PRÄDIKATIVES PROGRAMMIEREN

Wolfgang Bibel
Technische Universität München

Einleitung

Alle bekannten Programmiersprachen sind im weiteren Sinne Maschinensprachen; denn zu ihrem Verständnis ist eine gewisse Kenntnis der Struktur einer Rechenmaschine unerläßlich, sei es auch nur in Form eines abstrakten Modells. Elemente wie Variable, Wertzuweisungen, Referenzstufen, Sprünge etc., die in keiner dieser Sprachen fehlen, können hier als Beispiele angeführt werden. Genau diese Elemente sind es auch, die

1. dem durchschnittlichen Benützer die Aufgabe des Programmierens so erschweren,
2. die Schwierigkeiten im Zusammenhang mit der Semantik dieser Programmiersprachen hervorrufen und
3. einer fortschreitenden Automatisierung des Programmierprozesses im Wege stehen.

Noch bevor die ältesten dieser algorithmischen Sprachen entstanden sind, war den Logikern bekannt, daß jede Berechnung auch deduktiv durchgeführt werden kann (vgl. [4, Theorem 62]); diese Tatsache bedeutet, daß die Sprache der Prädikatenlogik zusammen mit einem allgemeinen Beweisverfahren eine Programmiersprache darstellt, die völlig frei ist von Elementen, wie sie oben beschrieben sind.

Erst jetzt, da man sich bewußt wird, daß die oben erwähnten Schwierigkeiten bei den algorithmischen Sprachen von inhärenter Natur sind, beginnt man, diese Möglichkeit als ernstzunehmende Alternative zu den algorithmischen Sprachen zu diskutieren [11; 7; 10; 6; 8], wobei offenbar auch die zunehmende Leistungsfähigkeit der entwickelten Beweisverfahren eine entscheidende Rolle spielt.

Diese Arbeit versucht, die enormen Vorteile, die sich bei diesem Ansatz im Vergleich zu den herkömmlichen Sprachen ergeben, exemplarisch herauszuarbeiten. Sie sind summarisch in den Feststellungen F1 - F8 innerhalb des Textes zusammengefaßt und bilden die Grundlage für ein zukünftiges Programmiersystem, dessen Umrisse in der Figur 3 skizziert sind.

1. Berechnung der Fakultätsfunktion

Anhand eines sehr einfachen Beispiels, nämlich der Fakultätsfunktion, soll zunächst die Möglichkeit einer deduktiven Berechnung demonstriert werden. Als zugehöriges Beweisverfahren wird ein vom Autor entwickeltes Verfahren zugrundegelegt, das auf einem Gentzen-artigen Kalkül des natürlichen Schließens basiert. Obwohl für ein tieferes Verständnis des folgenden eine gewisse Kenntnis dieses (oder eines ähnlichen) Verfahrens nötig wäre, muß - von wenigen Andeutungen abgesehen - aus Platzgründen hierzu auf die Arbeiten [3; 2; 4; 5] verwiesen werden.

Ausgangspunkt ist eine exakte Problembeschreibung in Form der üblichen Definition der Fakultätsfunktion:

1.1. Induktive Definition der Funktion fak(x):
(1) fak(0) = 1 und
(2) ist y der Wert von fak(x), dann ist fak(x+1) = y·(x+1).

Von hier ist es nur ein minimaler (und daher weitgehend automatisierbarer) Schritt zur Formalisierung dieser Definition in der Sprache der Prädikatenlogik:

1.2. $\mathrm{fak}(0) = 1 \wedge \forall x \forall y(\mathrm{fak}(x) = y \rightarrow \mathrm{fak}(x+1) = y\cdot(x+1)) \rightarrow \forall x \exists_1 y\ \mathrm{fak}(x) = y.$

Nun soll der Wert von $\mathrm{fak}(\bar{x})$ für einen bestimmten Eingabewert $\bar{x}$ berechnet werden, was, so behaupten wir, durch Erarbeitung eines Beweises für die Spezialisierung von 1.2 auf $\bar{x}$ möglich ist. Wählen wir $\bar{x} = 2$, so lautet diese Spezialisierung:

1.3. $\mathrm{fak}(0) = 1 \wedge \forall x \forall y(\mathrm{fak}(x) = y \rightarrow \mathrm{fak}(x+1) = y\cdot(x+1)) \rightarrow \exists_1 y\ \mathrm{fak}(2) = y.$

