BUCHBERGER

In: Jahrbuch Überblicke Mathematik 1976.
(B.Fuchssteiner et al. ed.).
Bibl.Institut Mannheim (BI).

Das Problem der Programmverifikation

von Bruno Buchberger

1. Das Lösen von Problemen auf Computern

Dem Programmierer sind vorgegeben:

ein „*Problem*" und
ein „*Computer*".

Er soll das gegebene Problem mit Hilfe des gegebenen Computers „*lösen*".

Was ist ein Problem?

Ein *Problem* ist charakterisiert durch

eine Menge X von möglichen „*Angaben*" (Eingaben, Inputs),
eine Menge Y von möglichen „*Resultaten*" (Ausgaben, Outputs),
eine Relation $R \subset X \times Y$.
(Für „$(x, y) \in R$" lies: „y ist eine mögliche *Lösung* des Problems R für die Angabe x".)

Beispiele:

Eine mögliche Charakterisierung des Problems „Wurzelziehen" wäre z.B.

$X_W := Y_W :=$ Menge der reellen Zahlen;
$R_W := \{(x,y) \in X_W \times Y_W \mid y^2 = x\}$

– 3 wäre eine mögliche Lösung des Problems R_W für die Eingabe 9;
zur Angabe – 1 hat das hier charakterisierte Problem keine Lösung.

Ein spezielles Dokumentationsproblem in einer Bibliothek könnte z.B. so charakterisiert werden:

X_D := eine gewisse Menge von Stichwörtern;
Y_D := die Menge der Titel der in der betreffenden Bibliothek vorhandenen Bücher;
R_D := $\{(x,y) \in X_D \times Y_D \mid$ der Buchtitel y enthält das Stichwort $x\}$;

Der Buchtitel „H. MAIR, PROGRAMMIEREN FÜR ANFÄNGER" wäre z.B. eine Lösung des Problems R_D für die Angabe „PROGRAMMIEREN", falls das entsprechende Buch in der Bibliothek vorhanden ist.

Praktische Probleme der Datenverarbeitung sind oft sehr umständlich zu beschreiben. Immer haben sie aber die hier angegebene Struktur.

Was ist ein Computer?

Ein *Computer* ist charakterisiert durch

eine Menge Z von *„Zuständen"*,
eine *„Überführungsfunktion"* $\delta : Z \to Z$.
(Für „$\delta(z) = z'$" lies: „z' ist der Nachfolger des Zustandes z in dem durch δ charakterisierten Computer".)

Ein Computer „funktioniert" so, daß er, ausgehend von einem beliebigen Zustand $z \in Z$, selbständig nacheinander die Zustände

$$z, \delta(z), \delta(\delta(z)), \ldots .$$

annimmt.

Beispiel:

Ein Zustand eines realen Computers ist charakterisiert durch den Inhalt sämtlicher Speichersätze (einschließlich spezieller Register).
Die Überführungsfunktion eines realen Computers ist fix gegeben durch seine „Hardware", die für jeden Zustand festlegt, welcher Zustand im nächsten Takt angenommen wird.

Die Überführungsfunktionen δ realer Computer sind sehr komplex. Ihre (oft unvollständige) Beschreibung füllt dicke Manuals. Alle Computer funktionieren aber nach dem hier angegebenen Grundprinzip (bzw. setzen sich größere Computersysteme aus einzelnen Moduln zusammen, die nach diesem Prinzip funktionieren).

Wie löst man ein Problem mit einem Computer?

Die heutigen Computer werden für die Lösung eines gegebenen Problems *„programmiert"*. Der Programmierer schreibt ein „Programm", das ist ein nach genauen Regeln zusammengestelltes Textstück, das zusammen mit jeder beliebigen Angabe für das Problem einen Anfangszustand des Computers und damit auch einen nachfolgenden „Ablauf", d.h. eine Aufeinanderfolge

von Zuständen im Computer, festlegt. Dieser Ablauf soll zu einem von außen erkennbaren Endzustand führen, der die Lösung des Problems für die gewählte Angabe enthält.

Genauer:

Gegeben ein Computer: Z, δ.
Ein *Programmiersystem* für diesen Computer ist charakterisiert durch

eine Menge P von „*Programmen*",
eine Menge D von „*Daten*",
eine Menge $E \subset Z$ von Zuständen, die als *Endzustände* aufgefaßt werden,
eine „*Initialisierungsvorschrift*" $\gamma : P \times D \to Z$
(für „$\gamma(p, d) = z$" lies: „z ist der durch das Programm p und die Daten d definierte Anfangszustand des Computers").

Mit diesen Bestimmungsstücken ist gewissen Programmen p und Datensätzen d auf natürliche Weise ein Endzustand zugeordnet, nämlich der erste Endzustand, der in der Zustandsfolge

(1)
$$\begin{aligned} z_0 &:= \gamma(p, d) \\ z_1 &:= \delta(z_0) \\ z_2 &:= \delta(z_1) \\ &\vdots \\ z_{t+1} &:= \delta(z_t) \\ &\vdots \end{aligned}$$

vorkommt. Man beachte, daß nicht jede solche Folge zu einem Endzustand führen muß.

Beispiel:

Für einen realen Computer und ein reales Programmiersystem wie z.B. ALGOL könnte $\gamma(p, d)$ z.B. als der Zustand des Computers betrachtet werden, der (im wesentlichen) durch ein in den Lochkartenleser eingelegtes Lochkartenpaket mit dem abgelochten Programm p und den abgelochten Daten d charakterisiert ist. E könnte z.B. die Menge jener Zustände sein, bei denen am Zeilendrucker eine Meldung „END OF ALGOL RUN" o.ä. erscheint. (Man kann das hier gegebene Begriffsschema in diesem Falle aber auch so verwenden, daß man z.B. in den Initialisierungsvorgang den gesam-

ten Compilationsvorgang hineinsteckt, so daß $\gamma(p, d)$ im wesentlichen durch den internen Speicherzustand bestimmt ist, der das übersetzte Programm enthält.)

Um nun ein gegebenes Problem $X, Y, R \subset X \times Y$ auf dem gegebenen Computer durch ein Programm eines zugehörigen Programmiersystems lösen zu können, müssen wir noch die möglichen Problemangaben als Daten vercodieren und die erreichbaren Endzustände als Resultate interpretieren. Das geschieht durch zwei Konventionen:

$f : X \to D$ *(„Eingabevercodierung"),*
$g : E \to Y$ *(„Resultatentcodierung").*

Dann können wir den oben erwähnten Begriff des „Problemlösens durch Programmierung eines Computers" wie folgt präzisieren:

Zunächst sei

(2) $\delta^*(z, 0) := z$
$\delta^*(z, t+1) := \delta^*(\delta(z), t)$

($\delta^*(z, t)$ gibt also jenen Zustand, den der Computer vom Zustand z aus in t Schritten erreicht),

(3) $\mathrm{End}(z, t) :\leftrightarrow \delta^*(z, t) \in E \wedge (\forall \tau < t)\ (\delta^*(z, \tau) \notin E)$

(d.h., End(z, t) ist die Behauptung, daß der Computer nach genau t Schritten einen Endzustand erreicht)

und

(4) $\mathrm{Def}(R) := \{ x \mid (\exists y)(x, y) \in R \}$

(d.h., Def(R), der *„Definitionsbereich* von R", ist nach unserer Interpretation die Menge aller Angaben, für welche das Problem R eine Lösung hat).

