AXIOMATISIERUNG VON PROGRAMMIERSPRACHEN UND IHRE GRENZEN

Günter Hotz

Zusammenfassung : Es gibt für realistische Programmiersprachen kein endliches Axiomensystem, das es gestattet die einschlägigen Theoreme einer Theorie der Programmiersprache abzuleiten. Die Transportabilität von Programmen kann durch Korrektheitsbeweise im Rahmen einer axiomatischen Theorie nicht vollständig gesichert werden.

Einleitung : Dieses Kolloquium umfaßt praktische und theoretische Themen. Der Teilnehmerkreis ist entsprechend heterogen. Deshalb sei es mir gestattet, die Gesichtspunkte, die uns bei dem Axiomatisierungsversuch leiten, an klassischen Beispielen zu erläutern.

Das erste Beispiel einer axiomatisierten Theorie stellt die euklidische Geometrie dar. Geometrische Einsichten und ihr logischer Zusammenhang hatten schon Jahrhunderte vor Euklid großes Interesse gefunden, ohne daß jedoch ein Bedürfnis nach einer systematischen Durchdringung des aufgehäuften Wissens in diesen Zeiträumen nachgewiesen werden kann.

Das axiomatische Interesse besteht darin die angesammelten Erkenntnisse logisch aus möglichst wenigen, möglichst evidenten und widerspruchsfreien Resultaten der Theorie abzuleiten. Diese Resultate werden nun als gegeben hingenommen. Zu ihrer Begründung benötigen sie - vom formalen Standpunkt aus - nichts außer ihrer Widerspruchsfreiheit. Diese Grundlage für eine systematische Durchdringung des Wissens heißt ein Axiomensystem. Das Axiomensystem heißt vollständig, wenn es möglich ist, alle anderen Einsichten der Theorie mittels formalen Regeln - d.h. ohne Verwendung einer inhaltlichen Interpretation der Axiome - aus den Axiomen abzuleiten.

Wie sehen Axiome der euklidischen Geometrie aus ?

Nun man spricht in der Geometrie über Gegenstände, die Punkte und Geraden heißen und davon, daß Strecken durch Punkte gehen oder daß Punkte auf Strecken liegen, daß sich Geraden schneiden und Punkte auf einer Geraden liegen.

Die Gegenstände, um die es geht bilden also Mengen P und S. Die Elemente von P sind die Punkte, die Elemente von S sind die Strecken.

Daß ein Punkt auf einer Geraden liegt, oder eine Gerade durch einen Punkt geht, sind zwei verschiedene Bezeichnungen für den gleichen Sachverhalt, den man durch eine Relation

$$R_1 \subset P \times S$$

beschreiben kann. (p,g) ε R_1 bedeutet eben, daß p auf g liegt. Daß Geraden sich schneiden können, kommt in einer Relation

$$R_2 \subset P \times S \times S$$

zum Ausdruck : (p, g_1, g_2) ε R_2 bedeutet, daß g_1 und g_2 sich in p schneiden.

Entsprechend gehört zu " verbinden " eine Relation

$$R_3 \subset S \times P \times P,$$

mit der Interpretation (g, p_1, p_2) ε R_3 genau dann, wenn g durch p_1 und p_2 geht.

Nun sind die Relationen im Lichte unserer anschaulichen Interpretation nicht unabhängig voneinander. Es gilt dann vielmehr

$$(p, g_1, g_2) \varepsilon R_2 \Rightarrow (p, g_1) \varepsilon R_1 \text{ und } (p, g_2) \varepsilon R_1.$$

Weiter etwa

$$(g, p_1, p_2) \varepsilon R_2 \Rightarrow (g, p_2, p_1) \varepsilon R_2$$

oder

$$(g, p_1, p_2) \varepsilon R_2 \text{ und } (g', p_1, p_2) \varepsilon R_2 \Rightarrow g = g'.$$

Solche Eigenschaften von Relationen, die wir als " evident " oder " wünschenswert " als Basis unserer Theorie wählen, heißen im Rahmen der Theorie Axiome.

Axiomensysteme müssen widerspruchsfrei und sollen vollständig sein.

