t), $b_t > b_{t+1}$ \implies größer(vertausche(b,t,t+1),t+1).

speichere(speichere(b,i,b_k),k,b_i) = vertausche(b,i,k).

permut(vertausche(b,i,k),b).

permut(a,a).

permut(a,b) \implies permut(b,a).

permut(a,b), permut(b,c) \implies permut(a,c).

Diese Eigenschaften des Sortierbegriffs und von in diesem Begriff involvierten
 Begriffen sind ausreichend, um die sich im früheren Beispiel ergebenden Verifika-
 tionsbedingungen zu beweisen.

Zusammenfassend ist das Zusammenspiel zwischen Mensch und Maschine in einem Computer-unterstützten Programmverifikationssystem ein iterativer Vorgang, bei dem ein "Schritt" ("pass") wie folgt aussieht:

Text bestehend aus teilweise entwickeltem Programm (Ergebnis der vorhergehenden Stufe, am Anfang leeres Programm) und Spezifikation von Unterproblemen (am Anfang nur das vom Menschen spezifizierte Gesamtproblem).



Zusammenstellen von relevantem Wissen über die involvierten Grundbegriffe durch den Menschen (nachschaugen, selber erfinden und beweisen, allenfalls Unterstützung beim Beweis durch universelle oder spezielle Theorem Prover).



Vorschlag für induktive Behauptungen und Programmteile für die noch nicht bearbeiteten Unterprobleme durch den Menschen



Erzeugen der Verifikationsbedingungen bzw. von Problemspezifikationen für weitere Unterprobleme durch die Maschine



Versuch des Beweises der Verifikationsbedingung unter Verwendung des relevanten Wissens (in Rewrite-Form) durch einen automatischen "Simplifier" (das relevante Wissen, versehen mit strategischen Hinweisen für seine Verwendung wird durch den Menschen beigestellt, siehe oben)



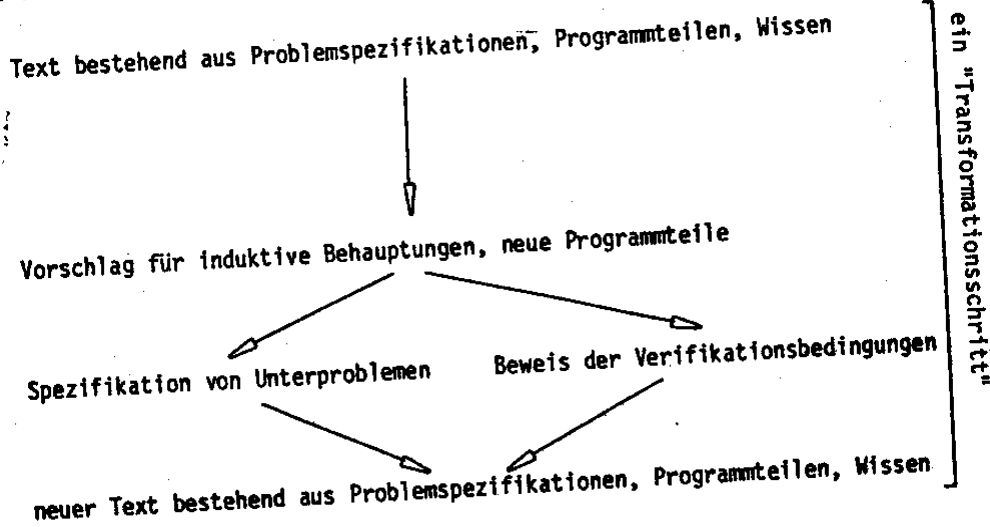
Analyse des Beweisergebnisses durch den Menschen, Hinweise durch die Maschine

Stand der Entwicklung: Computer-unterstützte Algorithmenverifikation hat einen Stand erreicht, wo logisch anspruchsvolle Algorithmen (im Stile der Algorithmen in Aho, Hopcroft, Ullman [1]) durchaus sinnvoll interaktiv mit maschinellen Hilfen an nicht-trivialen Stellen entwickelt werden können. Das am weitesten fortgeschrittene System ist der Stanford-PASCAL-Verifier (Luckham et al. [31], Polak [40]).

Andere computer-unterstützte Verifikationssysteme, bei denen die Methode der induktiven Behauptungen z.T. nur eine untergeordnete Rolle spielt, sind z.B. Boyer, Moore [9], Gerhart et al. [20] und Gordon et al. [23].

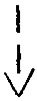
Computer-unterstützte Programmtransformationen

Ausgehend von der Übersicht über das Zusammenspiel zwischen Mensch und Maschine in der Computer-unterstützten Programmverifikation ergibt sich in natürlicher Weise die Frage, ob der Schritt



in welchem also zuerst Programmteile und induktive Behauptungen vorgeschlagen und dann ihre gegenseitige Verträglichkeit bewiesen wird, nicht einfacher und natürlicher ersetzt werden kann durch einen Schritt der Art

Text bestehend aus Problemspezifikationen, Programmteilen, Wissen



Vorschlag für eine korrekte Transformation des "Textes" durch den Menschen



Durchführung bzw. Kontrolle der korrekten Durchführung der Transformation durch die Maschine



neuer Text bestehend aus Problemspezifikationen, Programmteilen, Wissen

Dies setzt voraus, daß man durch Beobachten des Programmentwicklungsprozesses genügend viele und allgemeine Transformationsregeln herausfiltern kann, die einerseits algorithmisch durchführbar und andererseits "korrekt" sind in dem Sinne, daß ein "Problem/Programm"-Text immer in einen äquivalenten "Problem/Programm"-Text überführt wird,

und daß ein genügend allgemeine Sprache vorhanden ist, in welcher Konglomerate aus (deskriptiven) Problemspezifikationen und (algorithmischen) Programmbeschreibungen formuliert werden können. Transformationsregeln sind dann in zwei verschiedenen Phasen der Programmentwicklung notwendig:

Transformationsregeln, die angewandt auf die Spezifikation von (Teil)Problemen einen bezüglich der Problemspezifikation korrekten (Teil)Programmtext liefern und

Transformationsregeln, die angewandt auf ein (relativ zu einer bestimmten Problemspezifikation korrektes) Programm ein dazu äquivalentes (und damit ebenfalls korrektes) Programm liefern, das effizienter ist.

Der zunächst schwierig erscheinende Übergang ist der von einer "statischen" Problemspezifikation (Beschreibung eines Wunsches) zum "dynamischen" Algorithmus (Methode zur Erfüllung des Wunsches). An dieser kritischen Stelle "Problem --> Algorithmus" sind Sprachkonstrukte nützlich, die

gleichzeitig als Problembeschreibung und
 gleichzeitig als Beschreibung eines Programmes zur Lösung des Problems
 aufgefaßt werden können, weil mit ihnen in natürlicher Weise zwei "Semantiken" ver-
 bindbar sind, nämlich eine deskriptive (statische) Semantik (Semantik des
 Sprachkonstrukts = ein Sachverhalt) und eine algorithmische (dynamische) Semantik
 (Semantik des Sprachkonstrukts = eine Funktion). Solche Sprachkonstrukte sind typisch
 Rekursionen,
 Rewrite-Rules und
 Horn-Clausen-Mengen.

Dementsprechend haben sich die Bemühungen der letzten Jahre in zwei Richtungen
 entfaltet: Die Entwicklung von korrekten Transformationen, mit denen man von beliebigen
Problemspezifikationen zu rekursiven (Rewrite, Horn-Clausen) Programmen (siehe
 z.B. Kowalski [29]) für die Probleme kommt, (wobei man noch keinen besonderen Wert
 auf die Effizienz der Programme legt) und auf die Entwicklung von korrekten Transfor-
 mationsregeln, mit denen man von rekursiven (Rewrite, Horn-Clausen) Programmen zu
äquivalenten, aber effizienzmäßig besseren rekursiven und schließlich zu iterativen
 Programmen kommt. Wir zeigen die dazu entwickelten Ideen wieder an Beispielen.

Beispiel für die Synthese eines rekursiven Programms durch Anwenden von Transfor-
 mationsregeln ausgehend von der Problemspezifikation (aus Manna, Waldinger [34],
 [35]). (Wir hoffen, daß die in diesem Beispiel verwendete Notation für Sprach-
 konstrukte und Darstellung von Entwurfsvorgängen selbsterklärend ist).

Spezifikation des expliziten Problems "Maximum":

Gegeben: a, n (a Zahlenfolge der Länge n).

Gesucht: z

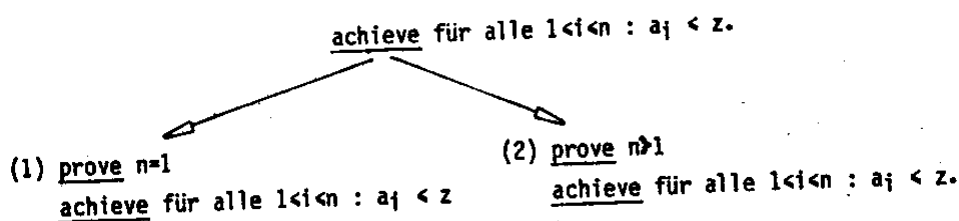
sodaß: für alle i mit $1 < i < n$: $a_i < z$.

Wir geben dem (noch nicht vorhandenen) lösenden Programm einen Namen und erzeugen ein
 Lösungsprogramm durch das Sprachkonstrukt "achieve":

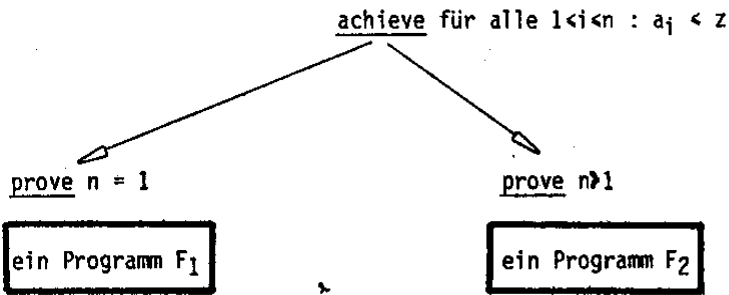
$\text{max}(n, a)$ (output: z):

achieve für alle $1 < i < n$: $a_i < z$. Ein nicht algorithmisches "Programm".