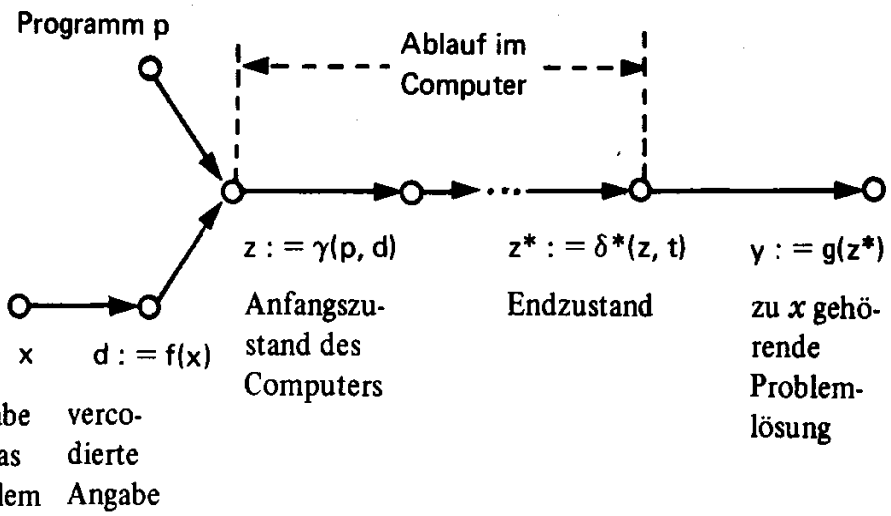
Dann heißt ein $p \in P$ *korrektes Programm zur Lösung des gegebenen Problems* R auf dem gegebenen Computer (relativ zu dem betrachteten Programmiersystem und den verwendeten Ein-, Ausgabevercodierungen) $:\leftrightarrow$

(5) $(\forall x \in \mathrm{Def}(R))\ (\exists t)\ (\mathrm{End}(\gamma(p, f(x)), t) \wedge (x, g\delta^*(\gamma(p, f(x)), t)) \in R)$

(Lies: „Für alle Angaben x, für welche das Problem R überhaupt eine Lösung hat, gibt es eine Schrittanzahl t, nach welcher der Computer

bei der Ausführung des Programms p für die Daten $f(x)$ in einen Endzustand gerät, der eine Lösung des Problems R für die Angabe x enthält.")

oder graphisch:



Die Aussage (5) wollen wir im folgenden die *Verifikationsaussage bezüglich R und p nennen.*

Beispiel:

Wir betrachten wieder das Programmiersystem des letzten Beispiels mit demselben Computer.
Sei p das ALGOL-Programm

```
begin
integer a, b;
read (a, b);
print (a + b)
end;
```

Mit einer Eingabevercodierung f, die jedem Paar ganzer Zahlen die übliche Dezimaldarstellung (mit Vorzeichen und genügend vielen Zwischenräumen zwischen den Zahlen) auf Papier zuordnet und der Ausgabefunktion g, daß wir die auf dem Zeilendrucker vor der Meldung „END OF ALGOL RUN" ausgedruckte Symbolfolge, als Dezimaldarstellung einer ganzen Zahl aufgefaßt, als die in einem Endzustand enthaltene Ausgabeinformation betrachten, sollte dieses Programm ein Programm zur Lösung des Problems $\{((x_1, x_2), y) \in (G \times G) \times G \mid x_1 + x_2 = y\}$ sein (also des „Problems der Addition auf der Menge G der ganzen Zahlen").

2. Das Problem der Programmverifikation

Das Problem zu entscheiden, ob bei gegebenem Problem R und vorgeschlagenem Programm p (für einen Computer mit Programmiersystem) die Verifikationsaussage (5) bezüglich R und p richtig ist, nennt man das *Problem der Programmverifikation.*

Das Problem der Programmverifikation ist eines der wichtigsten praktischen Probleme der Datenverarbeitung. Denn ein Programm, von dem man nicht mit einiger Sicherheit sagen kann, daß die Verifikationsaussage erfüllt ist, daß es also „immer richtige", besser „immer die vom Problem her gewünschten" Resultate liefern wird, ist praktisch wertlos.

In der heute üblichen Praxis gewinnt der Computerbenutzer die Überzeugung, daß die Verifikationsaussage erfüllt ist, auf einem zweifachen Weg:

a) Er *„überdenkt die Logik seines Programmes"* immer wieder, d.h., er überlegt sich immer wieder, ob die durch die Instruktionen seines Programms im Computer angestoßenen Abläufe geeignet sind, eine Problemlösung in Abhängigkeit von der Eingabe schrittweise zu konstruieren. Dazu ist notwendig:
   aa) Eine klare Vorstellung, welche Abläufe überhaupt zur Konstruktion von Problemlösungen geeignet wären,
   ab) die genaue Kenntnis, wie die notwendigen Abläufe im Rahmen des gewählten Programmiersystems mit einem Programm im Computer ausgelöst werden können.

   Die Forderung aa) ist zum Unterschied von der Forderung ab) vom gewählten Computer und Programmiersystem weitgehend unabhängig.

b) Er *„testet"* das von ihm konstruierte Programm p an zahlreichen Eingaben $x_1, x_2, \ldots, x_k \in \mathrm{Def}(R)$, d.h., er überprüft die Richtigkeit der in der Verifikationsaussage für alle $x \in \mathrm{Def}(R)$ geforderten Bedingung für die ausgewählten $x_1, \ldots, x_k$, indem er das Programm auf dem Computer für die Daten $f(x_1), \ldots, f(x_k)$ „rechnen läßt" und die erhaltenen Resultate $y_1, \ldots, y_k$ daraufhin untersucht, ob sie Lösungen des gegebenen Problems für die Angaben $x_1, \ldots, x_k$ sind.

Obwohl die Forderung ab) durch verbesserte Programmiersysteme (bei denen die Programme als sprachliche Objekte die angestoßenen Abläufe immer besser mit den aus der Umgangssprache gewohnten Mitteln charakterisieren) schon seit einiger Zeit nicht mehr in dem Maße ins Gewicht fällt und andererseits die in b) beschriebene Technik des Testens gegenüber der Frühzeit der Datenverarbeitung methodisch bereits stark verbessert ist, sind die in a) und b) skizzierten üblichen Wege der „Programmverifikation"

für die Praxis in keiner Weise ausreichend. Einerseits täuscht sich der Mensch auch nach oftmaligem Überdenken über die Wirkung geplanter Abläufe, und der aus den Tests in b) nahegelegte „Schluß" von der Richtigkeit einiger Testläufe auf die Richtigkeit der Verifikationsaussage ist nicht nur mathematisch, sondern – wie die Erfahrung zeigt – auch faktisch unzulässig.

Das Problem der Programmverifikation ist also nach wie vor eines der zentralen praktischen Probleme in der Datenverarbeitung. Eine Hauptstoßrichtung in dem Bestreben, hier nicht nur eine graduelle Verbesserung, sondern einen prinzipiellen Fortschritt zu erzielen, geht dahin, bei gegebenem R und p die *Richtigkeit der Verifikationsaussage bezüglich R und p mit mathematischen Hilfsmitteln zu beweisen.*

In diesem Bestreben gibt es natürlich innerhalb der Mathematik bereits eine Jahrhunderte alte Tradition. Denn jedes neue Verfahren zur Lösung eines Problems in der Mathematik bedarf eines Beweises zu seiner Verifikation. So gesehen besteht zwischen dem Problem der Programmverifikation und dem Problem der Beweispflicht für ein mathematisches Verfahren nur der graduelle Unterschied, der sich daraus ergibt, daß mathematische „Verfahren" meist als „Programm" für menschliche Rechner formuliert werden (es sei dem Leser als Übung überlassen, die Begriffsbildungen des ersten Abschnitts am Beispiel eines menschlichen „Computers" durchzuexerzieren, um festzustellen, daß sich auch dieser „Fall" hier organisch einfügt).

Bei der Verfolgung des Ziels, Verifikationsaussagen mathematisch zu beweisen, muß man sich natürlich darüber im klaren sein, daß auch das Beweisen als menschliche Aktivität der allgemeinen Fehleranfälligkeit des Menschen unterworfen ist. Dementsprechend ist auch die Meinung darüber, was man als mathematischen Beweis akzeptiert, in der mathematischen Praxis gar nicht hundertprozentig einheitlich. Insbesondere werden Beweise normalerweise als „inhaltliche" Beweise geführt, wo es durchaus vom Vorstellungsvermögen der beteiligten Mathematiker abhängt, in wie großen Schritten gedacht wird und was als unbezweifeltes Basiswissen anerkannt wird. Dies gilt insbesondere bei allen mathematischen Formulierungen über „Abläufe" (zum Unterschied von Behauptungen über statische Gegebenheiten!). In diesem Zusammenhang kann auch der oben erwähnte Weg, die Gültigkeit einer Verifikationsaussage durch wiederholtes „Überdenken der Logik des Programms" zu rechtfertigen, als ein Versuch eines Beweises angesprochen werden (vgl. auch den vierten Abschnitt.) So gesehen würde also zwischen dem Beweisen von Verifikationsaussagen und dem „Überdenken der Logik von Programmen" wieder nur ein gradueller Unterschied bestehen.

Der Unterschied kann (vorbehaltlich einer weiteren Analyse im dritten Abschnitt) zu einem prinzipiellen gemacht werden, wenn man extreme Ansprüche an den Beweisbegriff stellt, nämlich in konsequenter Anwendung des Grundkonzepts der Datenverarbeitung als Beweis nur zuläßt, was selbst automatisch, d.h. also wieder durch einen Computer und damit (hoffentlich!) fehlerfrei durchgeführt werden kann.