Ein Axiomensystem ist widerspruchsfrei, wenn sich aus ihm mittels logischer Schlüsse nicht zugleich ein Satz und die Negation dieses Satzes ableiten läßt. Die Widerspruchsfreiheit zeigt man durch Konstruktion eines Modells. Ein Axiomensystem heißt vollständig, wenn sich jeder Satz der Theorie, der richtig ist, aus den Axiomen logisch ableiten läßt.

Hieraus ergibt sich für unser Problem schon soviel, daß der Anspruch, eine Programmiersprache axiomatisch vollständig definiert zu haben, ohne eine Umschreibung der zu erfassenden Theorie als Tantologie aufgefaßt werden muß.

Uns interessiert an unserem Beispiel noch ein weiterer Gesichtspunkt : Man unterscheidet verschiedene Arten von Geometrien. Als Beispiel sei die projektive Geometrie genannt, die sich nur auf Aussagen wie schneiden und verbinden bezieht, ohne eine Anordnung der Punkte auf der Geraden ins Auge zu fassen, also eine reine Inzidenzgeometrie G_1. Durch Hinzunahme von Anordnungsaxiomen erhält man eine reichere Geometrie G_2. Man kann sich nun

fragen, ob man nicht natürlicher die Geometrie umgekehrt aufgebaut hätte, nämlich indem man Anordnungsaxiome an die Spitze stellt. Nun rein formal kann man das tun, nur wird kaum jemand ein System, das keine Aussagen über das Schneiden von Geraden und das Verbinden von Punkten macht, als Geometrie bezeichnen.

Wir merken uns für später :

Eine axiomatische Theorie, die in verschiedenen Richtungen Verfeinerungen enthält, besitzt eine natürliche Hierarchie in ihren Axiomen.

Wir wollen dies an einem zweiten Beispiel erläutern, das auch noch aus einem anderen Grund für uns wichtig wird, nämlich am Beispiel der Gruppentheorie.

In der Gruppentheorie haben wir es nur mit einer Menge G von Objekten zu tun, einer Relation, nämlich der Multiplikation und einiger weniger Axiome.

Eine Menge G mit einer Operation

$$\tau : G \times G \to G,$$

die die Gruppenaxiome erfüllen, heißt eine Gruppe.

Es gibt bekanntlich sehr verschiedene Gruppen, z.B. gibt es Gruppen mit 17 und solche mit 31 Elementen. Wenn man nun eine Rechnung in der Gruppe mit 17 Elementen in der Gruppe mit 31 Elementen machen will, dann gerät man in Schwierigkeiten, obwohl die Gruppe, in der wir die Berechnung nachbilden wollen, größer ist. Die Ursache liegt bekanntlich darin, daß es von der Gruppe mit 17 Elementen in die mit 31 Elementen nur einen Homomorphismus gibt, der alle Elemente auf das 1-Element abbildet. Das entsprechende gilt für die " Simulation der Berechnung " in der anderen Richtung.

Wir behalten von diesem Beispiel in Erinnerung :

Es gibt Gruppen derart, daß " Berechnungen " in einer Gruppe als Ergebnis das 1-Element liefern, während die " gleichen " Berechnungen in anderen Gruppen dies nicht tun.

Weiter fällt uns an diesem Beispiel auf, daß zahlentehoretische Eigenschaften der Anzahl der Elemente einer Gruppe in der Gruppentheorie eine hervorragende Rolle spielen, ohne daß die natürlichen Zahlen in der Gruppentheorie axiomatisch eingeführt werden. Das heißt, daß man Resultate der Zahlentheorie zwar verwendet, diese aber als untypisch oder gar störend in dem interessierenden Bereich ansieht.

Das wirft die Frage auf :

Was sind die für eine Theorie der Programmiersprache typischen Resultate,

was ist ein Theorem der Theorie der Programmiersprachen, was ein Axiom der Theorie ? Gehören z.B. die Axiome der natürlichen Zahlen in eine Theorie der Programmiersprachen ?

Bevor wir einen weiteren wichtigen Begriff an Hand der Gruppe erläutern, sei auch an diesem Beispiel auf die hierarchische Struktur in Axiomensystemen verwiesen.

Man kann z.B. von Gruppen zu abelschen Gruppen, stetigen Gruppen und durch Hinzunahme weiterer Relationen zu Ringen u.s.w. übergehen.