Ein solcher Beweis (genauer Ableitung im formalen System) ist in Figur 2 für eine zu 1.3 äquivalente Formel F dargestellt. Dabei ist die Darstellung so gewählt, daß sie in etwa das Vorgehen des Beweisverfahrens motiviert. Tatsächlich operiert dieses in seiner weitestentwickelten Form nicht wie in Figur 2 auf einer Menge von Formeln, sondern ausschließlich auf der gegebenen Formel, so daß man insbesondere aus dieser Darstellung unmittelbar keine Rückschlüsse auf den erforderlichen Aufwand ziehen kann. Dieses Operieren besteht grob gesprochen aus einer Folge von Vergleichen von je zwei *Literalen*, womit die Grundeinheiten einer Formel nach Abzug der logischen Zeichen bis auf ¬ bezeichnet werden (so besteht F aus den Literalen L1 ≡ ¬fak(0) = 1, L2 ≡ fak(x) = y, L3 ≡ ¬fak(x+1) = y·(x+1), L4 ≡ fak(2) = y). Jedem solchen Vergleich entspricht ein Vergleich in Figur 1, weshalb wir uns nun zur Illustration auf sie konzentrieren.

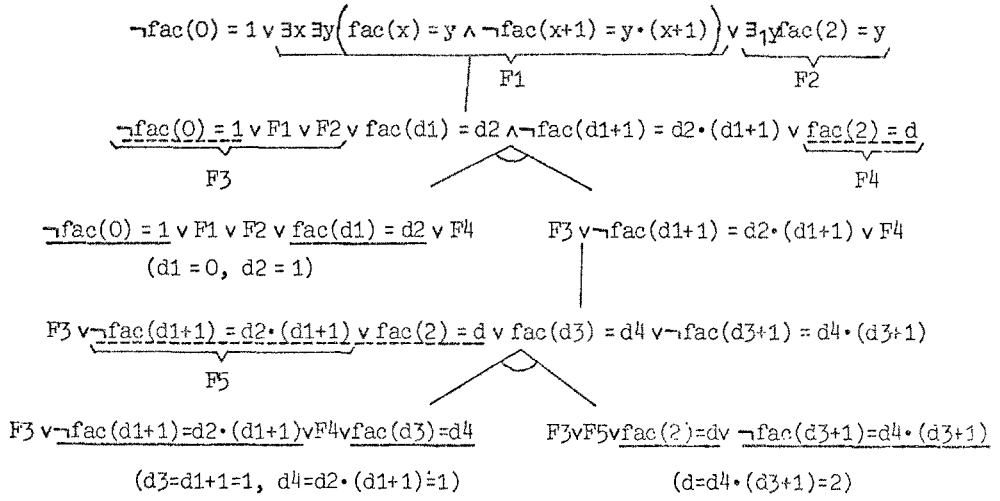

Fig. 1. Berechnung von fac(2) durch Herleitung

In der zweiten Zeile dieser Figur sind die durch Existenzquantoren in F gebundenen Variablen durch Platzhalter (Dummies) für Terme ersetzt, da man ja diese, deren Existenz behauptet wird, nicht von vornherein kennt. Nur bei dem letzten Quantor wird auch die Eindeutigkeit gefordert; bei den anderen kann (und muß) daher diese Ersetzung mit anderen Dummies wiederholt werden (Zeile 4). Jeweils nach Zeile 2 und 4 wird eine Konjunktion "aufgespalten". Bei den unterstrichenen Literalpaaren stellt man fest, daß sie nach Ersetzung der Dummies durch die in den angegebenen Gleichungen stehenden Terme *komplementär* sind, d.h. sich nur durch ein ¬ unterscheiden. Formeln, die solche komplementären Literalpaare enthalten, sind im logischen Sinne Axiome und brauchen nicht weiter reduziert zu werden. Alle Endformeln in Figur 1 sind solche Axiome; die Figur stellt also eine Ableitung von F dar und für den gesuchten Wert d ergibt sich das gewünschte Resultat 2.

Durch Auffinden einer Ableitung wurde also der gesuchte Wert bestimmt. Es ist eine in der Logik seit langem bekannte Tatsache, daß sich in dieser Weise uneingeschränkt jede rekursive Funktion berechnen läßt [12; 9]. In der Informatik muß man sich aber zugleich Gedanken darüber machen, inwieweit ein solches Berechnungsverfahren hinsichtlich des Aufwandes auch praktikabel ist, was im folgenden Abschnitt geschehen soll.