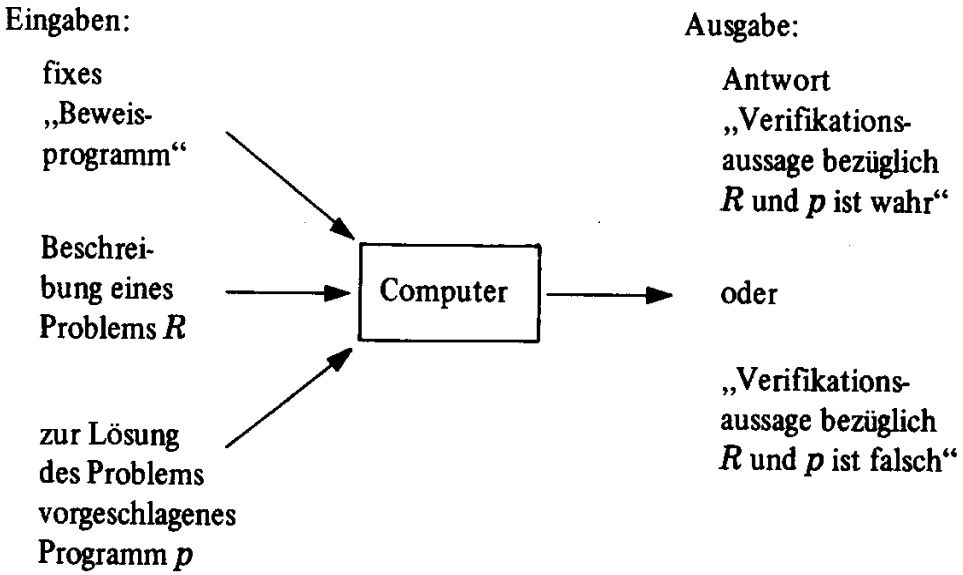
Das Idealziel, das im Zusammenhang mit der Programmverifikation aufgestellt werden kann und heute auch als Forschungsanliegen intensiv bearbeitet wird, lautet also: *der automatische Beweis von Programmverifikationsaussagen mit Hilfe von Computern.*

Dieser Version des Problems der Programmverifikation ist der nächste Abschnitt gewidmet. Im letzten Abschnitt werden wir noch eine praktische Methode angeben, wie die üblichen Verifikations-„Überlegungen" zum Überdenken der Logik von Programmen in der Praxis etwas systematisiert werden können, so daß sie zusammen einen Beweis der Programmverifikationsaussage ergeben.

3. Automatische Programmverifikation

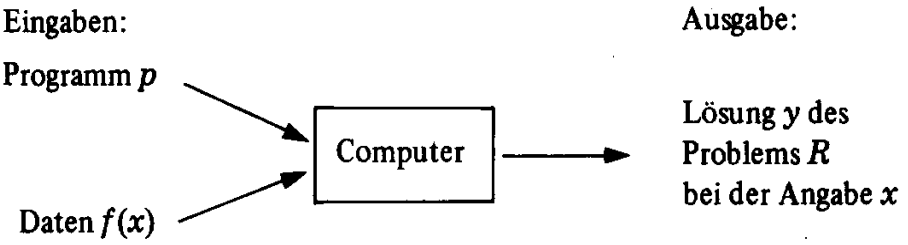
Als Ideal eines Computereinsatzes mit vorhergehender Programmverifikation hat man also in letzter Konsequenz ein Vorgehen der folgenden Art vor Augen:

1. Schritt:



2. Schritt:

falls die Antwort beim ersten Schritt lautet:
„Verifikationsaussage ist falsch“:
Änderung des Programms p;
falls die Antwort beim ersten Schritt lautet:
„Verifikationsaussage ist wahr“:



Für einen derartigen Computereinsatz ergeben sich vier Probleme:

Ein „philosophisches“ Problem:

Im Falle, daß ein „Beweisprogramm“, wie es hier benötigt würde, überhaupt geschrieben werden kann: Wer garantiert, daß es selbst „richtig“ ist?

Die Antwort ist natürlich: „Niemand“. Daher muß die im letzten Abschnitt abgebrochene Analyse dahingehend fortgesetzt werden, daß es im letzten natürlich immer nur einen „graduellen“ Fortschritt bei der Verbesserung der Überprüfung von Verifikationsaussagen gibt. Allerdings mag einem hier der „Werkzeuggedanke“ helfen und das hier skizzierte Anliegen zunächst praktisch rechtfertigen: der gesamte (technologische) Fortschritt des Menschen beruht im letzten darauf, daß es der Mensch verstanden hat, mit schlechten Werkzeugen (hier einem nicht garantierbar richtigen Beweisprogramm) verbesserte Werkzeuge (hier möglichst fehlerfreie Problemprogramme) zu schaffen. Auch sollte natürlich nicht übersehen werden, daß ein vielleicht richtiges Beweisprogramm das Verifikationsproblem für sämtliche anderen Programme lösen würde.

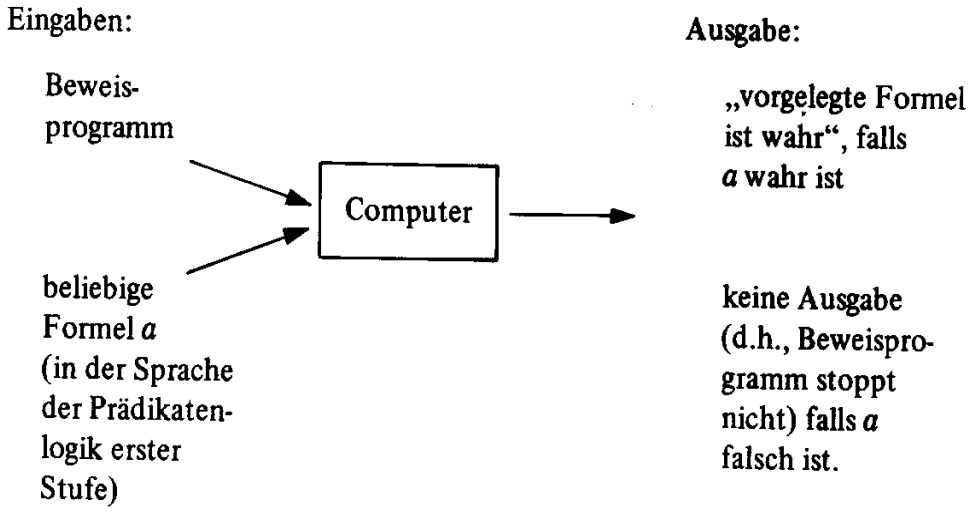
Ein theoretisches Problem der mathematischen Logik:

Kann der Vorgang des mathematischen Beweisens prinzipiell automatisiert werden?

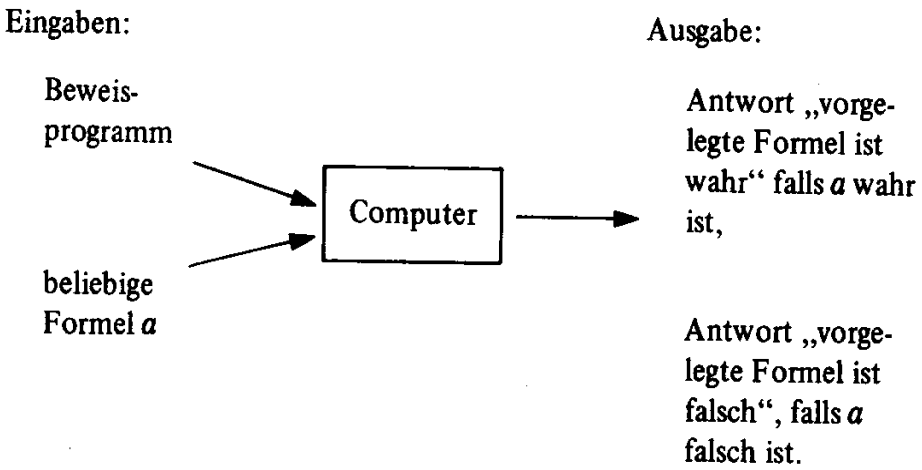
Die Antwort auf diese Frage ist in einem eingeschränkten Sinn positiv:

Für jenen Teil der Mathematik, der in der „Sprache der Prädikatenlogik erster Stufe“ (das ist im Prinzip die mathematische Umgangssprache, syntaktisch gereinigt und mit der Einschränkung, daß die Quantoren „es gibt“ und „für alle“ nur auf Individuenvariable und nicht auf Funktionen- oder

Prädikatensymbole angewendet werden dürfen) formuliert werden kann, kann ein Beweisprogramm (ein sogenannter Theorem prover) geschrieben werden, der folgendes leistet:



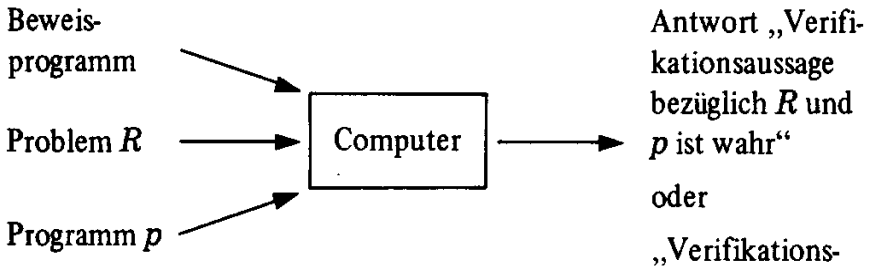
Diese Tatsache ist im wesentlichen der Inhalt der bekannten Vollständigkeitssätze für die verschiedenen Formulierungen der Prädikatenlogik erster Stufe (siehe z.B. Shoenfield 67). Die Sprache der Prädikatenlogik erster Stufe ist im Prinzip für die Formulierung der bei der Programmierung von Problemen benutzten Mathematik leicht ausreichend, allerdings im Gebrauch manchmal unhandlich. Bedeutsamer ist die Einschränkung, die sich aus einem anderen bekannten Resultat der mathematischen Logik (dem sogenannten Unentscheidbarkeitssatz, siehe ebenfalls Shoenfield 67) ergibt, das zeigt, daß kein Beweisprogramm für die Prädikatenlogik erster Stufe existieren kann, das folgendes leisten könnte:



Allerdings ist es immerhin noch möglich, daß ein verbessertes Beweisprogramm auch einige falsche Formeln nach endlich vielen Schritten erkennt, nie aber alle wahren und alle falschen Formeln. Das oben skizzierte Ideal einer automatischen Programmverifikation muß also auf jeden Fall zunächst reduziert werden auf das Konzept:

1. Schritt:

Eingabe: Ausgabe:



Antwort „Verifikationsaussage bezüglich R und p ist wahr"

oder

„Verifikationsaussage bezüglich R und p ist falsch"

oder

keine Antwort

2. Schritt:

falls Antwort „Verifikationsaussage falsch" oder „Verifikationsaussage wahr":
Vorgangsweise wie früher,

falls keine Antwort: man kann weder die Richtigkeit noch die Falschheit der Verifikationsaussage behaupten.

Man darf auch nicht glauben, daß die oben erwähnte Nicht-Existenz eines Beweisprogrammes, das auch falsche Formeln immer in endlich vielen Schritten erkennt, vielleicht für die Formeln, die im Zusammenhang mit der automatischen Programmverifikation bewiesen werden müßten, nicht relevant sei. Ganz im Gegenteil: die Beweise des Unentscheidbarkeitssatzes – auch die ersten Beweise vor vierzig Jahren, also noch einige Zeit vor der Erfindung des Computers (!) – benutzen gerade die Tatsache, daß sich gewisse „Programmierprobleme" in der betrachteten Sprache formulieren lassen.

Ein praktisches Problem der Programmierung:

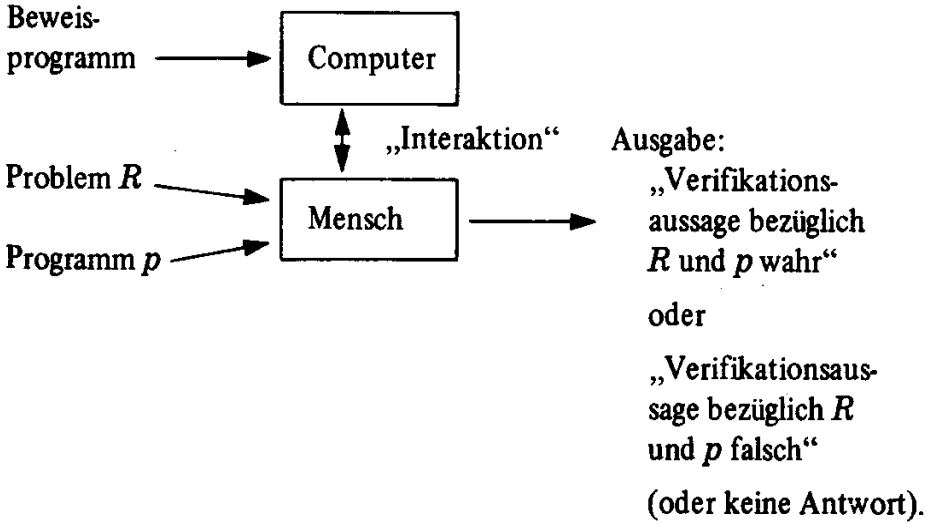
Kann man mit den heute zur Verfügung stehenden technologischen Mitteln Beweisprogramme schreiben, deren Zeit- und Speicheraufwand so gering ist, daß sie bei praktisch interessanten Problemen eingesetzt werden können?

Die ersten Beweisprogramme wurden vor etwas mehr als zehn Jahren geschrieben. Obwohl ihre Effizienz auch durch neue theoretische Erkenntnisse inzwischen stark verbessert wurde, können auch heute erst einfachste mathematische Formeln in vernünftiger Zeit bewiesen werden. Insbesondere liegen die komplizierten Verifikationsaussagen, auf die es bei der automatischen Programmverifikation ankommt, weit außerhalb dessen, was man vernünftigerweise von den heutigen (und zukünftigen) Beweisprogrammen erwarten kann. Zur Information über den heutigen Stand auf dem Gebiet der Beweisprogramme siehe das Lehrbuch Chang/Lee 73 und den Übersichtsartikel Bibel 76 in diesem Band.

Diese Erfahrungen haben aber nicht zu der Konsequenz geführt, daß man das Anliegen der automatischen Programmverifikation als unrealistisch aufgegeben hätte. Vielmehr wird es modifiziert in Richtung *„halbautomatische Programmverifikation"* nach dem Schema:

1. Schritt:

Eingabe:



2. Schritt: Wie oben.

Das heißt: Es wird an der Konstruktion von Systemen gearbeitet, wo der Mensch noch die Steuerung des Beweisprozesses für die Verifikationsaussagen inne hat, den Beweisvorgang z.B. in sinnvolle Teile zerlegt und nur versucht, möglichst viel Routinearbeit innerhalb des Beweisvorganges an einen (mit Beweisprogramm versehenen) Computer abzugeben. Mit derartigen interaktiven Programmverifikationssystemen wird derzeit in zahlreichen Forschungslabors auf der ganzen Welt experimentiert. Zusammenfassend kann man sagen, daß die dabei erzielten Erfolge noch nicht von der Art sind, daß eine kommerzielle Ausnutzung in Software-Paketen (mit dem Grad an Selbstverständlichkeit, mit dem heute Compiler oder Betriebssysteme angeboten werden) in den nächsten Jahren zu erwarten ist. Die Klasse der Programme, die auf (halb-)automatischem Wege verifiziert werden kann, ist im wesentlichen die Klasse der Übungsprogramme, die in einem Programmierkurs für Anfänger in den ersten Stunden geschrieben werden. Zudem werden praktisch nur Programmiersprachen betrachtet, die gegenüber den praktisch verwendeten Sprachen stark vereinfacht und gereinigt sind. Dennoch ist hier für die nächste Zeit eine stürmische Entwicklung zu erwarten, die den Programmierstil und die Programmierpraxis entscheidend verändern wird. Über den derzeitigen Stand auf dem Gebiet der halbautomatischen Programmverifikation gibt Proceedings 75 ein gutes Bild. Dort sind auch zahlreiche Literaturangaben zu finden.

Ein Problem der Konzeptklärung:

Welche Aussage muß durch ein Beweisprogramm bewiesen werden, um die Korrektheit eines Programmes p für ein Problem R zu garantieren?

Nach der Analyse im ersten Abschnitt ist die Antwort natürlich klar; es ist genau die Verifikationsaussage (5), die allerdings, um als Eingabe für ein Beweisprogramm geeignet zu sein, als Aussage in der vom Beweisprogramm bearbeiteten Sprache, also normalerweise in der Sprache der Prädikatenlogik erster Stufe, formalisiert sein muß. Dieser Formalisierungsprozeß wäre in jedem konkreten Fall äußerst mühselig, da ja in die Aussage (5) abgesehen von der Problembeschreibung für R vor allem auch eine Beschreibung der komplizierten Überführungsfunktion δ eingehen müßte. Dieser Weg ist praktisch ungangbar. Vielmehr sollten wir hier auf die Unterscheidung zurückkommen, die wir bei der Methode des „Überdenkens" der Programmlogik im zweiten Abschnitt getroffen haben: In die Programmierung geht sowohl ein Wissen um (weitgehend „maschinenunabhängige") Abläufe, die zu einer Problemlösung führen könnten, als auch das Wissen um die Realisierung genau dieser Abläufe auf dem gegebenen Computer ein.