Wir haben oben den Begriff der Berechnung in Gruppen verwendet. Eine Berechnung ist hier eine Folge von Operationen, die auf eine gewisse Menge von Gruppenelementen angewendet wird. Man kann nun versuchen eine Berechnung nicht in konkreten Gruppen auszuführen, sondern symbolisch, indem man zur Umformung von Ausdrücken nur die Axiome der Gruppentheorie zuläßt.

Beispiel : Sind a und b Elemente irgendeiner Menge, dann bilden wir gruppentheoretische Ausdrücke wie

$$(a.b)^{-1}. (a.b)$$

und rechnen mit diesen Ausdrücken unter Verwendung der Axiome. Man erhält dann aus obigem Ausdruck etwa

$$(a.b)^{-1}.(a.b) = (b^{-1}.a^{-1}).(a.b) = ((b^{-1}.a^{-1}).a).b$$

$$= (b^{-1}.(a^{-1}.a)).b = b^{-1}.b = \underline{1}$$

Man erhält durch das symbolische Rechnen eine Gruppe, die durch die Elemente a und b " erzeugt " wird; diese Gruppe ist die freie Gruppe über { a, b }.

Damit haben wir genug Material zur Veranschaulichung der folgenden Ausführungen.

Zwei einfache Programme über Gruppen.

Es seien a.b und a^{-1} Gruppenoperationen.
Wir betrachten das Programm

```
begin
    x1 := a;
    x2 := a;
    x3 := 1;
```

```
m : x1:=x1.x1;
    x3:=x2.x3;
    if x1=1 then goto ℓ else goto m;
ℓ : end
```

Das Programm quadriert also den Inhalt von x_1 so lange bis 1 herauskommt. Ebenso oft wie es quadriert, multipliziert es a mit dem Inhalt von x_3. Nach n-maligem Durchlaufen der Schleife haben wir also

$$\text{content } (x_1)=a^{2^n} \text{ und content } (x_3)=a^n.$$

Rechnen wir in einer freien Gruppe, bricht das Programm niemals ab. Rechnen wir in der additiven Gruppe, dann haben wir nach n Zyklen

$$\text{content } (x_1)=2^n.a \text{ und content } (x_3)=n.a.$$

Ist unsere Gruppe die additive Gruppe von $\mathbb{Z}_{15}$, dann bricht der Algorithmus für $a \neq o$ auch niemals ab (das 1-Element ist hier die "o"!).

Rechnen wir in der additiven Gruppe von $\mathbb{Z}_{2^m}$, dann bricht der Algorithmus nach spätestens m Schritten ab. Starten wir die Rechnung mit a = 1 (hier nicht die Einheit der Gruppe), dann bricht der Algorithmus nach genau m Schritten mit

$$\text{content } (x_3)=m$$

ab.

Wir fassen zusammen :

Ein Programm mit Operationen, die den Axiomen der Gruppe genügen, kann bei Rechnungen in freien Gruppen niemals abbrechen, während es bei Rechnungen über endlichen Gruppen abbrechen kann oder auch nicht. Im Falle des Abbruches des Programmes, brauchen die Resultate bei Rechnungen in verschiedenen Gruppen nichts miteinander zu tun haben.

Wir ändern das Beispiel leicht ab :

```
begin
        x1:=a;
        x2:=a;
        x3:=1;
m :     x1:=x1.x1;
        x3:=x2.x3;
        if x1=1 then goto m else goto ℓ;
ℓ :     end
```

Wir haben also in der bedingten Anweisung nur die Sprungziele vertauscht.

Nun haben wir :

Bei Rechnungen in der freien Gruppe bricht das Programm immer ab. Es gibt endliche Gruppen, in denen das Programm niemals abbricht und solche in denen es für a ≠ o stets abbricht.

Axiomatisierung der Programmsprachen und Transportabilität von Programmen.

Den Hauptanstoß für die Versuche die Semantik von Programmiersprachen axioamtisch festzulegen, ging wohl von Floyd [1] aus. Vorzüglich durch Hoare [2], Manna [5] und Wirth [3] wurde dieser Ansatz theoretisch weit vorangetrieben und in [3] bei der Definition von Pascal einer praktischen Probe unterzogen.

Die Begründungen für diese Forschungen seien kurz zusammengefaßt.