Rekursive Programme entstehen durch Zerlegen eines Problems in Unterprobleme:



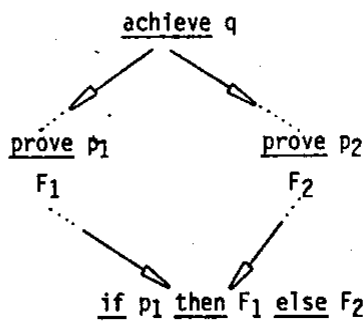
(Diese Zerlegung hat zum Ziel, daß man schließlich zu einer Situation kommen möchte, wo



(Von dieser Situation kommt man mit der folgenden Transformationsregel schließlich zum Ziel:)

Transformationsregel für if:

Aus einer Entwurfssituation

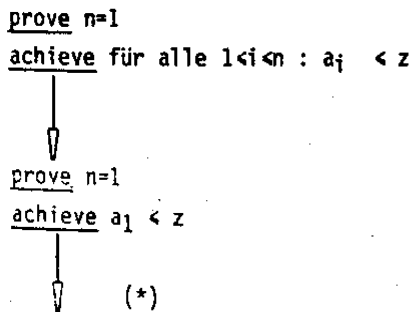


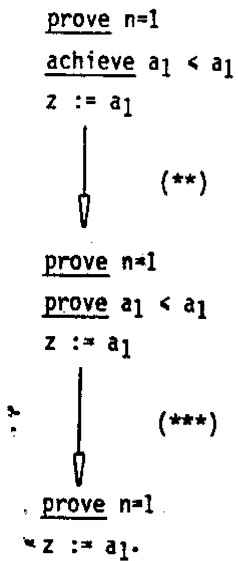
kann man zu

übergehen, falls man "p₁ oder p₂" weiß.

Wir beschäftigen uns zunächst mit dem Entwurfsproblem (1):

Folgende Transformationen sind möglich (verwendete Transformationsregeln siehe weiter unten):





Kreationsregel für beliebige Anweisungen F (verwendet in (*)):

Von $\underline{\text{achieve } p}$
 kann man zu $\underline{\text{achieve } wp(F,p)}$
 F
 übergehen.

$wp(F,p)$ ist hier die "weakest precondition (schwächste Vorbedingung, Dijkstra [18]) für F bezüglich p", das ist die Eigenschaft, die die Werte der Variablen vor Ausführung von F genau dann haben müssen, wenn die Werte der Variablen nach Ausführung von F die Eigenschaft p haben. (Oder extensional ausgedrückt: $wp(F,p)$ hat als Extension genau die Menge aller "Zustände" (Wertekombinationen der Variablen), die durch Ausführung von F in Zustände übergehen, die p erfüllen).

Z.B. $wp(x := t(x), p(x)) = p(t(x))$,
 also $wp(z := a_1, a_1 < z) = (a_1 < a_1)$.

Eliminierungsregel für achieve (verwendet in (**)):

Von (achieve p) kann man zu (prove p) übergehen.

Eliminierungsregel für prove (verwendet in (***)):

(prove p) kann man streichen, wenn sich p an der betreffenden Stelle (unter den dort wirksamen Voraussetzungen) beweisen läßt.

(Die bisher angegebenen unscheinbaren Regeln schließen

einerseits die gesamte Logik ein

(weil (prove p) die Aufforderung enthält, p zu beweisen, um dann (prove p) eliminieren zu können) und

andererseits die Charakteristik der wesentlichen Programmiersprachenkonstrukte (weil achieve p die Aufforderung enthält, ein geeignetes F vorzuschlagen, sodaß $wp(F,p)$ an der betreffenden Stelle (durch eine weitere Programmverfeinerung) erreichbar oder (aufgrund des bereits konstruierten Programms) beweisbar ist).

Beachte hier und an späteren Stellen, daß der Vorschlag für ein geeignetes F vom Problemlöser kommen muß. Die Transformationsregeln kontrollieren nur die Korrektheit jedes einzelnen Entwurfsschrittes.

Wir beschäftigen uns jetzt mit dem Entwurfsproblem (2):

Folgende Transformationen sind möglich (zusätzlich verwendete Transformationsregeln siehe unten):

prove $n \triangleright 1$

achieve für alle $1 < i < n : a_i < z$

↓ (daß diese Zerlegung des Problems etwas bringt, braucht einen "Einfall")

prove $n \triangleright 1$

achieve (für alle $1 < i < n-1 : a_i < z$) und

$a_n < z$
 (*) ↓

prove $n \triangleright 1$

achieve (für alle $1 < i < n-1 : a_i < z$)

achieve $a_n < z$

prove für alle $1 < i < n-1 : a_i < z$

(**) ↓

prove $n \triangleright 1$

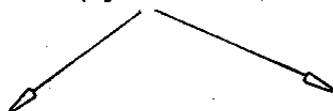
$\max(n-1, a)$

assert für alle $1 < i < n-1 : a_i < z$

achieve $a_n < z$

prove für alle $1 < i < n-1 : a_i < z$

(vgl. Problem (1))



prove $n \geq 1$
 $\max(n-1, a)$
assert für alle $1 < i < n-1 : a_i < z$
prove $a_n < z$
prove für alle $1 < i < n-1 : a_i < z$

(Verwenden von logischen Schlußregeln)

prove $n \geq 1$
 $\max(n-1, a)$
prove $a_n < z$

prove $n \geq 1$
 $\max(n-1, a)$
assert für alle $1 < i < n-1 : a_i < z$
 $z := a_n$
prove für alle $1 < i < n-1 : a_i < z$
 (***)

prove $n \geq 1$
 $\max(n-1, a)$
assert für alle $1 < i < n-1 : a_i < z$
prove für alle $1 < i < n-1 : a_i < a_n$
 $z := a_n$

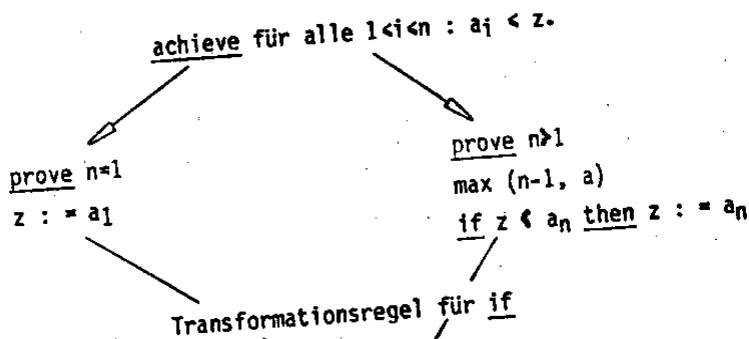
(Verwenden von logischen Schlußregeln)

prove $n \geq 1$
 $\max(n-1, a)$
prove $z < a_n$
 $z := a_n$

Transformationsregel für if

prove $n \geq 1$
 $\max(n-1, a)$
if $z < a_n$ then $z := a_n$

Wir haben also folgende Situation:



```

max(n,a): (output z)
  if n=1 then z := a1
    else max(n-1,a)
      if z < a_n then z := a_n

```

Das ist das synthetisierte korrekte Programm.

Regel für die gleichzeitige Erfüllung von Zielen (verwendet in (*)):

Von	<u>achieve</u> (p und q)	
kann man zu	<u>achieve</u> p	(lies: schreibe ein Programmstück, das p erfüllt;
	<u>achieve</u> q	schreibe ein Programmstück, das q erfüllt;
	<u>prove</u> p	beweise, daß p noch immer erfüllt ist)
übergehen.		

(Warum kann man nicht einfach zu achieve p übergehen?)

achieve q

Regel für die Generierung von Rekursionen (verwendet in (**)):

Wenn man bei der Bearbeitung eines Programmes $f(x)$ ausgehend von achieve $p(x)$ die Situation achieve $p(t(x))$ erreicht, wo $x \rightarrow t(x)$ in Bezug auf eine noethersche Ordnung \succ (Terminationsgarantie!), dann kann man wie folgt ersetzen:
achieve $p(t(x))$ durch $f(t(x))$.

Regressionsregel (verwendet in (***)):

Von	F	kann man zu	<u>achieve</u> $wp(F,p)$	übergehen.
	<u>achieve</u> p		F	

Eine allgemeine Strategie zur Transformation rekursiver Programme besteht in folgenden Dreischritt:

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

(siehe Burstall, Darlington [17], [16]).

"Entfalten" besteht im wesentlichen 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").

Genauer sind z.B. die folgenden Transformationen zur Umformung rekursiver Programme nützlich:

Definitionsregel:

Man kann jederzeit eine neue Gleichung einführen, deren linke Seite keine Instanz einer früheren Gleichung ist.

Substitutionsregel:

Man kann jederzeit eine Gleichung einführen, die aus einer früheren Gleichung durch Einsetzen (von Termen für Variable) entsteht.

Entfaltungsregel:

Wenn $E \Leftarrow E'$ und $F \Leftarrow F'$ Gleichungen sind und in F' eine Instanz $E_x[t]$ von E vorkommt, dann darf man

$$F \Leftarrow F''$$

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

Wobei-Regel:

Von einer Gleichung $E \Leftarrow E'$ kann man zu

$$(E \Leftarrow E' t_1, \dots, t_n [u_1, \dots, u_n], \text{ wobei } (u_1, \dots, u_n) = (t_1, \dots, t_n))$$

übergehen (hier sind t_1, \dots, t_n Terme und u_1, \dots, u_n Variable).

(Die Wobei-Regel kann mehrmaliges Ausrechnen ein und desselben Terms vermeiden!)

Regel über das Anwenden algebraischer Gesetze:

Man kann von einer Gleichung $E \Leftarrow E'$ zu einer Gleichung $E \Leftarrow E''$ übergehen, wo E'' aus E' durch Anwenden der für die beteiligten Grundfunktionen gültigen Gesetze (Assoziativität, Kommutativität ...) entsteht.

Falten:

Wenn $E \Leftarrow E'$ und $F \Leftarrow F'$ Gleichungen sind und in F' eine Instanz $E'_x[t]$ von E' vorkommt, dann darf man

$$F \Leftarrow F''$$

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

Diese 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 Programms in ein effizienteres:

Wir gehen von folgender rekursiver "Definition" der Fibonacci-Zahlen aus:

$$(1) f(0) \Leftarrow 1$$

$$(2) f(1) \Leftarrow 1$$

$$(3) f(x+2) \Leftarrow f(x+1) + f(x)$$

und transformieren in folgenden Schritten:

$$* (4) g(x) \Leftarrow (f(x+1), f(x)) \quad (\text{Definitionsregel})$$

$$(5) g(0) \Leftarrow (f(1), f(0)) \quad (\text{mit Substitutionsregel aus (4)})$$

$$(6) g(0) \Leftarrow (1, 1) \quad (\text{mit Entfaltungsregel aus (5) und (1),(2)})$$

$$(7) g(x+1) \Leftarrow (f(x+2), f(x+1)) \quad (\text{mit Substitutionsregel aus (4)})$$

$$(8) g(x+1) \Leftarrow (f(x+1)+f(x), f(x+1)) \quad (\text{mit Entfaltungsregel aus (7),(3)})$$

$$(9) g(x+1) \Leftarrow (u+v, u), \text{ wobei } (u, v) = (f(x+1), f(x)) \quad (\text{mit Wobei-Regel aus (8)})$$

$$(10) g(x+1) \Leftarrow (u+v, u), \text{ wobei } (u, v) = g(x) \quad (\text{mit Faltungsregel aus (9),(4)})$$

$$(11) f(x+2) \Leftarrow u+v, \text{ wobei } (u, v) = (f(x+1), f(x)) \quad (\text{mit Wobei-Regel aus (3)})$$

$$(12) f(x+2) \Leftarrow u+v, \text{ wobei } (u, v) = g(x) \quad (\text{mit Faltungsregel aus (11),(4)})$$

Zusammenfassend erhält man folgendes rekursive Programm für f :

$$f(0) \Leftarrow 1$$

$$f(1) \Leftarrow 1$$

$$f(x+2) \Leftarrow u+v, \text{ wobei } (u, v) = g(x)$$

$$g(0) \Leftarrow (1, 1)$$

$$g(x+1) \Leftarrow (u+v, u), \text{ wobei } (u, v) = g(x).$$

Die Berechnung von $f(n)$ nach dem ursprünglichen Programm braucht exponentiell viele Schritte (Additionen), nach dem zweiten nur linear viele Schritte.