Dementsprechend wollen wir auch das mehr technische (aber wichtige) Problem, wie man kontrolliert, ob im verwendeten Computer auch die „gemeinten" Abläufe erzeugt werden, trennen vom „eigentlichen" Problem der Programmverifikation, als welches wir ab nun mehr den maschinenunabhängigen Teil des in (5) präzisierten Gesamtproblems betrachten wollen. Damit entsteht aber das Problem, wie die Aussage der Korrektheit eines Programms p für ein gegebenes Problem R neu und maschinenunabhängig formalisiert werden kann! Dazu muß vor allem geklärt werden, wie „die Wirkung eines Programms auf Daten" unabhängig von konkreten Computern erklärt werden kann.

Dieses Problem der maschinenunabhängigen „Bedeutung" („Semantik") von Programmen wird seit den frühen Sechziger-Jahren intensiv bearbeitet. Viele dabei verfolgte Konzepte bauen in expliziter oder versteckter Form auf dem Konzept der „operationalen Semantik" auf, dessen Grundgedanke nach den Darlegungen im ersten Abschnitt sofort verstanden werden kann, weil wir dazu nur die bereits eingeführten Begriffsbildungen des ersten Abschnitts neu interpretieren und lesen müssen:

Anstatt daß wir wie im ersten Abschnitt die Exekution von Programmen einer zu untersuchenden „Programmiersprache" auf einem realen Computer vor Augen haben, betrachten wir einen abstrakten Computer, der wieder durch eine Zustandsmenge Z und eine Überführungsfunktion δ charakterisiert ist und in der Lage sein soll, beliebige Programme der untersuchten Sprache für beliebige Daten zu exekutieren. Bei der Definition von δ braucht man jetzt aber auf keine Fragen der technischen Realisierbarkeit Rücksicht zu nehmen, sondern vielmehr nur darauf, daß die Funktionsweise des abstrakten Computers möglichst klar und übersichtlich die durch die Programme der betrachteten Sprache hervorzurufenden Abläufe charakterisiert. Durch Angabe eines solchen abstrakten Computers wird die Semantik der untersuchten Programmiersprache definiert, siehe z.B. Lucas/Lauer/Stigleitner 68. Theoretische Aspekte der Methode werden in Buchberger 74 und Buchberger/Roider 75 behandelt.

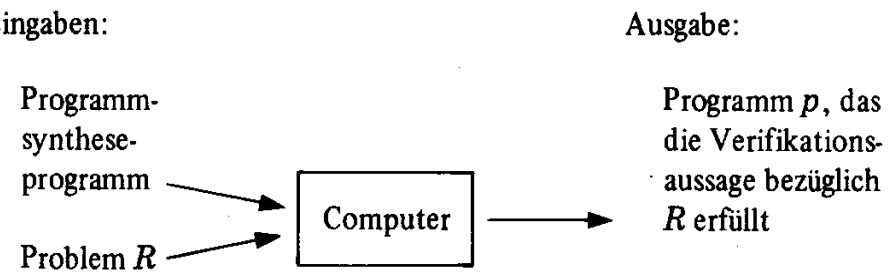
Ein zweites Grundkonzept, auf dem die Definition der Semantik von Programmiersprachen und Verifikationsmethoden (vor allem für rekursive Programme) aufgebaut werden können, ist die sogenannte Fixpunkt-Theorie der Programmiersprachen, die in den letzten Jahren immer mehr an Bedeutung gewinnt, jedoch in dieser Einführung nicht dargestellt werden kann (siehe dazu das Lehrbuch Manna 74, Kapitel 3).

Zum Abschluß dieses Abschnitts sei noch bemerkt, daß es heute auch Versuche gibt, das Problem der Programmverifikation überhaupt zu vermeiden. Dazu wird mit zwei Grundgedanken operiert:

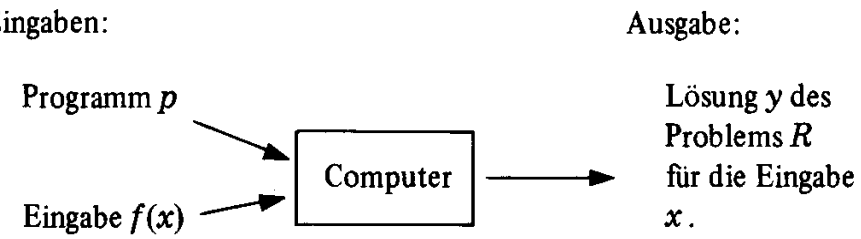
Der Gedanke der automatischen Programmsynthese:

Das Problem der Programmverifikation wäre aus der Welt geschafft, wenn es gelänge, ein (korrektes!) „Programmsyntheseprogramm" zu schaffen, das einen Computereinsatz der folgenden Art ermöglichen würde:

1. Schritt:



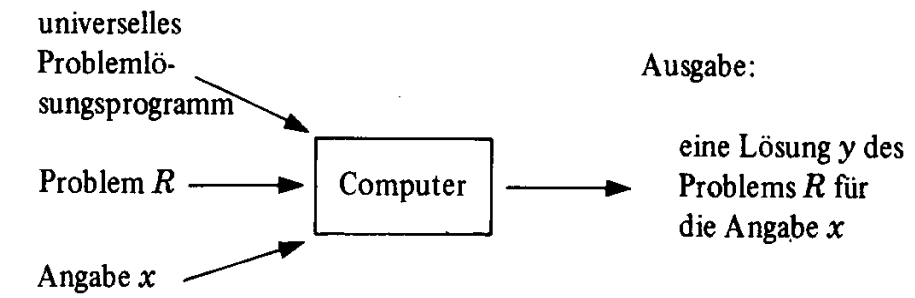
2. Schritt:



So utopisch dieses Vorhaben auf den ersten Blick erscheint, gibt es sowohl eine theoretische Rechtfertigung (siehe Constable 74) als auch durchaus vielversprechende praktische Versuche (siehe Manna/Waldinger 71), die ein solches Vorhaben zumindest gleich realistisch erscheinen lassen wie das Ziel einer (halb-)automatischen Programmverifikation. Die tragende Rolle in einem derartigen Programmsyntheseprogramm kommt wieder einem Beweisprogramm zu, das bei gegebener Problembeschreibung für R eine Formalisierung der Aussage „$(\forall x)(\exists y)((x, y) \in R)$" zu beweisen versucht. Aus einem gelungenen Beweis b für diese Aussage wird dann das gesuchte Programm p automatisch konstruiert.

Der Gedanke des Rechnens ohne Programm:

Das Problem der Programmverifikation wäre auch vermieden, wenn es gelänge, ein (korrektes!) „universelles Problemlösungsprogramm" zu schaffen, das einen Computereinsatz der folgenden Art ermöglichen würde:



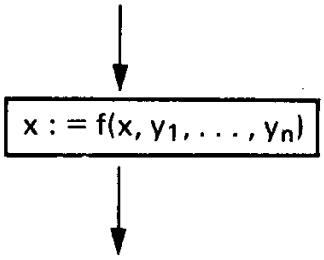
Obwohl dieser Gedanke zunächst noch utopischer erscheint als der vorhin erwähnte, erweist er sich eher als leichter realisierbar. Auch dazu gibt es schon konkrete Implementierungsversuche (siehe dazu Bibel 75). Die zentrale Rolle in derartigen Systemen kommt wieder einem Beweisprogramm zu, das bei gegebenem R und x eine Formalisierung der Formel „$(\exists y)((x, y) \in R)$" zu beweisen trachtet und aus einem gelungenen Beweis die Lösung y konstruiert.

Im Rahmen dieser einführenden Darstellung können wir auch diese beiden Gedanken nicht weiter verfolgen.

4. Eine praktische Methode zur Kultivierung von Programmverifikationsüberlegungen

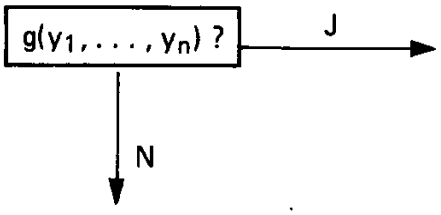
Wir wollen hier noch eine von Floyd (siehe Floyd 67) in die Literatur eingeführte Methode zur Programmverifikation erläutern, die einem helfen kann, die mehr oder weniger genau geführten üblichen Überlegungen zum „Überdenken der Logik eines Programms" (vgl. zweiter Abschnitt!) in systematischerer Weise abzuwickeln. Die Methode läßt sich auch (z.B. aufbauend auf dem Konzept der operationalen Semantik des letzten Abschnitts) formal rechtfertigen. Sie dient auch in vielen interaktiven halbautomatischen Programmverifikationssystemen als Basismethode.