1. Eine nur syntaktisch formal definierte Sprache läßt der Interpretation des Compilerkonstrukteurs zuviel Spielraum. Programme werden deshalb bei verschiedenen Sprachimplementationen verschiedene Resultate liefern.
2. Eine Sprache, deren Semantik formal vollständig definiert ist, gibt dem Benutzer die Möglichkeit die Korrektheit seines Programmes zu beweisen und so bei axiomatisch richtigen Implementationen der Sprache die Transportabilität seines Programmes zu sichern.
3. Die formalen Definitionen der Semantik unter Verwendung einer abstrakten Maschine wie z.B. bei der Wiener Methode [6] ist zu kompliziert, um auf dieser Basis Korrektheitsbeweise zu führen.

Wir wollen uns hier die Begründung kritisch anschauen.

Der erste Grund kann wohl von niemand bestritten werden. Die Notwendigkeit die verbale Definition durch eine formale zu ersetzen wird allgemein brennend empfunden.

Die zweite Begründung trifft nur zum Teil zu, wie unsere trivialen Beispiele aus dem vorigen Paragraphen zeigen.

a) Nämlich die beiden Programme sind für jede Implementation, der Operation a.b und a^{-1}, die den Gruppenaxiomen genügen, syntaktisch korrekt.

   Dies besagt weder etwas über die Terminierung der Programme, noch ihre Resultate im Falle der Terminierung aus.

b) Der Beweis, daß ein Programm eine bestimmte Funktion berechnet, unter alleiniger Verwendung der Axiome kann die *Korrektheit nur bei Rechnungen in der freien Gruppe* gewährleisten.

c) Die Implementation von Programmiersprachen auf Rechnern mit verschiedenen Wortlängen wird aber stets auf nicht isomorphe der axiomatisch definierten Algebra (Gruppe) führen und nie zu einer Realisierung der freien Algebra.

Folgerung aus a), b) und c).

Ohne eine genaue Untersuchung der in der Definition von Programmiersprachen festgelegten algebraischen Bereiche, kann man die Begründung 2. in dieser allgemeinen Form nicht aufrecht erhalten. Man wird bei Programmen, deren Korrekheit auf axiomatischer Basis nachgewiesen wurde, nur sicher sein können, daß gewisse Fehler ausgeschlossen sind.

Zum Beweis der " vollständigen Korrektheit " scheint stets der Nachweis notwendig zu sein, daß die konkrete Maschine, die freie Maschine (Maschine, die in den freien Algebren rechnet) " korrekt " simuliert.

Ist dieser Nachweis für eine konkrete Maschine erbracht, dann ist es nicht selbstverständlich, daß er für die anderen Maschinen überflüssig ist.

Nun zur dritten Begründung :

Die Definition der Semantik unter Verwendung einer abstrakten Maschine sind in der Tat sehr aufwendig und für einen Benutzer der Sprache schwer zumutbar. Zunächst muß man sagen, daß der obige Einwand bezüglich der Korrektheit von Programmen auf konkreten Maschinen, die auf der abstrakten Maschine korrekt sind, hier ebenso zutrifft. Falls die axiomatische Definition eine Programmiersprache so weit festlegt, wie die unter Verwendung einer abstrakten Maschine, stellt die abstrakte Maschine, die oben skizzierte freie Maschine dar.

Es stellt sich damit die Frage, wie aufwendig eine gleich vollständige axiomatische Definition einer Programmiersprache wird.

Der Umfang axiomatischer Definitionen von Programmiersprachen.

Wir erinnern uns an das in der Einleitung geschildete Schema axiomatischer Definitionen. Wir hatten eine Menge von Objektmengen, P_1, P_2,... eine Menge von Relationen R_1, R_2,... und eine Menge von Axiomen A_1, A_2,...,die Beziehungen zwischen den Relationen postulierten.

Was sind in einer Theorie der Programmiersprachen die Objekte ?

Zur Beantwortung dieser Frage gibt es nur die Möglichkeit, in der Literatur nachzusehen, welche Gegenstände in Definitionen von Programmiersprachen und in Aussagen über Programmiersprachen vorkommen, um dann die für programmiersprachen typischen Objekte auszusondern.