Die Transformation rekursiver Programme zu iterativen kann mit denselben Regeln bewerkstelligt werden. Eine Menge von Gleichungen für Funktionen f_1, \dots, f_m ist in iterativer Gestalt, wenn jede Gleichung entweder die folgende Gestalt hat:

$f_i(x_1, \dots, x_n) \Leftarrow E$ wo E keines der Funktionssymbole f_k enthält oder E von der Form $f_k(E_1, \dots, E_n)$ ist, wo keines der E_j ein f_i enthält.

Für solche "rekursive" Gleichungen ist es nämlich klar, wie man sie sofort als while-Programme schreiben kann.

Stand der Entwicklung: Das größte derzeit konsequent nach dem Gedanken der Transformationsregeln computer-unterstützte Algorithmen-Entwurfssystem ist CIP (TU München, siehe Bauer [2], Bauer et al. [4], Bauer [3], Bauer, Wössner [5]). Eine große Fülle von Transformationsregeln der obigen und ähnlicher Art wurde für dieses System theoretisch und praktisch untersucht (siehe z.B. Partsch, Pepper [39]). Das System wird unter anderem zu seiner eigenen Entwicklung verwendet. Für ein Beispiel einer effizienzverbessernden Umwandlung einer Rekursion in eine Iteration siehe auch das Nimmenspiel im zweiten Abschnitt dieser Vorlesung.

Computer-unterstützte Strategien zur Programmsynthese

Die Computer-Unterstützung bei der schrittweisen Programmtransformation bezieht sich zunächst nur auf die Durchführung bzw. die Kontrolle der Durchführung der einzelnen Transformationen, wobei der menschliche Problemlöser auf Grund seiner Einsicht in den Problemlöseprozeß stets die wesentlichen Impulse für die durchzuführenden Transformationen gibt. Es ist nun naheliegend zu untersuchen, inwieweit die Auswahl der im konkreten Stadium des Problemlöseprozesses günstigen Transformationen nicht auch durch den Computer unterstützt bzw. vom Computer selbständig durchgeführt werden kann.

In Verbindung mit den im vorigen Abschnitt angegebenen Regeln für die Transformation rekursiver Programme werden solche Strategien z.B. in Burstall, Darlington [16] entwickelt. Durchsicht der Beispiele zeigt, daß nur an sehr wenigen Stellen im Transformationsprozeß ein "Einfall" notwendig ist. Dieser besteht meist in einer geschickten Einführung einer neuen Funktion durch Definition, die in einem gewissen Sinne "allgemeiner" ist als die eigentlich interessierende Funktion (vergleiche das Prinzip der "Generalisierung" bei Polya [41]). Diese Stelle ist im Beispiel mit * gekennzeichnet. Das System von Burstall-Darlington ist in der Lage, im wesentlichen unter Eingabe nur dieser Information durch den Benutzer für viele Beispiele der im vorigen Abschnitt beschriebenen Art, den Umwandlungsprozeß selbständig durchzuführen, wobei im wesentlichen folgende Strategie verwendet wird:

- Entfalte die vorhandenen Hilfsgleichungen und die Gleichungen für die interessierenden Funktionen (vor allem auch die durch Definition neu eingeführten Funktionen), bis keine weitere Entfaltung mehr möglich ist (bzw. bis zu einer vorgegebenen Schachtelungstiefe),
- versuche unter Anwendung der für die Grundoperationen vorhandenen Gesetze bzw. unter Benutzung der Wobei-Regel eine Identifizierung von Teilen der rechten Seite der Gleichungen für die interessierenden Funktionen mit den rechten Seiten der definierenden Gleichungen selbst herbeizuführen, sodaß dann
- Faltungsoperationen durchgeführt werden können.

Die Anwendung von Gesetzen und der Wobei-Regel wird dabei bis knapp vor den Faltungsprozeß verschoben, um den Suchraum zu verkleinern.

Ein anderes auf die automatische Realisierung von Transformationsstrategien ausgerichtetes System ist das System LOPS von Bibel (siehe z.B. Bibel [6], Bibel, Hörning [7]). Wesentliche Charakteristika dieses Systems sind

Anwendung eines Theorem-Provers als Hilfswerkzeug an bestimmten Stellen des Transformationsprozesses und
Aufstellen von Vermutungen durch einen Beispielgenerator an anderen Stellen des Transformationsprozesses.

Die wesentlichen Basisstrategien in diesem Ansatz sind:

1. "Erraten" eines Elements,
2. Ermittlung eines vernünftigen "Bereichs" für das erratene Element und
3. "rekursive Entfaltung" der aus 1. und 2. resultierenden Problemspezifikation.

Die Strategie "Erraten" besteht im wesentlichen darin, daß eine Problemspezifikation für ein explizites Problem der Gestalt

"Gegeben: x , sodaß $E(x)$
Gesucht: y , sodaß $P(x,y)$ "

in wesentliche äquivalent umgewandelt wird in

"Gegeben: x, y' , sodaß $E(x)$ und
 $P'(x, y')$
Gesucht: y , sodaß $P(x, y)$ und
 $(y=y' \text{ oder } y \neq y')$ ".

(Im Falle, daß y eine Mengenvariable ist - was im Normalfall aus dem syntaktischen Kontext automatisch abgelesen werden kann - steht statt der letzten "Bedingung" $y=y'$ oder $y \neq y'$ die Bedingung " $y' \in y$ oder $y' \notin y$ ". Diese immer wahren Bedingungen haben zur Zweck, die automatische Problemerkennung einzuleiten).

Als $P'(x,y')$ kann irgendeine Abschwächung der Bedingung $P(x,y)$ genommen werden, d.h. eine Bedingung, die den Bereich der möglichen y' möglichst einschränkt, dabei aber von möglichst einfacher logischer Struktur ist. Die Strategie "Bereich" ist eine automatische Prozedur, ein geeignetes $P'(x,y)$ (z.B. in Klausen-Form) aus der syntaktischen Zerlegung von $P(x,y)$ (z.B. in Klausen-Form) zu finden. Jede Kombination von Klausen in $P(x,y)$, die ungleich P selbst ist, könnte zur Bildung von $P'(x,y')$ herangezogen werden (in der Tat entsprechen verschiedene Auswahlen von P' oft verschiedenen Algorithmen zur Lösung des vorgegebenen Problems). Die Strategie "Bereich" versucht eine Auswahl dadurch zu treffen, daß die

- Anzahl der Elemente y' , die $P'(x,y')$ erfüllen, möglichst klein wird,
- die Entscheidung über die Gültigkeit von $P'(x,y')$ für vorgegebene y' mit den vorhandenen Operationen komplexitätsmäßig möglichst einfach wird
- und sowohl der Fall " $P'(x,y')$ " als auch der Fall "nicht $P'(x,y')$ " eintreten kann.

Der erwähnte Beispielgenerator ist unter anderem für die automatische Beurteilung dieser Kriterien in konkreten Fällen gedacht.

Die Strategie "rekursive Entfaltung" besteht im äquivalenten Umwandeln der durch die Strategien "Erraten" und "Bereich" erhaltenen Formel, wobei

mit Hilfe des Theorem-provers nachgeprüft wird, ob systematisches Ersetzen von y durch Terme der Gestalt $t(y)$ zu äquivalenten Formeln führt.

Welche Terme an welchen Stellen günstig zu ersetzen sind, um eine äquivalente Formel zu erhalten, wird oft durch die syntaktische Struktur der Formeln nahegelegt, wobei sich gewisse immer wiederkehrende Rekursionsschemata anbieten.

Es werden zwei Beispiele aus Bibel [6] und Bibel, Hörnig [7] angegeben (für den Fall einer Problemstellung mit Zahlvariablen und einer Problemstellung mit Mengenvariablen y).

Beispiel Problem der Maximumbildung:

Gegeben: S (Menge),

sodaß: $S \neq \emptyset$.

Gesucht: m ,

sodaß: $m \in S, S < m$,
 $\text{Max}(S,m)$

($S < m : \{ \implies \}$ Für alle $x \in S : x < m$).

Anwenden der Strategie "Erraten" liefert die (äquivalente) Problemstellung :

Gegeben: S, m' ,

sodaß: $P'(S, m')$,

$S \neq \emptyset$.

Gesucht: m ,

sodaß: $\text{Max}(S, m)$,

$(m=m' \text{ oder } m \neq m')$.

Als $P'(S, m')$ bieten sich hier entweder $m' \in S$ bzw. $S < m'$ an. Nachdem $S < m'$ das Element fast völlig unbestimmt läßt, während $m' \in S$ die möglichen m' stark beschränkt, liefert die Strategie "Bereich" die Entscheidung für $P'(S, m') : \langle \implies \rangle m' \in S$.

Die triviale Alternative $(m=m' \text{ oder } m \neq m')$ hat die Kraft, die Problemstellung in zwei Teile aufzuspalten:

Gesucht m , sodaß $(m' \in S, S \neq \phi \implies \text{Max}(S, m), m=m')$

und

Gesucht m , sodaß $(m' \in S, S \neq \phi \implies \text{Max}(S, m), m \neq m')$.

Die Strategie "rekursive Entfaltung" legt nahe, vor allem in der zweiten Alternative zu versuchen, durch die Substitution $S \dashrightarrow S - \{m'\}$ eine (unter den gegebenen Bedingungen) äquivalente Formel zu erhalten. Daß $S - \{m'\}$ der geeignete Term ist, wird nahegelegt, wenn man z.B. mit einer der Bedingungen in $\text{Max}(S, m)$ beginnt und schaut, in welcher Weise sie sich unter der Zusatzbedingung $m \neq m'$ äquivalent umformen läßt:

$$m' \in S \implies (m \in S, m \neq m' \langle \implies \rangle m \in S - \{m'\}, m \neq m')$$

(An solchen Stellen ist ein automatischer Beweiser das wesentliche Werkzeug). Wenn man diese Substitution konsequent durchführt, entdeckt der automatische Beweiser, daß das ursprüngliche Problem in folgende äquivalente Form übergeht:

(*) Gegeben: S, m' ,

sodaß $m' \in S$,

Gesucht: m

sodaß $(S \neq \phi \dashrightarrow \text{Max}(S, m))$ oder

$$S - \{m'\} \neq \phi \dashrightarrow \text{Max}(S - \{m'\}, m), m' \in m, S \neq \{m'\}$$

Der für die konstruktive Durchführung wichtige Term $m' \in m$ anstelle von $m' \neq m$ wird im Wechselspiel zwischen Beispielgenerator und automatischem Beweiser nahegelegt. (Siehe auch die grundsätzliche Beobachtung zur Erzeugung von neuem Wissen aus der Betrachtung von Beispielen im 2. Abschnitt).