2. Das Programmkonzept für die Prädikatenlogik

In Figur 2 ist das Flußdiagramm für die übliche iterative Berechnung der Fakultätsfunktion angegeben. Vergleichend kann man feststellen, daß jede der Aktionen im Flußdiagramm bei dem Eingabewert 2 sich in einem ent-

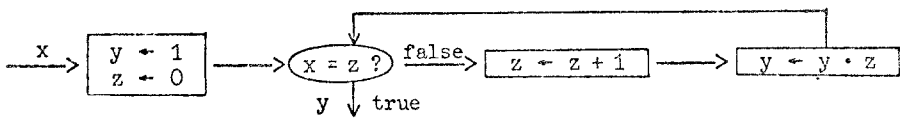

Fig. 2. Flußdiagramm zur Berechnung der Fakultätsfunktion

sprechenden Literalvergleich in der Figur 1 des ersten Abschnitts wiederfindet, was dort durch Unterstreichung bzw. -strichelung gekennzeichnet ist. So entsprechen den Initialisierungen in Figur 2 die Ersetzungen in Zeile 3 der Figur 1, dem ersten (nicht erfolgreichen) Test in Figur 2 der (nicht erfolgreiche) Literalvergleich in Zeile 2 der Figur 1, usw.. Wie man sich leicht überlegt, ist diese Aussage vom Eingabewert unabhängig.

Wie verhält es sich mit der Umkehrung dieser Aussage? Nun, das Verfahren, das ja völlig allgemein und problemunabhängig konzipiert ist, weiß natürlich ohne zusätzliche Information nicht von vornherein, welche Literalvergleiche zu einer vollständigen Ableitung führen, und wird daher eine Reihe von Literalvergleichen durchführen, die sich im nachhinein als überflüssig erweisen. Die Menge dieser überflüssigen Literalvergleiche wächst exponentiell mit dem Eingabewert, einem Mehraufwand, dem auf der Seite des Flußdiagramms nichts Vergleichbares gegenübersteht.

Dieses Mißverhältnis macht uns jedoch lediglich darauf aufmerksam, daß der Vergleich der beiden Verfahren unter inadäquaten Bedingungen ausgeführt wurde. Die Information, die in einem durch Figur 2 repräsentierten Programm steckt, umfaßt nicht nur die (implizit gegebene) exakte Problemstellung, sondern darüberhinaus eine genaue Vorschrift, in welcher Reihenfolge bestimmte Aktionen ausgeführt werden sollen. Eine solche zusätzliche Vorschrift, die wir im folgenden *Steuerung* nennen, fehlt in dem deduktiven Vorgehen noch vollständig, so daß der obige Vergleich gar nicht besser ausfallen *kann*.

Es liegt jedoch nach dem Vorangegangenen bereits auf der Hand, welche Information eine solche Steuerung dem Beweissystem zusätzlich zur Problemstellung an die Hand geben müßte: nämlich daß es im Beispiel der Figur 1 die unterstrichenen bzw. -strichelten Literalvergleiche und nur diese durchzuführen hat, womit überflüssige Vergleiche entfallen würden. Mit einer solchen zusätzlichen Steuerung ergibt sich offenbar für die beiden Berechnungsmethoden, der deduktiven und der algorithmischen, wie wir sie nennen wollen, ein vergleichbarer Berechnungsaufwand. Diese Erkenntnis wurde am Beispiel der Fakultätsfunktion gewonnen; sie läßt sich jedoch ohne Einschränkungen verallgemeinern. Doch gilt sie natürlich nur

in erster Näherung, da für eine vollständige Analyse das Beweisverfahren im Detail mit einem Compiler verglichen werden müßte (was erfahrungsgemäß sicher nicht zum Nachteil des Beweisverfahrens ausfallen würde). Aufgrund dieser Überlegungen geben wir also folgende Definition.

2.1. Ein prädikatives Programm ist ein Paar (D,S), wobei der Definitionsteil D eine Formel in der Sprache der Prädikatenlogik darstellt und der Steuerungsteil S die Reihenfolge der Literalvergleiche bei einem Beweis (einer Spezialisierung) von D bestimmt.