In der Arbeit von Floyd [1] werden einige Formen für Axiome der Programmiersprachen angegeben, die allgemein akzeptiert werden. Ein Beispiel ist : Für alle P_1, P_2, Q_1, Q_2 und C gilt

$$P_1 \{ C \} Q_1 \ \& \ P_2 \{ C \} Q_2 \Rightarrow (P_1 \wedge P_2) \{ C \} (Q_1 \wedge Q_2)$$

Hierin sind P_i und Q_i Prädikatsausdrücke und C ist ein operationeller Ausdruck. Die P_i, Q_i und C sind also Objekte, die in Relationen $P\{C\}Q$ stehen können und diese Relationen stehen in dem oben angegebenen Zusammenhang.

Das sind unendlich viele Axiome in denen beliebig komplizierte Objekte vorkommen.

Diese und die weiteren bei Floyd angegebenen Axiome besagen genau, daß Programmiersprachen aus Teilen C aufgebaut werden, die als Relationen im mengentheoretischen Sinn interpretiert werden sollen.

Hiergegen kann man wenig einwenden. Diese unendlich vielen Axiome haben eine sehr einfache Gestalt und sie sind unter natürlichen Voraussetzungen über die Prädikatsausdrücke aufzählbar im Sinne der rekursiven Funktionentheorie.

Ließen sich Programme stets aus primitiven Elementen C_i durch Aneinanderreihen

$$C_1 ; C_2 ; \ldots ; C_k$$

aufbauen, wäre man fertig. Leider beginnt aber hier erst das eigentliche Problem. Der Aufbau der Programme C aus primitiven Strukturen $C_1, \ldots$ ist sehr kompliziert, wie aus der syntaktischen Beschreibung der Programmiersprachen hervorgeht. Hoare und Wirth haben in [3] den Versuch unternommen die Semantik der sehr übersichtlich aufgebauten Programmiersprache Pascal zu einem wesentlichen Teil axiomatisch zu fassen. Hierbei treten in der Tat nahezu alle (vielleicht sind es alle, ich habe es nicht nachgeprüft) syntaktischen Grundbegriffe der Sprachdefinition von Pascal auf. - Nach meinem Empfinden sollten es alle sein, wenn nicht durch die syntaktischen Sprachmittel überflüssige Begriffe hereingekommen sind. - Dies sind aber größenordnungsmäßig 50 Begriffe. Das heißt, daß wir bei einer vollständigen axiomatischen Beschreibung einer höheren Programmiersprache mit einem unerhört großen Axiomensystem rechnen müssen.

Hier stellen sich sofort zwei Fragen :

Kann dieses Axiomensystem nicht sehr systematisch sein ? Wir haben dies doch oben am Beispiel eines unendlichen Axiomensystems gesehen.

Was heißt hier das Wort Vollständigkeit ?

Die erste Frage läßt sich nicht unabhängig von der zweiten beantworten.

Man kann diese Frage auch nicht nur von existierenden Programmiersprachen ausgehend beantworten.

Genauer: von der Definition existierender Sprachen ausgehend. Es ist denkbar, daß wir uns in der starken Beschränkung der syntaktischen Beschreibungsmittel bei der fast ausschließlichen Verwendung von kontextfreien Chromsky-Sprachen oder überhaupt von semi-Thue-Systemen ein Prokrustesbett geschaffen haben.

Was auf den ersten Blick auffällt ist das folgende : Nimmt man die Floyd'schen Axiome und die Ergänzung durch Manna[5], die die Behandlung des " While " aus, *dann besitzen die vorgeschlagenen Axiomensysteme keine hierarchische Struktur.*

Diesen Abschnitt zusammenfassend bemerken wir, daß auch die axiomatische Definition der Semantik von Programmiersprachen, wie es scheint notgedrungen, sehr aufwendig wird.

Die Strukturierung von Axiomensystemen.

Wenn wir die Versuche eines durchgehend axiomatischen Aufbaues der Mathematik anschauen, dann entdecken wir auch hier sehr umfangreiche Axiomensysteme. Nur begegnen diese einem Mathematiker so gut wie nir. Er richtet sich ein Arbeitsfeld axiomatisch her, bearbeitet es aber unter naiver Verwendung aller in der Mathematik zur Verfügung stehender Mittel.

In der Theorie der Programmierung oder der Programmiersprachen fehlt uns bis jetzt die Einsicht, ob und wie sich Bereiche abgrenzen lassen, die ein ähnlich selbständiges Leben nebeneinander führen können.