Eine Problemspezifikation wie die in (*) kann nun entweder als Programm für einen automatischen Beweiser mit spezieller Kontrolle zur zielstrebigen Durchführung von Beweisen bei rekursiver Problemdefinition (PROLOG-artige Interpreter) genommen werden. Oder es wird das Beweisverhalten direkt beschrieben, d.h. der dementsprechende rekursive Algorithmus in einer algorithmischen Sprache angegeben:

$\max(S)$:

$m' := \text{such that } m' \in S$
 if $S = \{m'\}$ then return m' ;
 $m_0 := \max(S - \{m'\})$
 if $m_0 < m'$ then return m'
 else return m_0 .

Beispiel Partition einer Menge:

Gegeben: S (Menge), a (Element),

sodaß: $a \in S$.

Gesucht: S_1 ,

sodaß: $S_1 \subseteq S - \{a\}$,

Für alle x : $(x \in S_1 \iff x \in S, x \neq a)$.
 (Part (S, a, S_1)).

Anwenden der Strategie "Erraten" liefert die Problemstellung

Gegeben: S, a, u

sodaß: $P'(S, a, u)$,

$a \in S$.

Gesucht: S_1 ,

sodaß: $\text{Part}(S, a, S_1)$,

$(u \in S_1 \text{ oder } u \notin S_1)$.

Eine erste Wahl für $P'(S, a, u)$ ergibt sich aus der Bedingung $S_1 \subseteq S - \{a\}$ in $\text{Part}(a, S, S_1)$. Die dementsprechende Bedingung P' ist $u \in S - \{a\}$ (beachte: S_1 ist eine Mengenvariable, u eine Variable über Elemente).

Wieder spaltet sich das Problem durch die Alternative $(u \in S_1)$ in zwei Fälle auf. Aus der syntaktischen Struktur der Bedingung $u \in S - \{a\}$ ergibt sich als erster Hinweis für eine mögliche Rekursion die Ersetzung von S durch $S - \{u\}$.

Die Überprüfung, in welcher Weise die Formel nach Ersetzen $S \rightarrow S - \{u\}$ äquivalent erhalten wird, ist wieder eine Aufgabe, die von einem automatischen Beweiser gelöst werden kann. In diesem Fall erhält man die Formulierung:

Gegeben: S, a, u ,

sodaß $u \in S - \{a\}$,

$a \in S$.

Gesucht: S_1 ,

sodaß $\text{Part}(S - \{u\}, a, S_1 \cup \{u\})$, $u \in S_1$ oder

$\text{Part}(S - \{u\}, a, S_1)$, $u \notin S_1$.

Um die algorithmisch nicht brauchbare Bedingung $u \in S_1$ bzw. $u \notin S_1$, die das zu konstruierende S_1 involviert, in eine brauchbare Bedingung umzuwandeln, ist wieder ein Beispielgenerator nützlich, der an einem Beispiel "erkennt" (und dann zum allgemeinen Beweis vorschlägt), warum $u \notin S_1$ eintreffen kann, nämlich weil $u \triangleright a$. Damit geht die Spezifikation schließlich über in

Gegeben: S, a, u ,
 sodaß $u \in S - \{a\}$,
 $a \in S$.

Gesucht: S_1 ,
 sodaß $\text{Part}(S - \{u\}, a, S_1 \cup \{u\})$, $u \ntriangleleft a$ oder
 $\text{Part}(S - \{u\}, a, S_1)$, $u \triangleright a$.

Es ist klar, wie man eine solche Problemspezifikation in einen rekursiven und dann iterativen Algorithmus umwandeln kann.

Die Entwicklung von allenfalls automatisierbaren Strategien zur Programmtransformation ist auch ein wesentliches Ziel in den Arbeiten von Manna seit 1970 (siehe z.B. Manna, Waldinger [33]). Da wir aber im nächsten Abschnitt näher auf Manna's Zugang zur Programmsynthese von automatischen Existenzbeweisen her eingehen werden, sollen diese Gedanken hier nicht weiter verfolgt werden.

Stand der Entwicklung: Die automatische Synthese gelingt bisher nur für relativ einfache Programme. Der minimum-cost-spanning-tree Algorithmus (siehe z.B. Aho, Hopcroft, Ullman [1]) ist z.B. ein Algorithmus, der typisch ist für die Algorithmik-Klasse, die sich in absehbarer Zeit automatisch synthetisieren lassen werden.

Computer-unterstützte Extraktion von Algorithmen aus Existenzbeweisen

Beim Paradigma "Computer-unterstützte Programmtransformationen" war der Grundzyklus:

Mensch schlägt Transformation vor,
 Maschine führt Transformation aus bzw. kontrolliert sie,

Beim Paradigma "Computer-unterstützte Strategien zur Programmanalyse" war der Grundzyklus:

Maschine entwickelt gemäß gewissen Strategien selbst Vorschläge für Transformationen (mit gewissen Hilfen),
 Maschine führt Transformationen aus bzw. kontrolliert sie.

In diesem Abschnitt soll ein anderer Gedanke verfolgt werden, mit welchem die automatische oder halbautomatische Entwicklung (rekursiver) Programme aus Problemspezifikationen ermöglicht werden soll: Die Extraktion von Algorithmen aus automatischen oder halbautomatischen Existenzbeweisen. Wir gehen aus von einem expliziten Problem der Form

Gegeben: x
 sodaß $E(x)$.
 Gesucht: y
 sodaß $P(x,y)$.

Angenommen man hätte einen Beweis B der zugehörigen Existenzaussage (automatisch, halbautomatisch oder händisch gefunden):

(*) Für alle x existiert ein y , sodaß $P(x,y)$.

Dann wird in diesem Beweis normalerweise sehr viel an Information stecken, wie man aus einem beliebigen x ein geeignetes y mit $P(x,y)$ findet, denn Existenzbeweise verlangen (je nach verwendeter Logik mehr oder weniger explizit) die Angabe von "Beispielen" y , die aus dem vorhandenen "Material" (den Konstanten) mit den vorausgesetzten elementaren Operationen gebildet werden können.

Die automatische oder halbautomatische Synthese von Programmen läßt sich deshalb auch zerlegen in die zwei Aufgaben:

1. Automatischer oder halbautomatischer Beweis der Existenzaussage (*).
2. Automatische Extraktion des im Beweis von (*) konstruierten "lösenden" Terms aus dem Beweis.

Dieser Gedanke zur (halb)automatischen Programmsynthese wurde Anfang 1970 stark verfolgt. Zwischenzeitlich wurde 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.

Zunächst zwei Beispiele, was mit "Extraktion des lösenden Terms aus einem Existenzbeweis" gemeint ist.

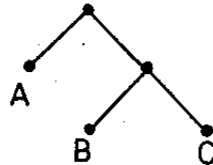
Beispiel:

Betrachte das Problem

Gegeben: Nichts.

Gesucht: y ,

sodaß y ist die Anzahl der Buchstaben, die sich an den Blättern des folgenden Baumes befinden



Definition der vorkommenden Grundbegriffe (als Horn-Clausen):

$\text{Anzahl}(x,1) \leftarrow \text{Blatt}(x)$

$\text{Anzahl}(\text{cons}(x,y),w) \leftarrow \text{Anzahl}(x,u), \text{Anzahl}(y,v), u+v=w$

(Für " $\text{Anzahl}(x,y)$ " ließ " y ist die Anzahl der Blätter am Baum x ").

Wir beweisen: Es existiert ein y , sodaß $\text{Anzahl}(\text{cons}(A,\text{cons}(B,C)),y)$.

Dieser Existenzbeweis kann durch einen Resolutionsbeweiser dadurch geliefert werden, daß aus der Horn-Clause

$\leftarrow \text{Anzahl}(\text{cons}(A,\text{cons}(B,C)),y)$

die leere Clause (ein Widerspruch) abgeleitet wird (unter Voraussetzung des Wissens

$\text{Blatt}(A) \leftarrow, \text{Blatt}(B) \leftarrow, \text{Blatt}(C) \leftarrow, 1+1 = 2 \leftarrow, 1+2 = 3 \leftarrow$).

Beweis: $\leftarrow \text{Anzahl}(\text{cons}(A,\text{cons}(B,C)),y)$

$\leftarrow \text{Anzahl}(A,u), \text{Anzahl}(\text{cons}(B,C),v), u+v=y$ Substitution: $u=1$

$\leftarrow \text{Blatt}(A), \text{Anzahl}(\text{cons}(B,C),v), 1+v=y$

$\leftarrow \text{Anzahl}(\text{cons}(B,C),v), 1+v=y$

$\leftarrow \text{Anzahl}(B,u'), \text{Anzahl}(C,v'), u'+v'=v, 1+v=y$ Substitutionen: $u'=1, v'=1$

$\leftarrow \text{Blatt}(B), \text{Blatt}(C), 1+1=v, 1+v=y$

$\leftarrow 1+1=v, 1+v=y$ Substitution: $v=2$

$\leftarrow 1+2=y$ Substitution: $y=3$

□

Der Beweis zeigt nicht nur, daß ein gewünschtes y existiert, sondern liefert auch einen Term, nämlich 3, der als Beispiel eines Resultats genommen werden kann. Der lösende Term 3 kann mit dem Beweis mitgerechnet und bei Vorliegen des Beweises aus dem Beweis extrahiert werden.

Beispiel:

Der Satz "Jede konvergente Folge ist beschränkt", d.h.

$$\text{" } \bigwedge_{\text{Folgen } a} \bigvee_{b \in \mathbb{Q}} \bigwedge_{N \in \mathbb{N}} \bigvee_{S \in \mathbb{N}} \bigwedge_{n > N} |a_n - b| < \epsilon \implies \bigvee_{S \in \mathbb{N}} \bigwedge_{n \in \mathbb{N}} |a_n| < S \text{"}$$

kann als die Existenzaussage zum expliziten Problem

"Gegeben: a

sodaß a konvergent.

Gesucht: S

sodaß $\bigwedge_{n \in \mathbb{N}} |a_n| < S$."

aufgefaßt werden. Ein Beweis dieses Satzes liefert folgenden lösenden Term:

$$\max(\max\{|a_n| \mid n < \text{Index}(a, 1)\}, |\text{lim}(a)| + 1),$$

wo lim und Index zwei (nicht konstruktive) Grundfunktionen ("Basis-Algorithmen") sind, die

zu gegebenem konvergenten a den Grenzwert $\text{lim}(a)$ bzw.

zu gegebenem konvergenten a und vorgegebenem $\epsilon > 0$ einen Wert $\text{Index}(a, \epsilon)$ liefern,

sodaß $\bigwedge_{n > \text{Index}(a, \epsilon)} |a_n - \text{lim}(a)| < \epsilon$.