Zur Vervollständigung dieses Ansatzes müssen wir uns noch eine Darstellung für eine solche Steuerung überlegen. Sie muß die Reihenfolge der zu vergleichenden Literalpaare in D angeben und dabei verschiedene Inkarnationen des gleichen Literals unterscheiden. Zur Erläuterung ziehen wir nochmals das Beispiel der Figur 1 heran und wählen die Darstellung

2.2. $S_{fak(2)}=((L1^1,L4^1);(L1^1,L2^1);(L3^1,L4^1);(L2^2,L3^1);(L3^2,L4^1))$

Würde man sich die zu Figur 1 analoge Figur für den Fall des Eingabewertes 3 erarbeiten, so ergäbe sich analog

2.3. $S_{fak(3)}=((L1^1,L4^1);(L1^1,L2^1);(L3^1,L4^1);(L2^2,L3^1);(L3^2,L4^1);$
$(L2^3,L3^2);(L3^3,L4^1))$

Vergleicht man nun 1.5. und 1.6., so drängt sich die Vermutung auf, daß für einen beliebigen Eingabewert die folgende Steuerung zum Ziel führt:

2.4. $S_{fak}=((L1^1,L4^1);(L1^1,L2^1);[(L3^i,L4^1);(L2^{i+1},L3^i)]_{i=1;2;...})$

Bezeichnen wir jetzt die Formel 1.2 mit D_{fak}, so bildet (D_{fak},S_{fak}) ein prädikatives Programm. Offenbar ist jedoch eine solche Steuerung zu gegebenem D nicht eindeutig bestimmt. Selbst zu dem einfachen D_{fak} haben wir ja bereits im ersten Abschnitt die durch das Beweisverfahren vorgegebene Grundsteuerung S^o_{fak} kennengelernt, bei der auch die überflüssigen Literalvergleiche mit durchgeführt werden.
Im Hinblick auf das Problem, die Erarbeitung der Steuerung zu automatisieren, halte man sich den Weg von S^o_{fak} nach S_{fak} vor Augen: Mithilfe von S^o_{fak} wurde für kleine Testwerte S_{fak} berechnet und durch einen simplen Vergleich zweier Zeichenlisten S_{fak} erschlossen. Auch unter Berücksichtigung des erforderlichen Aufwands ist ersteres immer automatisch möglich, während letzteres mit bekannten heuristischen Methoden in nicht allzu komplexen Fällen ebenfalls automatisch gelingen kann.

Zusammenfassend ergeben sich also aus der bisherigen Diskussion in (zulässiger) Verallgemeinerung die folgenden Feststellungen.

F1. Das prädikative Programmkonzept ist so allgemein wie jedes andere Konzept.

F2. Unter gleichen Voraussetzungen unterscheidet es sich auch im erforderlichen Aufwand (in erster Näherung) nicht von den bekannten Programmkonzepten. - Soweit erscheint es also als völlig gleichwertig mit anderen Konzepten.

F3. Die natürliche Trennung zwischen Definitions- und Steuerungsteil schafft eine begriffliche Klarheit, die in anderen Konzepten nicht gegeben ist. Diese Klarheit schlägt sich in einer natürlichen und jedem logisch Denkenden geläufigen Semantik nieder. (Ein Blick auf den Definitionsteil verrät, worum es im Programm geht, was sich im Vergleich dazu bei einem algorithmischen Programm oft zu einer höchst komplizierten Interpretationsaufgabe ausweitet.)

F4. Diese einfachere Semantik zieht eine Vereinfachung der Programmierung eines Problems nach sich, die sich hier auf das Auffinden einer effizienten Steuerung (auf der Basis einer Grundsteuerung S^o) reduziert. Die sich dabei bietenden Möglichkeiten einer Automatisierung sind in anderen Konzepten in so einfacher Form nicht gegeben.

F5. Das Problem der Korrektheit eines Programms ist hier von untergeordneter Bedeutung (und einfach zu bewältigen), da der durch das Beweisverfahren vorgegebene Rahmen eine falsche Berechnung grundsätzlich ausschließt (da schlimmstenfalls die Berechnung ergebnislos abbricht).