Zum Beispiel fehlt vollständig die Antwort auf die Frage, *was ist ein Axiom einer Theorie der Programmiersprachen und was nicht ?*
Haben die Axiome für die natürlichen Zahlen in einer Theorie der Programmiersprachen mehr zu suchen als in der Gruppentheorie ?

Eine Antwort auf die erste beider Fragen wäre ein Schritt auf eine hierarchische Strukturierung:

Man entwickelt zunächst eine Theorie der Programmiersprachen z.B. mit *freien* Typen. Das heißt, daß die Verwendung des Typenalphabates durch Axiome nicht eingeschränkt wird. Man betrachtet die mit den allgemeinen " Typenaxiomen " verträglichen Interpretationen. Ein Beispiel für die Behandlung dieser Typen findet man etwa in [4].

Man nimmt den ausgezeichneten Typ boolean auf. Prozedurtechniken. Schließlich Axiomensysteme für integer, real, usw.

Irgendwo in diesem hierarchischen Aufbau gibt es Abzweigungen zu Algol 68 oder Pascal.

Ein solcher Aufbau würde die Definition der einzelnen Sprachen sehr entlasten. Er würde es erlauben in allgemeinen Theorien für die Bearbeitung spezieller Probleme so viel Vorarbeit zu leisten, daß die große Kompliziertheit der Sprachen weniger drückend empfunden würde.

In die oberste Hierarchie gehören die Untersuchungen über *Programmschemata* [7].

Die Vollständigkeit :

Ein Axiomensystem, das für eine Theorie der Programmiersprachen in dem Sinn vollständig ist, daß es die Äquivalenz von Programmen nachzuweisen gestattet, die bei jeder zulässigen Interpretation die gleiche Funktion berechnen, gibt es nur für sehr allgemeine Theorien. Gibt es in der Sprache ein Axiomensystem für einen Typ, z.B. integer, der die Interpretation von Integervariablen auf die natürlichen Zahlen einschränkt, dann gibt es *nicht einmal ein aufzählbares* Axiomensystem für diesen Zweck. Dies folgt leicht aus dem bekannten Satz, daß die Turingprogramme für eine fest vorgegebene Funktion nicht aufzählbar sind.

Abschließende Zusammenfassung :

Die Entwicklung einer axiomatischen Theorie der Programmiersprachen erscheint notwendig und bei geeigneter Strukturierung der Axiomensysteme vielversprechend.

Vollständigkeit sowohl beweistheoretisch als auch für praktische Zwecke wird nicht erreichbar sein. Eine ergänzende Betrachtung der Simulation von zu den axiomatischen Theorien gehörigen freien Maschinen auf konkreten Maschinen erscheint stets als notwendig.

Als dringend wünschenswert stellt sich eine Untersuchung der Begriffe integer und real heraus. Hieraus sollte schließlich eine Normung der Maschinenarithmetik resultieren.

Literatur :

[1] Floyd, R.W. (1967) "Assigning Meanings to Programs" in Proz.Sym. in Applied Math. 19, Mathematical Aspects of Computer Science (Schartz, J.T. ed), Amer.Math.Soc. pp. 19-32

[2] Hoare, C.A.R. (1969) "An Axiomatic Basis for Computer Programming". Comm. ACM 12, pp. 576-583

[3] Hoare, C.A.R. and Wirth, N. (1973)"An axiomatic Definition of the Programming Language". Pascal Acta Information Vol 2, pp. 335-357

[4] Hotz, G. (1972) "Grundlagen einer Theorie der Programmiersprachen II". Berichte des Fachbereiches für Angew. Math. + Informatik, pp. 1-51

[5] Manna, Z. and Paueli, A. (1974) " Axiomatic Approach to total

Correctures of Programs", Acta Informatica Vol.3, pp. 243-265

[6] Lucas, P. and Walk, K. (1969) "On the Formal Description of PL/I", Annual Review in Automatic Programming, Vol. 6, Part 3, Pergamon Press

[7] Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics, Vol. 188, Springer-Verlag 1971 (ed E. Engler).

Anschrift des Verfassers: Prof.Dr.G.Hotz,Universität des Saarlandes
Fachbereich Angewandte Mathematik und Informatik
66 Saarbrücken