Wir sehen also, daß die Extraktion von lösenden Termen aus Existenzbeweisen ein ganz naheliegender (und bei Vorliegen des Beweises auch algorithmisch einfacher) Vorgang ist, (der oft aber nicht explizit durchgeführt wird).

Das eigentliche Problem ist also die (halb) automatische Durchführung der Existenzbeweise, die mit algorithmisch interessanten Problemen verbunden sind (und im Normalfall des Beweisinstruments der Induktion bedürfen).

Manna und Waldinger [36] schlagen ein Beweissystem vor, das mit dem Existenzbeweis gleichzeitig auch den lösenden Term mitentwickelt und induktive Schlußregeln zuläßt (Induktion als Schlußregeln und nicht als Axiomenschema). 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.

Mit den Splitting Regeln kann man Behauptungen und Ziele in ihre Bestandteile zerlegen. Z.B.

wenn

F und G		t
---------	--	---

 in der Sequenz vorhanden ist

kann man zu

F		t
G		t

 übergehen.

Transformationsregeln erlauben es, beliebige allgemeine und für verschiedene Datenbereiche spezielle Beweistechniken bzw. gültige Sätze als Schlußregeln auszunützen:

(a) Eine Transformationsregel der Gestalt $(r \Leftarrow s, \text{ falls } P)$ darf man wie folgt zur Erzeugung einer neuen Behauptung verwenden:

wenn

F		t
---	--	---

 in der Sequenz vorhanden ist,

kann man zu

$P \Leftarrow s \implies$ $F \Leftarrow [r \Leftarrow s]$		$t \Leftarrow$
--	--	----------------

 übergehen.

(Hier ist θ ein (allgemeinster) Unifikator von r und einem Unterausdruck r' von F . $F\theta[r\theta \leftarrow s\theta]$ ist der Ausdruck, der aus $F\theta$ dadurch entsteht, daß man $r'\theta = r\theta$ überall durch $s\theta$ ersetzt).

(b) Analog kann man eine Transformationsregel der Gestalt $(r \vdash \Rightarrow s, \text{ falls } P)$ wie folgt zur Erzeugung eines neuen Ziels verwenden:

wenn		G	t	in der Sequenz vorhanden ist,
Kann man zu		$P\theta$ und $G\theta[r\theta \leftarrow s\theta]$	$t\theta$	übergehen.

Verallgemeinerte Resolution:

Seien F und G ein Ziel und eine Behauptung, die zwei Unterausdrücke P_1 und P_2 enthalten, die sich mit θ allgemeinst unifizieren lassen (P_1 und P_2 seien nicht im Wirkungsbereich eines Quantors).

Wenn		F	t_1	in der Sequenz vor-
	G		t_2	handen sind,
dann darf man zu		$F\theta[P_1\theta \leftarrow \text{true}]$ und $\text{nicht}(G\theta[P_2\theta \leftarrow \text{false}])$	<u>if</u> $P_1\theta$ <u>then</u> $t_1\theta$ <u>else</u> $t_2\theta$	übergehen.

Strukturelle Induktion:
Falls man ausgehend von

	$P(a)$	$R(a, z)$	z	
in der Sequenz zu		$R(s, z')$	$t(z')$	gelangt.

dann kann man die Induktionshypothese

Wenn $u \vdash a$ dann (wenn $P(u)$ dann $R(u, f(u))$)				hinzufügen.
--	--	--	--	-------------

\leftarrow ist hier eine wohl-fundierte Ordnung, f ist das Programm, das man konstruieren möchte.

(Diese Induktionshypothese erlaubt es, durch Resolution wie folgt weiter zu schließen:

	<u>true</u> und nicht (wenn $s \leftarrow a$ dann (wenn $P(s)$ dann <u>false</u>))	$t(f(s))$
--	---	-----------

das ist

	$s \leftarrow a$ und $P(s)$	$t(f(s))$
--	-----------------------------	-----------

d.h. als Beweisziele bleiben noch $s \leftarrow a$ ("das Argument für f hat sich verkleinert") und $P(s)$ ("das Argument s erfüllt die Eingabebedingung").

Beispiel für die Konstruktion eines Programmes als lösender Term eines Existenzbeweises:

Problem: Quotient und Rest bei ganzzahliger Division.

Als Startzeilen einer Sequenz stellt man dieses explizite Problem wie folgt dar:

Behauptungen	Ziele	lösende Terme	
		$div(i,j)$	$rem(i,j)$
1. $0 \leq i$ und $0 \leq j$ (Eingabebedingung)	2. $i = y \cdot j + z$ und $0 \leq z < j$ (Ausgabebedingung)	y	z

Als Grundwissen über $=$, $<$ fügen wir hinzu

3. $u = u$			
4. $(u < v \Rightarrow \text{nicht}(v < u))$			

Durch Anwenden der Splitting Regeln erhält man

5. $0 \leq i$			
6. $0 \leq j$			

Wissen über die Multiplikation verwendet man als Transformationsregeln, und zwar:

$$0 \cdot v \Leftrightarrow 0$$

$$(u+1) \cdot v \Leftrightarrow u \cdot v + v.$$

Anwendung der ersten dieser Transformationsregeln auf Ziel 2. führt zu:

	7. $i=0+z$ und $0 < z < j$	0	z
--	----------------------------	---	---

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

	8. $i=z$ und $0 < z < j$	0	z
--	--------------------------	---	---

Resolution angewandt auf 3. und 8. führt zu

	9. $0 < i < j$	0	i
--	----------------	---	---

Nochmals Resolution angewandt auf 5. und 9. führt zu

	10. $i < j$	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 Nr. 2 kann man jetzt die zweite Transformationsregel für die Multiplikation anwenden und erhält

	11. $i=y_1 \cdot j + z$ und $0 < z < j$	y_1+1	z
--	--	---------	---

Die Transformationsregel $u=v+w \Leftrightarrow u-v=w$, angewandt auf 11., ergibt:

	12. $i-j=y_1 \cdot j + z$ und $0 < z < j$	y_1+1	z
--	--	---------	---

Ziel 12. hat nun wieder genau die Gestalt des ursprünglichen Zieles 2. An dieser Stelle wird die entsprechende Induktionshypothese eingeführt:

13. Wenn $(u_1, u_2) < (i, j)$, dann (wenn $0 < u_1$ und $0 < u_2$, dann $u_1 = \text{div}(u_1, u_2) \cdot u_2 + \text{rem}(u_1, u_2)$ und $0 < \text{rem}(u_1, u_2) < u_2$)			
--	--	--	--

Durch Resolution zwischen 12. und 13. erhält man

	14. $(i-j, j) \ll (i, j)$ und $0 \ll i-j$ und $0 \ll j$	$\text{div}(i-j, j)+1$	$\text{rem}(i-j, j)$
--	--	------------------------	----------------------

Jetzt muß eine geeignete wohlfundierte Ordnung zur Verfügung stehen, z.B. die durch die folgende Transformationsregel definierte Ordnung \ll' :

$$(u_1, u_2) \ll' (v_1, v_2) \implies \text{true, wenn } u_1 \ll v_1.$$

Dadurch entsteht das neue Ziel

	15. $i-j \ll i$ und $0 \ll i-j$ und $0 \ll j$	$\text{div}(i-j, j)+1$	$\text{rem}(i-j, j)$
--	--	------------------------	----------------------

Daraus folgt im wesentlichen durch Resolution mit den Behauptungen 6. ($0 \ll j$) und 4. ($u \ll v \implies \text{nicht}(v \ll u)$)

	16. $\text{nicht}(i \ll j)$	$\text{div}(i-j, j)+1$	$\text{rem}(i-j, j)$
--	-----------------------------	------------------------	----------------------

(D.h. man weiß in diesem Stadium des Beweises, daß im Fall $j \ll i$ Quotient und Rest von i und j durch $\text{div}(i-j, j)+1$ und $\text{rem}(i-j, j)$ "rekursiv" bestimmt werden können).

Aus 10. und 16. kann man schließlich durch einen (etwas modifizierten) Resolutionsschritt

		$\text{div}(i, j)$	$\text{rem}(i, j)$
	19. <u>true</u>	<u>if</u> $i \ll j$ <u>then</u> 0 <u>else</u> $\text{div}(i-j, j)+1$	<u>if</u> $i \ll j$ <u>then</u> i <u>else</u> $\text{rem}(i-j, j)$

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.

Einen sehr interessanten Gedanken, in welcher Weise man die Information, die in Existenzbeweisen steckt, noch für algorithmische Zwecke verwenden kann, gibt Goad [21] an. Dazu muß man das Konzept eines Zwei-Parameter-Algorithmus (Goad [22]) einführen: Man betrachtet ein explizites Problem der Gestalt

Gegeben: p, x ($p, x \dots$ zwei Eingabeparameter(listen)).

Gesucht: y ,

sodaß $P(p, x, y)$.

Ein Algorithmus A zur Lösung des Problems hat also die Eigenschaft:

Für alle p, x : $P(p, x, A(x, y))$.

In Fällen, wo sich der Parameter p bei Anwendungen sehr viel langsamer ändert als der Parameter x ("sweep coherence") kann man oft

ausgehend von einem einfachen, naheliegenden, leicht zu findenden Algorithmus A, der gleichzeitig p und x als Eingaben nimmt (Zwei-Parameter-Algorithmus, Einstufen-Algorithmus)

den Algorithmus für jedes feste p symbolisch exekutieren und durch symbolische Transformationen vereinfachen ("Entfalten") (das führt für jedes p zu einem Algorithmus A'_p)

und dann den jeweiligen Einparameter-Algorithmus A'_p auf eine Reihe von Eingaben x anwenden.

Diese Methode führt oft in völlig automatischer Weise zu erstaunlich effizienten Algorithmen-Familien $\{A'_p\}$.

Beispiel der Entfaltung eines Zweiparameter-Algorithmus in eine Familie von Einparameter-Algorithmen:

Problem:

Gegeben: p (eine als Liste geschriebene Menge),
 x (ein Element).

Gesucht: y

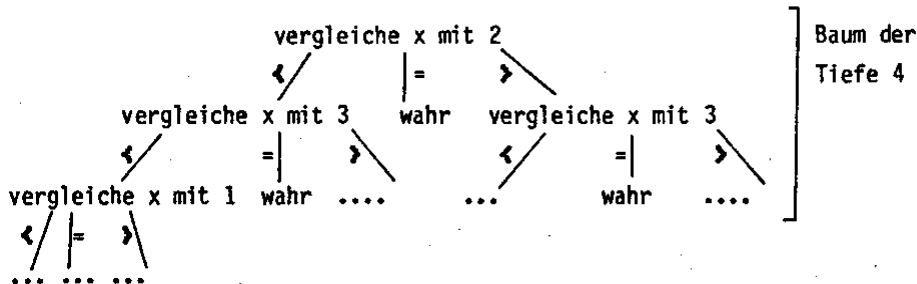
sodaß $y =$ "wahr", wenn $x \in p$
 "falsch", sonst

$P(p, x, y)$.