3. Vertiefung des Konzepts

Das Beispiel der Fakultätsfunktion erlaubte es, das grundsätzliche Vorgehen beim prädikativen Programmieren in relativ einfacher Weise zu demonstrieren. Es ist jedoch nur von bedingter Überzeugungskraft, weshalb in diesem Abschnitt zwei weitere weniger triviale Probleme kurz behandelt werden sollen, an denen sich weitere Vorteile des Konzeptes zeigen werden.

Das erste ist die größter-gemeinsamer-Teiler-Funktion GGT. Ihre Definition ist praktisch durch ihren Namen gegeben und lautet in prädikatenlogischer Schreibweise

3.1. $\forall x\forall y\forall z(GGT(x,y)=z\leftrightarrow\exists x1(x1\cdot z=x)\wedge\exists y1(y1\cdot z=y)\wedge$
$\forall z1(\exists x1(x1\cdot z1=x)\wedge\exists y1(y1\cdot z1=y)\rightarrow z1\leq z))\rightarrow\forall x\forall y\exists_1 zGGT(x,y)=z$

Wie in 1.2. enthält die Prämisse (kurz PR) die Definition (hier in einer expliziten Form), während in der Konklusion die Funktionseigenschaft ausgedrückt wird. Entsprechend dem prädikativen Konzept müßte sich etwa der Wert von GGT(12,8) durch einen Beweis der folgenden Formel bestimmen lassen:

3.2. $PR\rightarrow\exists_1 zGGT(12,8)=z$

Wenn man dies in einer in Figur 1 illustrierten Weise durchführt, so ergibt sich das Unterproblem, Werte für die Dummies d,d1,d2 so zu bestimmen, daß die folgende Formel gilt:

3.3. $d1 \cdot d=12 \wedge d2 \cdot d=8 \wedge (\neg x1 \cdot z1=12 \vee \neg y1 \cdot z1=8 \vee z1 \leq d)$.

Tatsächlich würde dies mit einem allgemeinen Beweissystem auch gelingen, jedoch würde sein Vorgehen dem am wenigsten effizienten Algorithmus entsprechen, der etwa alle Tripel natürlicher Zahlen in einer aufsteigenden Ordnung durchtestet. Hier kann die mangelhafte Effizienz offenbar nicht durch einen besseren Steuerungsteil der im 2. Abschnitt beschriebenen Art verbessert werden.

Man hat in dieser Situation auch das Gefühl, daß hier die Definition des Problems allein zu dürftig ist, um auf so einfache Weise wie im zweiten Abschnitt zu einem effizienten Programm zu gelangen; zur Ansteuerung eines bestimmten unter allen möglichen Algorithmen, die in der Definition stecken, muß man diese durch weitere Kenntnisse über das Problem gewissermaßen einschränken. Im Falle des GGT könnte man es z.B. mit der Eigenschaft GGT(x-y,y)=GGT(x,y) für x>y versuchen, was anstelle vom 3.2. die Formel

3.4. $PR \wedge \forall x \forall y \forall z (x>y \wedge GGT(x-y,y)=z \rightarrow GGT(x,y)=z) \rightarrow \exists_1 z GGT(12,8)=z$

ergibt. Mit einer naheliegenden Steuerung führt dies zum folgenden Unterproblem:

3.5. $d1 \cdot d=4 \wedge d2 \cdot d=8 \wedge (\neg x1 \cdot z1=4 \vee \neg y1 \cdot z1=8 \vee z1 \leq d)$,

also einem gegenüber 3.3 wegen des kleineren Eingabewertes merklich einfacheren Problem.

Fährt man in dieser Weise fort und fügt zwei weitere offensichtliche Eigenschaften des GGT hinzu, so erhält man

3.6. $PR \wedge \forall x \forall y \forall z (x>y \wedge GGT(x-y,y)=z \rightarrow GGT(x,y)=z) \wedge$
$\forall x \forall y \forall z (GGT(x,y)=z \rightarrow GGT(y,x)=z) \wedge$
$\forall x GGT(x,x)=x \rightarrow \exists_1 z GGT(12,8)=z.$

Zu dieser Formel findet man mit wenig Übung (oder automatisch) leicht eine Steuerung, so daß das resultierende Berechnungsverfahren dem Euklidischen Algorithmus entspricht.