Wir betrachten Flußdiagramme, das sind Programme, die im wesentlichen aus Instruktionen der Art



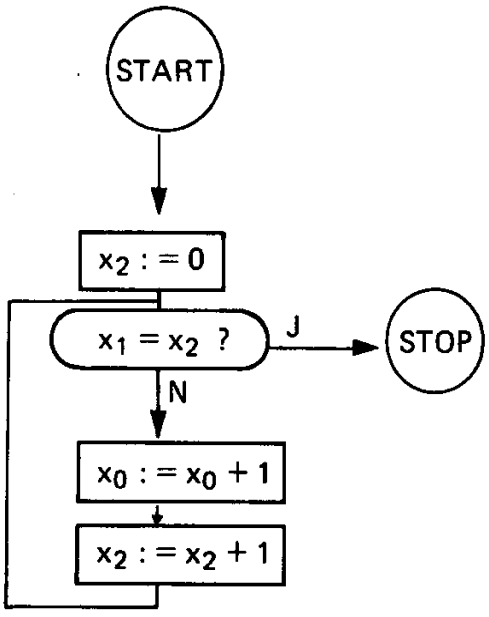
(„Zuweisungen",
f . . . Funktionskonstante)

und



(„Verzweigungen",
g . . . Prädikatenkonstante)

mit der üblichen Bedeutung zusammengesetzt sind. Wir verzichten in diesem Abschnitt auf jede formale Definition. Ein Flußdiagramm, das der Variablen x_0 den Wert $x_0 + x_1$ zuordnet (wenn man als einzige Grundoperation nur die Nachfolgeroperation zuläßt), würde z.B. wie folgt ausschauen:



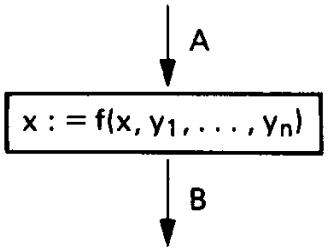
Der Grundgedanke für das Verfahren zur Programmverifikation nach Floyd ist naheliegend wie folgt:

a) Wir schreiben an den Eingang und die möglichen Ausgänge einer jeden Instruktion des gegebenen Flußdiagramms p Aussagen über die Werte der im Flußdiagramm vorkommenden Variablen (vor die Stop-Instruktionen schreiben wir dabei natürlich jene Aussagen, die für die endgültigen Werte der Variablen unserem Wunsche nach gelten sollen, also im wesentlichen eine Beschreibung des gegebenen Problems R!).

b) Wir überprüfen, ob die Aussage bei jedem Ausgang einer jeden Instruktion im Flußdiagramm aus der Aussage beim Eingang der Instruktion und der Information, die wir über die Wirkung der Instruktion haben, folgt. Wenn das der Fall ist, können wir sicher sein, daß, falls man bei Abarbeitung des Flußdiagramms für irgendwelche konkreten Daten d überhaupt einen Stop erreicht, schließlich für die erreichten Endwerte der Variablen die beim jeweiligen Stop stehende Aussage gelten muß (noch nicht gesichert ist damit allerdings, ob für beliebige Daten auch immer ein Stop erreicht wird!).

Genauer müssen wir bei b) folgende Kontrollen durchführen:

(K1) Bei Zuweisungsinstruktionen



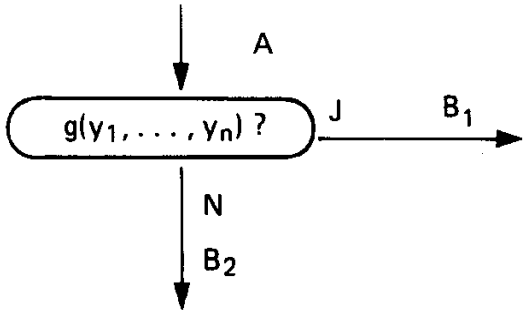
mit Aussagen A bzw. B vor bzw. nach der Instruktion ist zu kontrollieren:

$$x = f(\xi, y_1, \ldots, y_n) \wedge A_\xi^x \rightarrow B$$

(wobei A_ξ^x jene Aussage bezeichnet, die man aus A erhält, indem man die Variable x überall durch ξ ersetzt).

(Lies: „Wenn die Aussage A für den alten Wert ξ von x gilt, dann muß nach Ausführung der Instruktion die Aussage B für den neuen Wert von x gelten.")

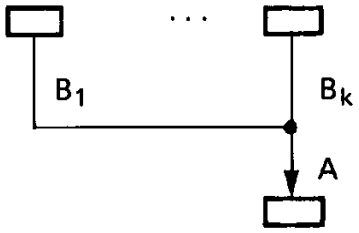
(K2) Bei Verzweigungsinstruktionen



mit Aussagen A, B_1 bzw. B_2 vor der Instruktion bzw. am Ja- bzw. am Nein-Ausgang ist zu kontrollieren:

$$A \wedge g \rightarrow B_1,$$
$$A \wedge \neg g \rightarrow B_2.$$

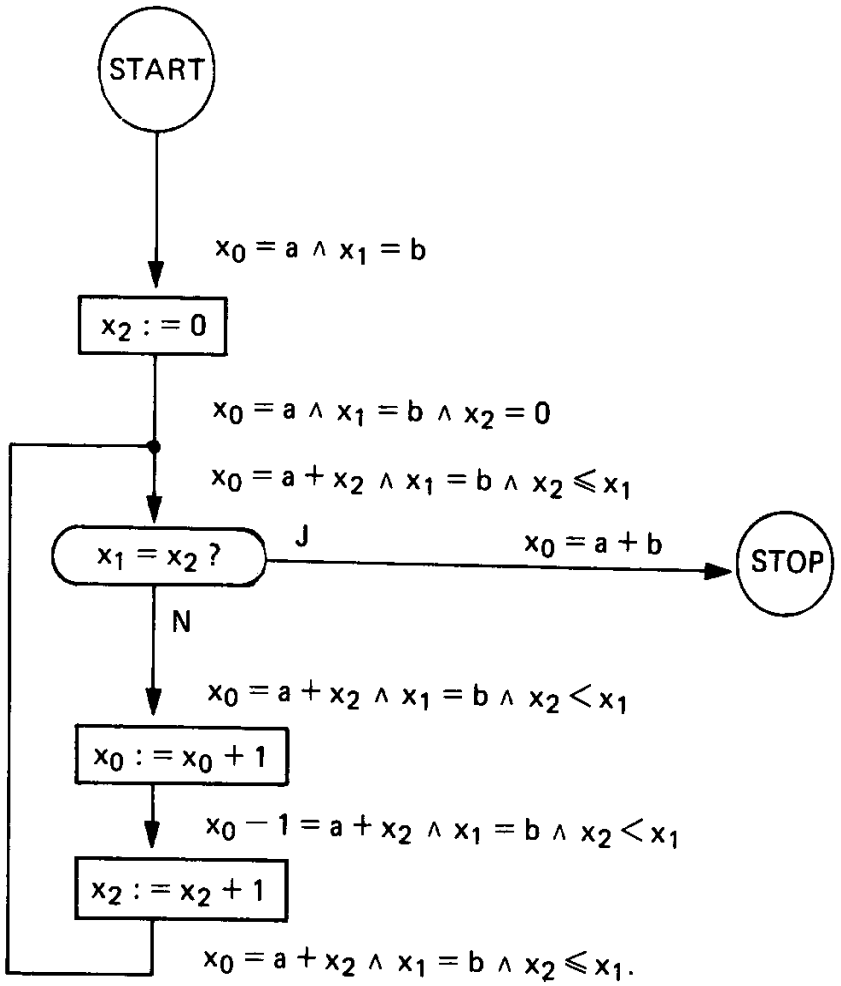
(K3) ist eine Instruktion Sprungziel mehrerer Instruktionen



mit der Aussage A vor dem Sprungziel und den Aussagen $B_1, \ldots, B_k$ nach den betreffenden Instruktionen, so ist zu kontrollieren:

$$B_1 \rightarrow A,$$
$$B_2 \rightarrow A,$$
$$\vdots$$
$$B_k \rightarrow A.$$

In unserem Additionsbeispiel würde die Anwendung der Methode folgendes Bild liefern (alle vorkommenden Größen werden aus N angenommen!):



Wir müssen gemäß (K1) z.B. für die erste Instruktion überprüfen, daß:

$$x_2 = 0 \wedge x_0 = a \wedge x_1 = b \rightarrow x_0 = a \wedge x_1 = b \wedge x_2 = 0;$$

das ist natürlich richtig.