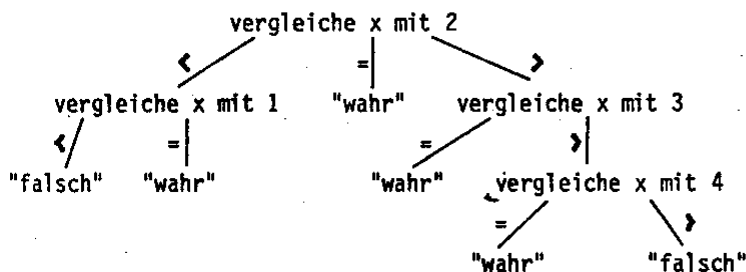
Ein naheliegender Zweiparameter-Algorithmus:

$A(p, x) =$ if $p = \phi$ then "falsch" else
if $x \text{head}(p)$ then $A(\text{tail}(p), x)$ else
if $x \text{>head}(p)$ then $A(\text{tail}(p), x)$ else
 "wahr".

Entfaltung des Algorithmus für $p = (2,3,1,4)$:



Man kann aus diesem Baum auf völlig automatische Weise alle redundanten Knoten "pflücken". Ein Knoten ist redundant, wenn das Ergebnis der betreffenden Abfrage bereits durch die vorhandenen Abfragen determiniert ist (siehe "Decision Trees" in Aho, Hopcroft, Ullman [1]). In diesem Beispiel führt dies zu



Es kann gezeigt werden (Robson [46]), daß die mittlere Tiefe der dadurch entstehenden Bäume $O(\log(\text{Länge von } p))$ ist gegenüber einer Rechenzeit von $O(\text{Länge von } p)$ beim zweiparametrischen, einfachen Algorithmus.

Natürlich könnte man in dem betrachteten Beispiel auch eine Familie $\{B_p\}$ von einparametrischen Algorithmen "erfinden", deren Rechenzeitverhalten ebenfalls von der Ordnung $\log(\text{Länge von } p)$ ist, z.B.

$B_p :=$ Wende binäre Suche auf eine sortierte Version von p an.

Man beachte aber den entscheidenden Unterschied:

B_p muß erfunden werden, d.h. man braucht die Einsicht, daß das Suchen in sortierten Listen in logarithmischer Zeit möglich ist,

A'_p sind automatisch entstanden aus einem Algorithmus, für welchen man kaum eine Idee brauchte.

Symbolisches Exekutieren für den speziellen Wert $y := 0,5$ liefert den vereinfachten Algorithmus

$$u_{0,5}(x) := \text{if } x < 1 \text{ then } 1,5 \\ \text{else } x+1.$$

Für $y := 0,5$ würde das Problem aber auch durch den noch einfacheren Algorithmus

$$u'(x) := x+1$$

gelöst. u' rechnet aber nicht dieselbe Funktion wie $u_{0,5}$. Deshalb ist eine weitere Korrektheitserhaltende Transformation von $u_{0,5}$ nicht möglich. Vereinfachen des zum Algorithmus gehörigen Existenzbeweises (der aus der Problemspezifikation entstanden ist), führt aber zu $x+1$, siehe Goad [21].

Stand der Entwicklung: Der Gedanke der Extraktion von Algorithmen aus Existenzbeweisen zur Programm-Synthese war bisher nur bei sehr einfachen Spielbeispielen erfolgreich. Manna und Waldinger haben jedoch ein Beispiel einer sehr nicht-trivialen Algorithmensynthese dieser Art angegeben, die zwar mit den heutigen Beweisern noch nicht automatisch durchgeführt werden kann, an welcher aber die notwendigen Fähigkeiten eines solchen Beweisers als durchaus erreichbar analysiert werden (Synthese des Unifikationsalgorithmus).

Spezifikationen abstrakter Datentypen als Programme

Implizite Problemspezifikationen, speziell Spezifikationen abstrakter Datentypen durch eine Menge von Gleichungen (Guttag, Horning [24]), können

als Beschreibung von Operationen, die man als Modell des Datentyps zulassen möchte, aufgefaßt werden,

aber auch als Regeln, wie man mit den Operationen rechnet.

Die Terme, aus denen die Spezifikationen aufgebaut sind, können selbst, faktorisiert durch die durch die Axiome erzeugte Kongruenzrelation, als ein Modell des Datentyps betrachtet werden. Das heißt für die Praxis: Wenn man in einem Programm als Grundoperationen nur die Operationen eines durch Gleichungen spezifizierten Datentyps verwendet, so kann man die Terme selbst als ein erstes Modell für den Datentyp verwenden, sofern man eine effektive Methode hat, die Gleichheit von Termen modulo den Axiomen zu testen. Dieses Modell ist zwar meist nicht sehr effektiv, kann aber vom

methodologischen Gesichtspunkt aus sehr wertvoll sein, weil dadurch z.B. ein Hauptprogramm, in welchem Datentyp Operationen verwendet werden, bereits ausgetestet werden kann, bevor die Implementierung der Datentypen durchgeführt sind (siehe Lichtenberger [30], Musser [37]). Andere Situationen, in welchen das Rechnen modulo Gleichungen im Computer-unterstützten Algorithmenentwurf eine Rolle spielt, sind z.B. das Beweisen von Verifikationsbedingungen durch einen "Simplifier" und das symbolische Vereinfachen von zweiparametrischen Algorithmen, wenn man einen Parameter festhält.

Die Gleichheit von Termen modulo Gleichungen ist nun im allgemeinen ein unentscheidbares Problem. In diesem Abschnitt soll eine sehr allgemeine Methode beschrieben werden, mit welcher oft in algorithmischer Weise Entscheidungsverfahren für Termgleichheit konstruiert werden können. Wir wollen diese Methode in einen etwas allgemeineren Rahmen stellen und nennen die hier zu besprechende Algorithmenklasse "critical pair/completion-Algorithmen" (kurz: CPC-Algorithmen). Zunächst aber ein Beispiel für eine typische Problemstellung:

Beispiel:

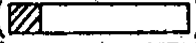
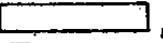
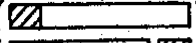

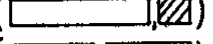

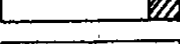
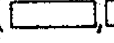
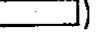

Wir betrachten folgende Spezifikation des Datentyps "Queue":

```

Remove(Newq) = Newq,
Remove(Add(q,i)) =
    if Isnew(q) then q else Add(Remove(q),i),
Front(Newq) = 0,
Front(Add(q,i)) =
    if Isnew(q) then i else Front(q),
Isnew(Newq) = true,
Isnew(Add(q,i)) = false,
Append(q1,Newq) = q1,
Append(q1,Add(q2,i2)) = Add(Append(q1,q2),i2).

```

```

( Remove(  ) =  ,
  Front(  ) =  ,
  Add(  ,  ) = 
  Append(  ,  ) =  ).

```

Als ein Modell für diesen Datentyp können wir uns die Menge aller Terme, die aus den Operationssymbolen des Datentyps gebildet sind, vorstellen, wobei allerdings Terme, die auf Grund der verlangten Gleichungen als "gleich" bewiesen werden können,

identifiziert werden sollen, versehen mit den Operationen, die durch Voranstellen der betreffenden Operationssymbole vor die Argumentterme definiert sind, z.B.

$$\text{Add}(\text{Add}(\text{Newq},1),2) = \text{Add}(\text{Add}(\text{Newq},1),2).$$

Operation,
die durch Add auf
der Termmenge be-
schrieben wird

$\text{Remove}(\text{Add}(\text{Newq},1))$ und $\text{Remove}(\text{Add}(\text{Newq},2))$ werden als Terme identifiziert, weil $\text{Remove}(\text{Add}(\text{Newq},1)) = \text{Remove}(\text{Add}(\text{Newq},2))$ aus den obigen Gleichungen folgt.

Gilt auch

$$\text{Ap}(\text{R}(\text{A}(\text{N},1),2),\text{A}(\text{N},3)) = \\ \text{Ap}(\text{A}(\text{N},1),\text{R}(\text{A}(\text{N},1),3))?$$

(Wir verwenden hier die Anfangsbuchstaben der Operationen als Abkürzungen).

Wir haben das Gefühl, daß man diese Frage einfach durch "Reduzieren beider Seiten bis es nicht mehr geht" entscheiden kann: die linke Seite liefert dann $\text{A}(\text{A}(\text{N},2),3)$ und die rechte Seite $\text{A}(\text{A}(\text{N},1),3)$. "Deshalb sind diese beiden Terme nicht gleich".

Ist diese Methode gerechtfertigt?

Es gilt nun folgender

Satz (Critical-pair-Algorithmus, Knuth-Bendix [28]):

Sei E eine endliche Menge von Gleichungen zwischen Termen erster Ordnung, sodaß die zugehörige Reduktionsrelation $\text{---}\rightarrow_E$ noethersch ist, und sei S_E ein zugehöriger Normalformenalgorithmus. Dann gilt:

S_E ist ein kanonischer Simplifikator für \approx_E gdw.

für alle kritischen Paare (p,q) bezüglich E : $S_E(p) = S_E(q)$.

Hier brauchen wir einige Begriffe: Die durch E bestimmte Reduktionsrelation $\text{---}\rightarrow_E$ ist wie folgt definiert ($s,t \dots$ Terme):

$s \text{---}\rightarrow_E t$ (s reduziert auf Grund von E zu t): $\Leftarrow \Rightarrow$

t entsteht aus s durch Ersetzen einer Unterterme der Gestalt $\sigma(a)$ durch $\sigma(b)$, wo (a,b) eine Gleichung aus E ist und σ eine Substitution.

Eine binäre Relation \rightarrow in einer Menge ist noethersch, wenn es keine unendlichen absteigende Kette

$$t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \dots$$

gibt.

Der zu E gehörige Normalformen-Algorithmus S_E ist:

$S_E(t) :=$ der Term, der durch fortgesetztes Reduzieren aus t entsteht, d.h.

$:=$ if t ist in Normalform

then t

else $S_E Sel(t)$

wo Sel eine "Selektionsfunktion" ist, die $t \rightarrow_E Sel(t)$ erfüllt (im Falle, daß t nicht in Normalform ist, d.h. mit \rightarrow_E noch weiter reduziert werden kann).

$=_E$ ist die reflexive, symmetrische, transitive Hülle von \rightarrow_E . $=_E$ ist eine Äquivalenzrelation.