Dieses Beispiel lehrt das folgende:

F6. Zum effizienten Programmieren ist es im allgemeinen nötig, zur Definition noch weitere Kenntnisse über das Problem hinzuzuziehen, wobei man schrittweise und interaktiv vorgehen wird ("strukturiertes Programmieren"). Da diese Kenntnisse wiederum in ihrer natürlichen Form verarbeitet werden können, läßt sich z.B. der reiche Wissens-

fundus der Mathematik direkt einsetzen. Die Korrektheit der Eigenschaften (d.h. ihre Verträglichkeit mit der Definition) kann auf Wunsch das Beweisverfahren automatisch nachprüfen.

Das zweite Beispiel dieses Abschnitts ist die Fibonacci-Funktion FIB. Statt der iterativen Darstellung in 1.2. sei hier eine rekursive Definition gewählt.

3.7. $FIB(0)=1 \wedge FIB(1)=1 \wedge \forall z(FIB(z+2)=FIB(z+1)+FIB(z)) \rightarrow \forall x \exists_1 y FIB(x)=y$

Wie man sich leicht überlegt, erfordert zum Eingabewert $\bar{x}$ eine prädikative Berechnung $\bar{x}$ Inkarnationen des gleichen Programms genau wie beim üblichen rekursiven Programm. Neue Inkarnationen sind aber hier wie dort kostspielig und redundant. Im prädikativen Fall lassen sie sich aber generell durch (automatische) Anwendung der folgenden Regel vermeiden:

3.8. Ersetze jedes rekursive Auftreten einer Funktion durch eine neue Variable, füge die Identifizierung mit dieser Variablen als Prämisse hinzu, und quantifiziere solche Variablen.

Wendet man diese Regel auf 3.7. an, so ergibt sich

3.9. $FIB(0)=1 \wedge FIB(1)=1 \wedge \forall z \forall u \forall v(FIB(z+1)=u \wedge FIB(z)=v \rightarrow FIB(z+2)=u+v) \rightarrow \forall x \exists_1 y FIB(x)=y$

Hierzu findet man wiederum leicht die folgende Steuerung

3.10. $S_{FIB}=((L1^1,L6^1);(L2^1,L6^1);(L1^1,L3^1);(L2^1,L4^1);([(L5^i,L6^1);(L4^i,L3^{i+1});(L5^i,L4^{i+1})]_{i=1;2;\ldots})$

Aus diesem Beispiel ergibt sich

F7. In der Prädikatenlogik gibt es wohlbekannte Methoden (für weitere siehe etwa [4]), deren Anwendung auf ein gegebenes (prädikatives) Programm dieses automatisch zu einem effizienten transformieren.

In einem letzten Beispiel soll die Handhabung von Datenstrukturen allgemeinerer Art angedeutet werden. Im prädikativen Ansatz wäre eine allgemeine Listenstruktur wie folgt als geordnete, endliche Menge definiert:

3.11. $\forall x \forall <(list(x,<) \leftrightarrow finset(x) \wedge ord(x) \wedge \forall u \forall v(u \in x \wedge v \in x \wedge \neg u=v \rightarrow u<v \wedge \neg v<u \vee v<u \wedge \neg u<v))$

Dabei wird angenommen, daß finset,ord, bereits definierte oder primitive Prädikate darstellen. Ohne auf eine Reihe von interessanten Aspekten im Zusammenhang damit eingehen zu können, halten wir fest:

F8. Wie in algorithmischen Sprachen kann in der Prädikatenlogik auf Daten beliebigen Typs operiert werden. Ihre Definition, soweit sie nicht als primitiv angenommen sind, fügt sich zwanglos in den gegebenen Rahmen, d.h. erfolgt in der gleichen Sprache wie die Pro-

grammierung der Probleme selbst, womit eine Forderung erfüllt ist, wie sie etwa auch in [1] an ein Programmiersystem gestellt wird.

Abschließend sind die erarbeiteten Aspekte in Form einer groben Skizze eines zukünftigen prädikativen Programmiersystems in der Figur 3 zusammengefaßt.