Ebenso gemäß (K2) für die zweite Instruktion:

$$x_0 = a + x_2 \wedge x_1 = b \wedge x_2 \leqslant x_1 \wedge x_1 = x_2 \rightarrow x_0 = a + b;$$

auch das ist richtig.

Genauso leicht sind die anderen Checks. Wir betrachten noch die dritte Instruktion, wo wir kontrollieren müssen, daß:

$$x_0 = \xi + 1 \wedge \xi = a + x_2 \wedge x_1 = b \wedge x_2 < x_1$$
$$\rightarrow x_0 - 1 = a + x_2 \wedge x_1 = b \wedge x_2 < x_1;$$

auch das ergibt sich leicht durch Ausrechnen von $\xi = x_0 - 1$ aus der ersten Gleichung.

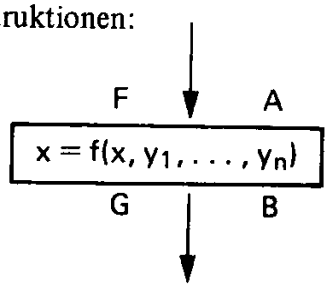
Nach Durchführung aller dieser Checks können wir behaupten: wenn die Werte der Variablen x_0 und x_1 am Anfang a bzw. b waren und man bei Abarbeitung des Flußdiagramms jemals zur Stop-Instruktion gelangt, so gilt am Ende, daß der Wert der Variablen x_0 gerade $a + b$ ist.

Um zu beweisen, daß ein Flußdiagramm für alle Daten stoppt, wird in Floyd 67 folgendes Verfahren vorgeschlagen, das auch wieder die üblichen intuitiven Überlegungen systematisiert:

Man schreibt vor und nach jeder Instruktion einen „Bewertungsausdruck" F bzw. G, der nur von den Variablen, die im Flußdiagramm vorkommen, abhängt. Die Werte dieser Ausdrücke sollen sämtlich in einer geordneten Menge liegen, deren Ordnung die Eigenschaft hat, daß es keine unendlichen absteigenden Ketten gibt. Beispiele derartiger geordneter Mengen sind die natürlichen Zahlen mit der üblichen Ordnung oder z.B. n-Tupel natürlicher Zahlen (für irgend ein fixes n) mit der lexikographischen Ordnung. Wenn wir dann zeigen können, daß die Werte der Bewertungsausdrücke bei Exekution einer jeden Instruktion des Flußdiagramms immer echt kleiner werden, dann muß bei beliebigen Eingabewerten nach endlich vielen Schritten ein Stop erreicht werden.

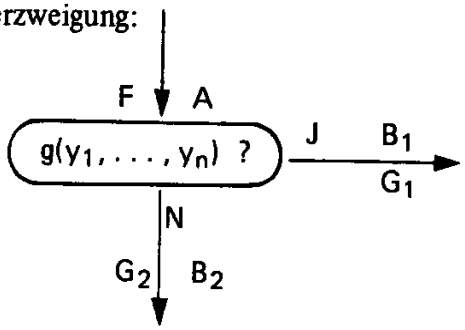
Genauer müssen wir die Regeln (K1) bis (K3) wie folgt ergänzen (wobei wir die Bewertungsausdrücke durch die – eventuell indizierten – Buchstaben F und G bezeichnen):

(K1') Zuweisungsinstruktionen:



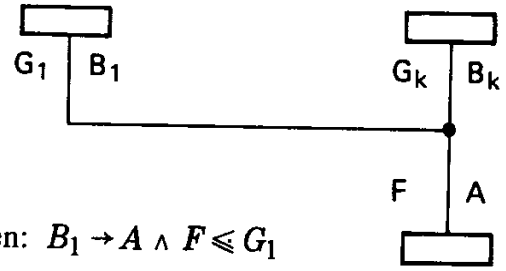
Zu zeigen: $x = f(\xi, y_1, \ldots, y_n) \wedge A^x_\xi \wedge F^x_\xi = w$
$\rightarrow B \wedge G \prec w$ ($\prec$ bezeichne die verwendete Ordnung)

(K2') Verzweigung:



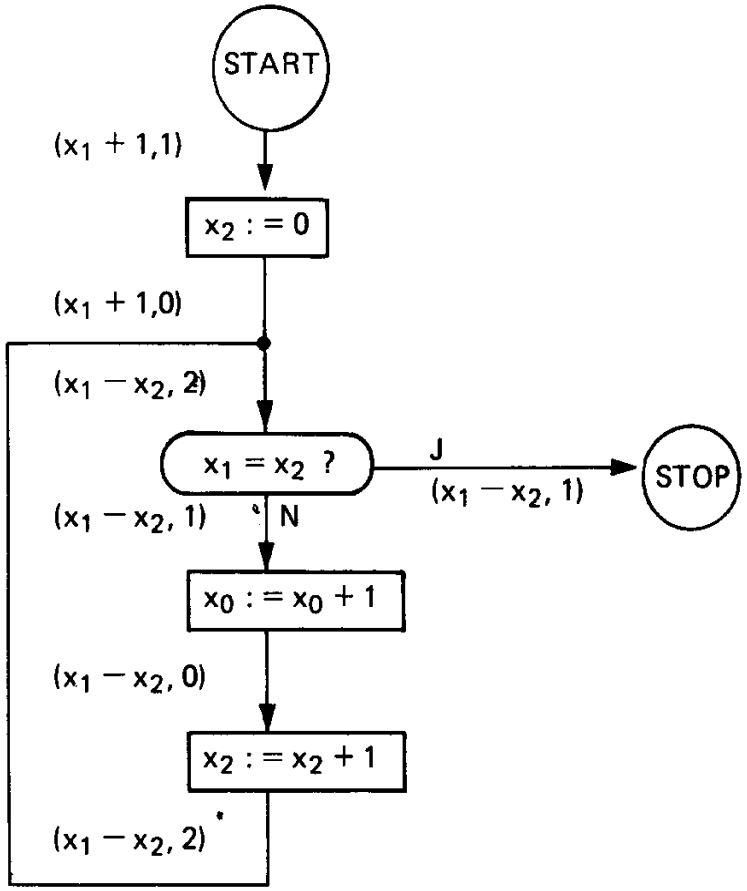
Zu zeigen: $A \wedge g \rightarrow B_1 \wedge G_1 \prec F$
$A \wedge \neg g \rightarrow B_2 \wedge G_2 \prec F.$

(K3') Eine Instruktion als Sprungziel mehrerer Instruktionen:



Zu zeigen: $B_1 \rightarrow A \wedge F \preccurlyeq G_1$
$\vdots$
$B_k \rightarrow A \wedge F \preccurlyeq G_k.$

Unser kleines Beispiel könnten wir wie folgt durch Paare von natürlichen Zahlen als Bewertungsausdrücke ergänzen. Als Ordnung würde die lexikographische Ordnung der Paare geeignet sein:



Bei der ersten Instruktion muß z.B. geprüft werden:

$x_2 = 0 \wedge x_0 = a \wedge x_1 = b \wedge (x_1 + 1,1) = w \rightarrow (x_1 + 1,0) < (x_1 + 1,1)$;