Eine Abbildung $S: M \rightarrow M$ heißt kanonischer Simplifikator für eine Äquivalenzrelation \sim in der Menge $M: \langle \sim \rangle$

Für alle $t \in M$:

$$S(t) \sim t,$$

$$s \sim t \implies S(s) = S(t).$$

Es gilt: Eine Äquivalenzrelation \sim in einer Menge M ist entscheidbar genau dann, wenn es einen kanonischen Simplifikator für \sim gibt.

Die Terme (p,q) formen ein kritisches Paar bezüglich $E: \langle \sim \rangle$

Es gibt Gleichungen (a_1, b_1) und (a_2, b_2) in E und eine Stelle u in a_1 , sodaß

der Term a'_1 an der Stelle u in a_1 keine Variable ist,

$\sigma_1(a'_1) = \sigma_2(a_2)$ eine allgemeinste Instanz von a'_1 und a_2 ist für gewisse Substitutionen σ_1 und σ_2

und

p entsteht aus $\sigma_1(a_1)$ durch Ersetzen des Terms an der Stelle u durch $\sigma_2(b_2)$,

$$q = \sigma_1(b_1).$$

(D.h. p und q entstehen, indem man die "Regeln" ("Rewrite Rules") (a_1, b_1) und (a_2, b_2) auf einen "allgemeinsten übereinstimmenden Teil" der linken Seiten a_1 und a_2 anwendet).

Der obige Satz besagt also, daß das "naheliegende Verfahren", die Gleichheit von Termen durch Reduzieren beider Terme bis auf Normalform und Vergleich der beiden erhaltenen Normalformen zu entscheiden, tatsächlich korrekt ist, sofern man bei Anwendung des Verfahrens auf gewisse endlich viele kritische Paare von Termen Identität erzielt.

Beispiel:

Die durch die Rewrite-Rules Q in der Spezifikation von Queue definierte Reduktionsrelation \rightarrow_Q ist noethersch. Diesen Beweis lassen wir hier aus. Die Regeln sind "superpositionsfrei", d.h. es gibt hier keine kritischen Paare. S_Q ist deshalb trivialerweise ein kanonischer Simplifikator, der das Entscheidungsproblem \approx_Q löst.

Wenn man zu dem Gleichungssystem jedoch noch die Regel

(*) $\text{Append}(\text{Append}(q'_1, q'_2), q'_3) = \text{Append}(q'_1, \text{Append}(q'_2, q'_3))$ hinzufügt, so entsteht z.B. ein kritisches Paar durch "Überlagern" mit der Regel

$$\text{Append}(q_1, \text{Add}(q_2, i_2)) = \text{Add}(\text{Append}(q_1, q_2), i_2).$$

Eine Unifikation kann nämlich erreicht werden durch die Substitutionen

$$\sigma_1 := \{q_1 \mapsto \text{Append}(q'_1, q'_2)\} \text{ und}$$

$$\sigma_2 := \{q'_3 \mapsto \text{Add}(q_2, i_2)\}.$$

Die gemeinsam unifizierte linke Seite

$$\text{Append}(\text{Append}(q'_1, q'_2), \text{Add}(q_2, i_2))$$

kann man dann auf zwei wesentlich verschiedene Arten reduzieren zum "kritischen Paar"

$$\text{Append}(q'_1, \text{Append}(q'_2, \text{Add}(q_2, i_2))) \quad \text{Add}(\text{Append}(\text{Append}(q'_1, q'_2), q_2), i_2).$$

Beide diese Terme reduzieren zur selben Normalform

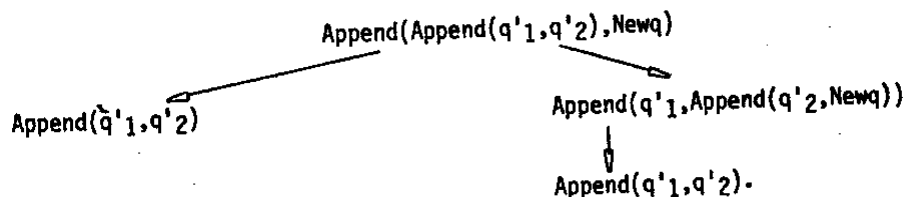
$$\text{Add}(\text{Append}(q'_1, \text{Append}(q'_2, q_2)), i_2).$$

Es gibt aber jetzt noch andere Überlagerungen, nämlich das innere Vorkommen von Append in (*) kann ebenfalls mit $\text{Append}(q_1, \text{Add}(q_2, i_2))$ überlagert werden:

$$\text{Append}(\text{Append}(q_1, \text{Add}(q_2, i_2)), q'_3)$$

$$(**) \quad \text{Append}(\text{Add}(\text{Append}(q_1, q_2), i_2), q'_3) \quad \text{Append}(q_1, \text{Append}(\text{Add}(q_2, i_2), q'_3))$$

Die beiden Terme in diesem kritischen Paar sind in Normalform. Das ist ein Zeichen dafür, daß die um (*) erweiterte Gleichungsmenge Q' nicht mehr von der Art ist, daß der zugehörige Normalformalgorithmus $\rightarrow_{Q'}$ ein kanonischer Simplifikator wäre. Es gibt noch eine Oberlagerung der Gleichung (*) mit der Gleichung $\text{Append}(q, \text{New}q) = q$, welche aber wieder zu identischen Normalformen des kritischen Paares führt:



Wenn der Kritische-Paar-Algorithmus zu einem negativen Ergebnis führt, wie bei Hinzunahme von (*) in obigem Beispiel, dann liegt der Gedanke nahe, das Paar der reduzierten Normalformen als Gleichung zu den bisherigen Gleichungen hinzuzunehmen und die neu entstehenden kritischen Paare weiter zu untersuchen. Das ist der wichtige Gedanke der Vervollständigung.

Completion-Algorithmus (Knuth-Bendix [28]):

Gegeben: E , eine endliche Menge von Gleichungen, sodaß \rightarrow_E noethersch.

Suche: F , eine endliche Menge von Gleichungen,

sodaß $\rightarrow_E = \rightarrow_F$ und

S_F ist ein kanonischer Simplifikator.

$F := E$

$C :=$ Menge der kritischen Paare von F

while $C \neq \emptyset$ do

$(p, q) :=$ ein Element von C

$(p_0, q_0) := (S_F(p), S_F(q))$

if $p_0 \neq q_0$ then

Analysiere (p_0, q_0)

$C := C \cup \{\text{neue krit. Paare}\}$

$F := F \cup \{(p_0, q_0)\}$

$C := C - \{(p, q)\}$

stop erfolgreich.

Das Unterprogramm "Analysiere" bestimmt, ob \rightarrow_f noethersch bleibt, wenn man (p_0, q_0) bzw. (q_0, p_0) zu F hinzufügt. Im ersten Fall beläßt man (p_0, q_0) ungeändert, im zweiten Fall wird die Rolle von p_0 und q_0 vertauscht. Wenn keiner der beiden Fälle zutrifft, stoppt "Analysiere" mit einer Fehlermeldung. Es gibt drei Möglichkeiten: 1. Erfolgreicher Stop: In diesem Fall erfüllt das erhaltene F die Problemspezifikation des Algorithmus. 2. Der Algorithmus stoppt mit Fehlermeldung: In diesem Fall kann man nichts Weiteres aussagen. 3. Der Algorithmus stoppt nicht: In diesem Fall ist der Algorithmus wenigstens eine partielle Entscheidungsprozedur für \rightarrow_f .

Beispiel:

Im obigen Beispiel stellt sich nach Hinzunahme der aus (**) entstandenen Gleichung heraus, daß im weiteren Verlauf des Vervollständigungsalgorithmus keine neuen Paare mehr entstehen. Der zu diesem Gleichungssystem gehörige Normalformenalgorithmus ist ein kanonischer Simplifikator.

An diesem Beispiel kann noch eine andere interessante Anwendung der CPC-Methode für die algorithmische Behandlung von Datentypen demonstriert werden: Die Assoziativität von Append kann nicht aus den anderen Axiomen des Datentyps Queue nur mit "Einsetzen" und "Ersetzen" hergeleitet werden (weil die beiden Seiten in Normalform, aber nicht identisch sind). Sie gilt also nicht für alle Modelle des Axiomensystems für Queue (Satz von Birkhoff über die Äquivalenz des Gültigkeitsbegriffs und des Ableitungsbegriffs für Gleichungstheorien). Offensichtlich gilt sie aber für das kleinste Modell, das nur aus durch variablenfreie Terme beschreibbaren Objekten besteht (das "initiale" Modell). Man sieht an diesem Beispiel: (*) führt bei Hinzunahme zu den bisherigen Gleichungen bei Anwendung des Completion-Algorithmus nicht zu einem "Widerspruch" (Gleichung der Art true = false). Das gilt allgemein.

Satz (Ersatz der Induktion, Musser [38] u.a.):

Unter gewissen Voraussetzungen an E gilt:

Eine neue Gleichung $s=t$ gilt im initialen Modell von $E \iff$

$E \cup \{s=t\}$ führt bei Anwendung des Completion-Algorithmus nicht zu einem Widerspruch.

Dies ist methodologisch deshalb interessant, weil im allgemeinen zum Beweis von Gleichungen für das initiale Modell Induktionsbeweise über den Aufbau der variablenfreien Terme notwendig sind, die meist ein gewisses Maß an Kreativität zum Erfinden der Induktionshypothese verlangen. Der obige Satz gibt ein vollständig mechanisches Verfahren (das allerdings sehr rechenintensiv sein kann).

Der Gedanke "CPC" ist ein sehr allgemein anwendbarer Algorithmientyp, der für viele verschiedene Reduktionsrelationen ein (partiell) Entscheidungsverfahren für die reflexive, symmetrische, transitive Hülle dieser Relationen liefert. Der Algorithmientyp CPC ist so alt wie die Algorithmen: der Euklidische Algorithmus kann als CPC-Algorithmus betrachtet werden (siehe Buchberger/Loos [15]). Explizit wurde der Gedanke "critical pair" + "completion" in ganz verschiedenen Bereichen unabhängig voneinander eingeführt (Buchberger [10], [11], Knuth-Bendix [28]). Für eine Übersicht über CPC siehe Buchberger/Loos [15].

Stand der Entwicklung: Ein Verifikationssystem, das hauptsächlich auf einer computer-unterstützten Manipulation mit abstrakten Datentypen basiert, ist das AFFIRM-System, siehe z.B. Gerhart et al [20]. Die Leistungsfähigkeit solcher Systeme ist mit den computer-unterstützten Programm-Verifikationssystemen vergleichbar.

Programmsynthese aus Beispielen

Über die bisherigen Ansätze hinausgehend gibt es auch zahlreiche Versuche, aus der Angabe von Beispielen von Ein/Ausgabe-Paaren für eine Problemlösung einen zugehörigen Algorithmus zu synthetisieren, der den Zusammenhang allgemein beschreibt. Diese Untersuchungen können hier aus Platzmangel nicht mehr besprochen werden. Eine Übersicht über vorhandene Methoden gibt Smith [47]. Der Einsatzbereich dieser Methoden ist beschränkt.