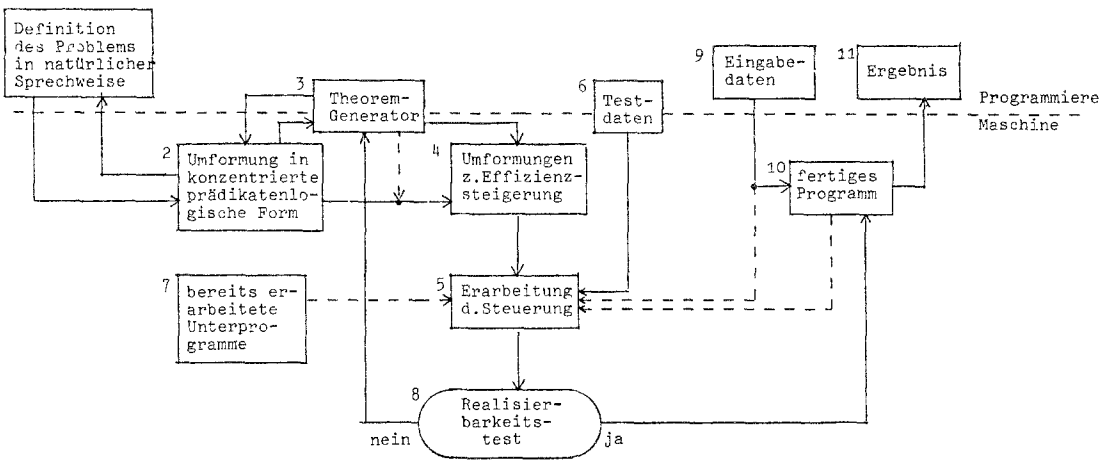

Fig. 3. Schema des Gesamtsystems

Danach wird die in 1 vom Programmierer gegebene Problemstellung interaktiv auf eine präzise maschinengerechte Form gebracht (2). Sodann werden Umformungen, wie sie am Beispiel der Fibonaccifunktion illustriert wurden, zur Steigerung der Effizienz in 4 vorgenommen. Zu diesem nunmehr vorliegenden Definitionsteil versucht die Maschine nun in 5, eventuell unter Ausnutzung bekannter Steuerungen (7), aus Steuerungen für Testdaten (6) auf eine allgemeine Steuerung heuristisch zu schließen. Das entstehende Programm wird in 8 daraufhin getestet, ob es für realistische Eingabewerte praktikabel ist. Ist dies nicht der Fall, so verlangt das System weitere Kenntnisse über das Problem. Auf lange Sicht erscheint auch hierfür eine teilweise Automatisierung denkbar, jedenfalls ist sie theoretisch möglich. Mit neuen Kenntnissen ausgestattet beginnt der Kreislauf von neuem, bis die in 8 gestellten Kriterien an Effizienz erfüllt sind. Damit ist dann die Programmierphase beendet und das resultierende Programm kann nun zu Berechnungen herangezogen werden.

Literaturverzeichnis

[1] Balzer, R.M.: *A global view of automatic programming.* Proc. Int. J. Conf. Artif. Intell. 3, Stanford, 494-499 (1973).

[2] Bibel, W.: *An approach to a systematic theorem proving procedure in first-order logic.* Computing 12, 43-55 (1974).

[3] Bibel, W. and Schreiber, J.: *Proof search in a Gentzen-like system of first-order-logic.* Bericht Nr. 7412, Techn.Universität München (1974). Erscheint in Proceedings of the International Computing Symposium (ICS 75), N.Holland P.C. (1975).

[4] Bibel, W.: *Programmieren in der Sprache der Prädikatenlogik.* Eingereicht als Habilitationsarbeit am Fachbereich Mathematik der Technischen Universität München (1975).

[5] Bibel, W.: *Predicative programming.* Unveröffentlichte Fassung in englischer Sprache (1975).

[6] Colmerauer, A., Kanoni, H., Roussel, P. and Pasero, R.: *Un système de communication homme-machine en francais.* Rapport, Université d'Aix-Marseille, Luminy (1972).

[7] Hayes, P.J.: *Computation and deduction.* Proc. Math. Foundations Comp. Science Symp., Czechoslovakian Acad. Science (1973).

[8] Hewitt, C.: *PLANNER: a language for proving theorems in robots.* Proc. Int. J. Conf. Artif. Intell. 1, Washington D.C., 219-239 (1969).

[9] Kleene, S.C.: *Introduction to Metamathematics.* North-Holland, Amsterdam (1952).

[10] Kowalski, R.: *Predicate logic as programming language.* Preprints IFIP 74, Stockholm, 569-574 (1974).

[11] Manna, Z. and Waldinger, R.J.: *Towards automatic program synthesis.* Comm. ACM 14, 151-165 (1971).

[12] Shoenfield, J.R.: *Mathematical logic.* Addison-Wesley, Reading (1967).