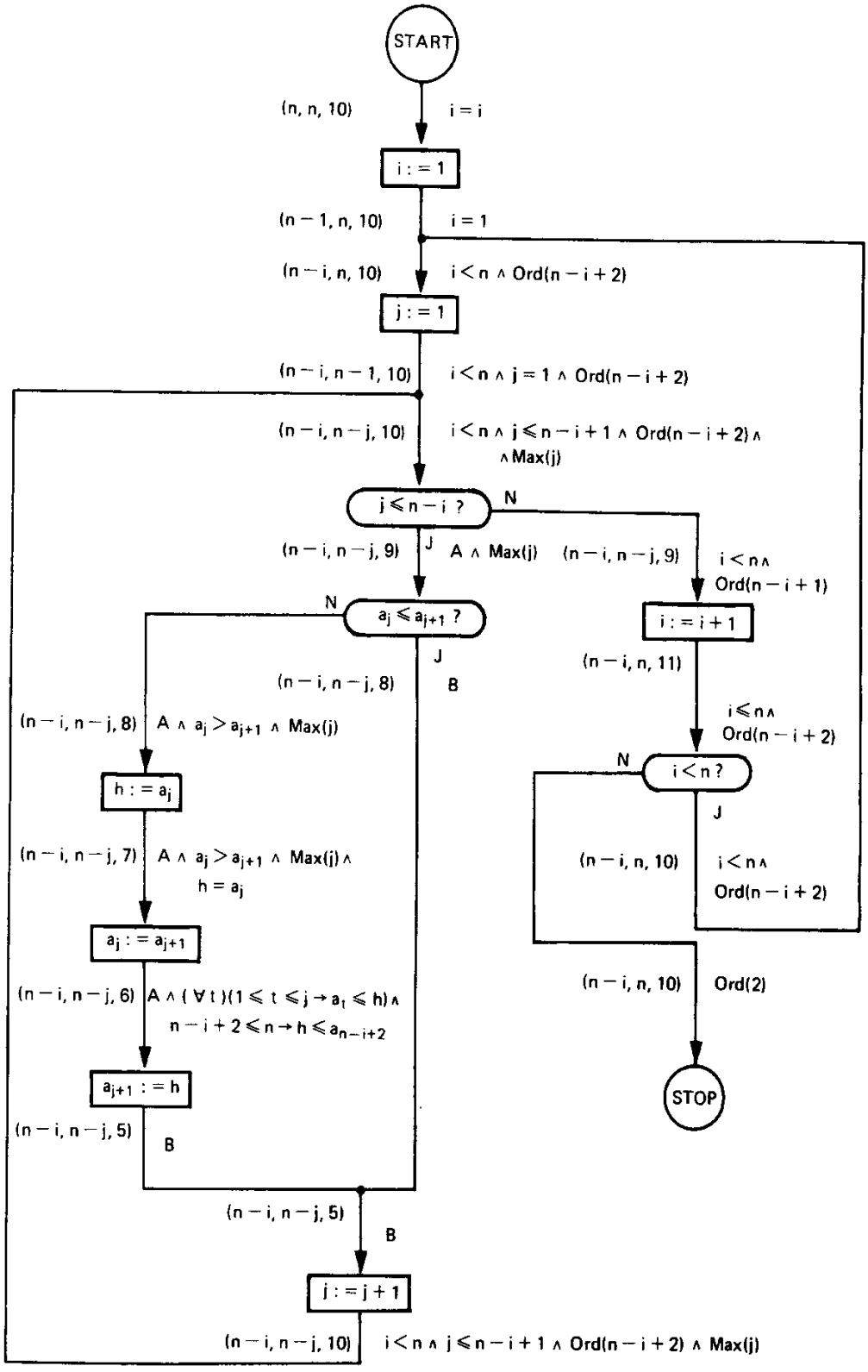
das ist richtig.

Ebenso muß z.B. bei der vierten Instruktion geprüft werden:

$$x_2 = \xi + 1 \wedge x_0 - 1 = a + \xi \wedge x_1 = b \wedge \xi < x_1 \wedge (x_1 - \xi, 0) = w \rightarrow (x_1 - x_2, 2) < w.$$

Nun ist aber $(x_1 - x_2, 2) = (x_1 - \xi - 1,2) < (x_1 - \xi, 0)$.

Die Übersichtlichkeit der hier skizzierten Methode kann durch Herausgreifen „wesentlicher" Stellen im Programm, zwischen denen der Beweisvorgang mechanisch durchgeführt werden kann, stark erhöht werden: siehe Manna 74, Kapitel 3.


START
(n, n, 10)
i = i
i : = 1
(n − 1, n, 10)
i = 1
(n − i, n, 10)
i < n ∧ Ord(n − i + 2)
j : = 1
(n − i, n − 1, 10)
i < n ∧ j = 1 ∧ Ord(n − i + 2)
(n − i, n − j, 10)
i < n ∧ j ⩽ n − i + 1 ∧ Ord(n − i + 2) ∧
∧ Max(j)
j ⩽ n − i ?
N
J
(n − i, n − j, 9)
A ∧ Max(j)
(n − i, n − j, 9)
i < n ∧
Ord(n − i + 1)
i : = i + 1
(n − i, n, 11)
i ⩽ n ∧
Ord(n − i + 2)
i < n ?
N
J
(n − i, n, 10)
i < n ∧
Ord(n − i + 2)
(n − i, n, 10)
Ord(2)
STOP
a_j ⩽ a_{j+1} ?
N
J
B
(n − i, n − j, 8)
(n − i, n − j, 8)
A ∧ a_j > a_{j+1} ∧ Max(j)
h : = a_j
(n − i, n − j, 7)
A ∧ a_j > a_{j+1} ∧ Max(j) ∧
h = a_j
a_j : = a_{j+1}
(n − i, n − j, 6)
A ∧ (∀ t)(1 ⩽ t ⩽ j → a_t ⩽ h) ∧
n − i + 2 ⩽ n → h ⩽ a_{n−i+2}
a_{j+1} : = h
(n − i, n − j, 5)
B
(n − i, n − j, 5)
B
j : = j + 1
(n − i, n − j, 10)
i < n ∧ j ⩽ n − i + 1 ∧ Ord(n − i + 2) ∧ Max(j)

Zum Abschluß geben wir noch ein ausgearbeitetes Beispiel eines Flußdiagramms für das Sortieren von n Zahlen $a_1, \ldots, a_n$ für $n \geqslant 2$ nach der Methode „Vertauschen von benachbarten Zahlen".

Der Leser möge nacheinander:

Die Instruktionen des Flußdiagramms so lange überdenken, bis ihm „klar" ist, wie das Verfahren funktioniert,
seine intuitiven Vorstellungen über das Funktionieren des Verfahrens in exakte Behauptungen über die Werte der vorkommenden Variablen an den einzelnen Stellen des Flußdiagramms umsetzen (und mit den bereits eingetragenen Behauptungen vergleichen),

ebenso seine intuitiven Vorstellungen, warum das Verfahren immer zu einem Ende kommen muß, in Bewertungsausdrücke umsetzen (und mit den bereits eingetragenen Ausdrücken vergleichen) und schließlich

die gemäß (K1') bis (K3') notwendigen Checks durchführen.

Wir verwenden im Flußdiagramm folgende Abkürzungen:

$\mathsf{Ord(k)} \quad : \leftrightarrow \quad (\forall t)\,(k \leqslant t < n \rightarrow a_t \leqslant a_{t+1}) \wedge \mathsf{Max(k)}$

$\mathsf{Max(k)} \quad : \leftrightarrow \quad (\forall t)\,(1 \leqslant t < k \leqslant n \rightarrow a_t \leqslant a_k)$

$\mathsf{A} \quad : \leftrightarrow \quad i < n \wedge j \leqslant n - i \wedge \mathsf{Ord}(n - i + 2)$

$\mathsf{B} \quad : \leftrightarrow \quad \mathsf{A} \wedge \mathsf{Max}(j + 1)$

Ord(2) ist dann genau die Behauptung, daß die Liste $(a_1, \ldots, a_n)$ sortiert ist.

Ich danke Herrn P. Sutter für die genaue Durchsicht des Manuskripts.

Literatur

W. Bibel 75: Predicative programming, Bericht des Instituts für Informatik, TU, München.

W. Bibel 76: Maschinelles Beweisen, Übersichtsartikel in diesem Band.

B. Buchberger 74: Certain decompositions of Gödel numberings and the semantics of programming languages, in: Internat. Symp. on Theoret. Programming, Novosibirsk (Lecture Notes in Computer Science, Vol. 5), Springer-Verlag, Berlin-Heidelberg-New York, S. 152 - 171.

B. Buchberger/B. Roider 75, Some results on universal automata, Bericht Nr. 34, Inst. f. Mathematik, Univ. Linz.

C.L. Chang/R.C.T. Lee 73: Symbolic logic and mechanical theorem proving, Academic Press.

R.L. Constable 74: Formal constructive mathematics as a programming language, Vorlesungsmanuskript, Univ. Saarbrücken.

R.W. Floyd, 67: Assigning meaning to programs, in: Proceed. Symp. on Applied Mathematics, Vol. 19, American Math. Soc., Providence, R.I., S. 19 - 3

P. Lucas/P. Lauer/ H. Stigleitner 68: Method and notation for the formal definition of programming languages, TR 25.087, IBM-Laboratorium, Wien.

Z. Manna/R.J. Waldinger 71: Toward automatic program synthesis, Communications of the ACM, Vol. 14/3, S. 151 - 165.

Z. Manna 74, Mathematical theory of computation, McGraw-Hill Book Company, New York.

Proceedings 75: Proceedings of the international conference on reliable software, Los Angeles, Association for Computing Machinery.

J.R. Shoenfield 67, Mathematical logic, Addison-Wesley Publ.Comp., London.

Anschrift des Autors:

Institut für Mathematik
Johannes-Kepler-Universität Linz

A-4020 Linz