Ausblick

Man sieht, daß bereits ein reichhaltiges Spektrum an Ideen zur Computer-Unterstützung des Algorithmenentwurfsvorgangs vorhanden ist. Auch wurden bereits sehr vielfältige Erfahrungen in der konkreten Implementierung dieser Ideen in Systemen gewonnen. Die meisten Systeme gehen dabei von einer der in den vorangehenden Abschnitten an Beispielen beschriebenen "Philosophien" aus. Es ist jedoch auch denkbar, daß man die einzelnen Möglichkeiten, den Algorithmenentwurf durch den Computer selbst zu unterstützen, als "Moduln" in einem Gesamtsystem zur Verfügung hält, die dann vom Benutzer je nach seinem eigenen Entwurfsstil an bestimmten Stellen des Entwurfsvorgangs auf das jeweilige Stadium seines Problem/Programmtextes angewandt werden können. Ein derartiges System müßte als Werkzeuge enthalten:

universelle und spezielle Beweiser,
Simplifikatoren,
Bausteine zum Manipulieren mit abstrakten Datentypen,
Bausteine zum Umgang mit Verifikationsbedingungen,
Möglichkeiten zum Aufruf von Programmtransformationen.

Dabei sollte versucht werden, diese logisch anspruchsvollen Moduln in bestehende Systeme zur Unterstützung des strukturierten Programms und der halbautomatischen Software-Dokumentation (siehe z.B. Hesse [26]) zu integrieren. Versuche in diese Richtung werden in dem gemeinsamen Projekt der Arbeitsgruppen Siekmann (Karlsruhe), Raulefs (Kaiserslautern), Buchberger (Linz) gemacht, siehe Raulefs, Siekmann [43], Buchberger [13].

Dank: Mein Dank gilt Herrn F. Lichtenberger für wertvolle Diskussionen über die Gegenstände dieser Vorlesung.

Literatur

- [1] Aho A.H., Hopcroft J.E., Ullman J.D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, Reading, Mass. (1974).
- [2] Bauer F.L.
A Philosophy of Programming.
Lecture Notes, University of London (1973) (appeared in LN in Computer Science,
Vol.46, Springer, Berlin, 1976).
- [3] Bauer F.L.
Program Development by Stepwise Transformations - The Project CIP.
In: Program Construction (F.L.Bauer, M.Broy ed.), Lecture Notes of an
Internat. Summer School, Marktoberdorf, Lecture Notes in Computer Science,
Vol.69, Springer, Berlin (1979).
- [4] Bauer F.L., Broy M., Gnatz R., Hesse W., Krieg-Brückner B., Partsch H.,
Pepper P., Wössner H. Towards a Wide Spectrum Language to Support
Program Specification and Program Development. ACM SIGPLAN Notices,
Vol.13/12, 12-24, (also in: Program Construction, F.L. Bauer, M.Broy ed.,
LN in Computer Science, Vol.69, Springer, 543-552 (1978).
- [5] Bauer F.L., Wössner H.
Algorithmic Language and Program Development.
Prentice Hall International, London (1979).
- [6] Bibel W.
Syntax-Directed, Semantics-Supported Program-Synthesis.
Artificial Intelligence 14, 243-261 (1980).
- [7] Bibel W., Hörnig K.M.
LOPS- A System Based on a Strategical Approach to Program Synthesis.
In: Automatic Program Construction Techniques, A. Biermann, G. Guiho, I. Kodra-
toff ed., MacMillan, im Druck.
- [8] Biermann A., Guiho G., Kodratoff I. (eds.)
Automatic Program Construction Techniques.
MacMillan, im Druck.
- [9] Boyer R.S., Moore J.S.
A Computational Logic.
Academic Press, New York, London (1979).
- [10] Buchberger B.
Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen
Gleichungssystems.
Aequationes mathematicae, Vol.4/3, 374-383 (1970). (Diss., Univ. Innsbruck,
1965).
- [11] Buchberger B.
A Criterion for Detecting Unnecessary Reductions in the Construction of
Gröbner-Bases.
Invited paper: EUROSAM 79, Marseille, Lecture Notes in Comput. Scie.,
Vol.72, 3-21 (1979).
- [12] Buchberger B.
Eine Fallstudie in systematischer Algorithmenentwicklung: Algorithmus für ein
Nimmspiel.
Univ. Linz, Inst. f. Math., Bericht Nr. 162 (1980).

- [13] Buchberger B.
Beschreibung des Forschungsprojektes "Programmverifikation: Teilprojekt
Simplifikation".
Univ. Linz, Inst. f. Math., Projekt-Proposal (1981).
- [14] Buchberger B., Lichtenberger F.
Mathematik für Informatiker I. (Die Methode der Mathematik).
Springer, Heidelberg (1980).
- [15] Buchberger B., Loos R.
Algebraic Simplification.
In: Computer Algebra (Buchberger B., Collins G., Loos R. eds.), Springer, Wien,
erscheint demnächst.
- [16] Burstall R.M., Darlington J.
A Transformation System for Developing Recursive Programs.
J. ACM, Vol.24/1, Jan. 1977.
- [17] Darlington J., Burstall R.M.
A System which Automatically Improves Programs.
Acta Informatica, Vol.6, pp.41-60 (1976).
- [18] Dijkstra E.W.
A Discipline of Programming.
Prentice Hall, Englewood Cliffs, N.J. (1976).
- [19] Floyd R.W.
Assigning Meanings to Programs.
Proc. of Symp. in Applied Mathematics, Vol.19, pp.19-32
(J.T.Schwartz ed., Mathematical Aspects of Computer Science,
American Mathematical Society, Providence, R.I.) (1967).
- [20] Gerhart S.L., Musser D.R., Thompson D.H., Baker D.A., Bates R.L., Erickson R.W.,
London R.L., Taylor D.G., Wile D.S.
An Overview of AFFIRM: A Specification and Verification System.
Proc. of the IFIP Congress 1980 (S.H.Lavington ed.), pp.343-347 (1980).
- [21] Goad C.A.
Proofs of Descriptions of Computation.
Proc. 5th Conf. on Automated Deductions, Springer LN Comput. Sci. 87, 39-52
(1980).
- [22] Goad C.A.
Automatic Construction of Special Purpose Programs.
Proc. 6th Conf. on Automated Deductions, to appear.
- [23] Gordon M.J., Milner A.J., Wadsworth C.P.
Edinburgh LCF.
Lecture Notes in Comp. Science Vol.78, Springer, Berlin (1979).
- [24] Guttag J.V., Horning J.J.
The Algebraic Specification of Abstract Data Types.
Acta Informatica, Vol.10, pp.27-52 (1978).
- [25] Henke F.W., Luckham D.C.
Automatic Program Verification III: A Methodology for Verifying Programs.
Stanford Artificial Intelligence Laboratory, Memo AIM-255 (1974).

- [26] Hesse W.
Methoden und Werkzeuge für Software-Entwicklung: Ein Marsch durch die Technologie-Landschaft.
Informatik-Spektrum, 4/4, 229-245 (1981).
- [27] Hoare C.A.R.
An Axiomatic Basis for Computer Programming.
Communications of the ACM, Vol.12/10, pp.576-580 (1969).
- [28] Knuth D.E., Bendix P.B.
Simple Word Problems in Universal Algebras.
In: Leech (ed.), Computational Problems in Abstract Algebra, Proc. of Symp., Oxford 1967, Pergamon Press, New York, pp. 263-297 (1970).
- [29] Kowalski R.
Logic for Problem Solving.
North Holland, New York, Oxford (1979).
- [30] Lichtenberger F.
PL/ADT: Ein System zur Verwendung algebraisch spezifizierter abstrakter Datentypen in PL/I.
Diss., Univ. Linz, Inst. f. Mathematik (1980).
- [31] Luckham D.C., German S.M., v.Henke F.W., Karp R.A., Milne P.W., Oppen D.C., Polak W., Scherlis W.L.
Stanford Pascal Verifier User Manual.
Stanford, Computer Science Department, Report No. STAN-CS-79-731 (1979).
- [32] Manna Z.
Mathematical Theory of Computation.
McGraw-Hill, New York (1974).
- [33] Manna Z., Waldinger R.
Knowledge and Reasoning in Program Synthesis.
Artif. Intelligence 6/2, 175-208 (1975).
- [34] Manna Z., Waldinger R.
Synthesis: Dreams \Rightarrow Programs.
SRI International, Menlo Park, Calif., Techn. Note No. 156 (1977).
- [35] Manna Z., Waldinger R.
The Logic of Computer Programming.
IEEE Trans. on Software Engin., SE-4/3, 199-229 (1978).
- [36] Manna Z., Waldinger R.
A Deductive Approach to Program Synthesis.
ACM TOPLAS 2/1, 92-121 (1980).
- [37] Musser D.R.
Abstract Data Type Specification in the AFFIRM System.
Proc. of the Conf. on Specification of Reliable Software, Boston, April 3-5, 1979, pp.47-57 (1979).
IEEE Transact. on Software Eng. Vol. SE-6, No. 1, Jan. 1980.
- [38] Musser D.R.
On Proving Inductive Properties of Abstract Data Types.
Seventh ACM Symp. on Principles of Programming Languages (1980).
- [39] Partsch H., Pepper P.
A Family of Rules for Recursion Removal Related to the Towers of Hanoi Problem.
Information Processing Letters, Vol.5, 174-177 (1976).

- [40] Polak W.
Program Verification at Stanford: Past, Present, Future.
Internal Report, Stanford University, Computer Systems Laboratory (1981).
- [41] Polya G.
How to Solve It.
(Deutsche Übersetzung: Schule des Denkens. Francke Verlag, Bern (1967)).
- [42] Prawitz D.
Natural Deduction.
Almqvist&Wiksell, Stockholm (1965).
- [43] Raulefs P., Siekmann J.
Programm-Verifikation: Darstellung eines Forschungsvorhabens.
Univ. Bonn, Univ. Karlsruhe, Inst. f. Informatik, Forschungs-Proposal (1980).
- [44] Rechenberg P.
Programmieren in PL/I.
Oldenburg, München (1978).
- [45] Rechenberg P.
Persönliche Mitteilung (1980).
- [46] Robson J.
The Height of Binary Search Trees.
The Australian Comput. J. 11, 151-153 (1979).
- [47] Smith D.R.
A Survey of the Synthesis of LISP Programs from Examples.
Proc. Int. Workshop on Progr. Constr., Chateau de Bonas, Sept. 1980.
- [48] Suzuki N.
Verifying Programs by Algebraic and Logical Reduction.
Proc. of the International Conf. on Reliable Software, Los Angeles, California,
pp.473-481 (1975).
- [49] Wegbreit B.
Heuristic Methods for Mechanically Deriving Inductive Assertions.
Third International Joint Conference on Artificial Intelligence,
Stanford, California, pp.524-536 (1973).