Skript zum Kurs
Einführung in die Computeralgebra
Wintersemester 1997/98

H.-G. Gräbe, Institut für Informatik,
http://www.informatik.uni-leipzig.de/~graebe

12. Februar 1998

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Das Anliegen dieses Kurses

Nach dem Siegeszug von Taschenrechner und (in klassischen Programmiersprachen geschriebenen) Numerik-Paketen spielen heute Computeralgebra-Systeme (CAS) mit ihren Möglichkeiten, auch stärker formalisiertes mathematisches Wissen, das algorithmisch aufbereitet ist, über eine einheitliche Schnittstelle zur Verfügung zu stellen, eine zunehmend wichtige Rolle. Sie werden zugleich in absehbarer Zeit wenigstens im naturwissenschaftlich-technischen Bereich den Kern umfassenderer Wissensrepräsentationssysteme bilden, die die dort heute oftmals noch anzutreffende Zweieinigkeit aus Taschenrechner und Formel- und Tabellensammlung ablösen werden.

Damit gehört der souveräne Umgang mit solchen Systemen zu einer der Grundfertigkeiten, die von Hochschul-Absolventen wenigstens des mathematisch-naturwissenschaftlichen Bereichs in Zukunft erwartet werden.

Wie für das Programmieren in klassischen Programmiersprachen ist hierbei ein ausgewogenes Verhältnis zwischen Erwerb von eigenen Erfahrungen und methodischer Systematisierung dieser Kenntnisse angezeigt. Hier gibt es viele Parallelen zum Einsatz des Taschenrechners im Schulunterricht. Obwohl dieser heute an verschiedenen Stellen zeitig im Fachunterricht eingesetzt wird und dabei etwa im Fach Physik auch zur Verschiebung von Lerninhalten geführt hat, wird doch nicht auf eine systematische Einführung verzichtet. Dies wohl vor allem deshalb, weil der qualifizierte Umgang mit dem Taschenrechner, insbesondere die Kenntnis seiner Eigenarten, Möglichkeiten und Grenzen, heute wie Lesen und Schreiben zu den elementaren Kulturtechniken gehört, die jeder Schüler mit Verlassen der Schule beherrschen sollte. Dasselbe gilt in noch viel größerem Maße für die Fähigkeiten von Hochschulabsolventen im Umgang mit CAS, deren Möglichkeiten, Eigenarten und Grenzen ob der Komplexität dieses Instruments viel schwieriger auszuloten sind.

Im Zentrum dieses Kurses soll deshalb das Vertrautmachen mit den Möglichkeiten, Grenzen, Inhalten und Methoden des Einsatzes von Computern zur Ausführung symbolischer Rechnungen stehen.

Im Gegensatz zu vielen Büchern, stellvertretend sei hier Hecks Maple-Einführung [9] genannt, soll dabei nicht eines der großen Systeme, sondern deren Gesamtheit im Mittelpunkt der Aufmerksamkeit stehen. Dies soll es besser als die Konzentration auf eines der Systeme ermöglichen, die grundlegenden Techniken und Begriffe, die mit der Anwendung des Computers für symbolische Rechnungen verbunden sind, abzuheben.

Ein solcher Zugang erscheint mir auch deshalb sowohl gerechtfertigt als auch wünschenswert, weil es mittlerweile für jedes der großen Systeme eine Fülle von einführender Literatur höchst unterschiedlicher Qualität und verschiedenen Anspruchsniveaus gibt. Mehr noch, die Entwicklerteams der großen Systeme haben in den letzten Jahren viel Energie darauf verwendet, ihre Produkte auch

von der äußeren Gestalt her nach modernen softwaretechnischen Gesichtspunkten aufzubereiten[1], so daß selbst ein ungeübter Nutzer in der Lage sein sollte, mit wenig Aufwand wenigstens die Grundfunktionalitäten des von ihm favorisierten Systems eigenständig zu erschließen. Bei Kenntnis grundlegender Techniken, die von all diesen Systemen verwendet werden, sollte diese Einarbeitung noch schneller gelingen.

Für den Nutzer von Computeralgebrasystemen wird es andererseits zunehmend schwieriger, die wirkliche Leistungsfähigkeit der Systeme hinter ihrer gleichermaßen glitzernden Fassade zu erkennen. Auch hierfür soll dieser Kurs ein gewisses Testmaterial zur Verfügung stellen, obwohl bei einem globalen Vergleich der Leistungsfähigkeit der verschiedenen Systeme durchaus Vorsicht geboten ist. Unterschiedliche Herangehensweisen im Design zusammen mit der jeweils spezifischen Entstehungsgeschichte führten dazu, daß die Leistungsfähigkeit unterschiedlicher Systeme in unterschiedlichen Bereichen sehr differiert und, mehr noch, sich verschiedene "mathematische Rigorosa" wie etwa in einem Teil der Wester-Liste [18] enthalten, mit sehr unterschiedlichem Aufwand im Nachhinein integrieren lassen. Unser Schwerpunkt soll deshalb mehr darauf liegen, wie ehrlich die Systementwickler die Grenzen ihres eigenen Tuns markiert haben, d.h. inwiefern sie die Geister, die sie in ihre Systeme implantiert haben, auch wirklich beherrschen.

Letzteres steht als generelle Einsatzvoraussetzung auch für den Nutzer eines Computeralgebrasystems an zentraler Stelle. Es ist in diesem Gebiet noch wichtiger als beim Taschenrechnereinsatz, sich kritisch mit dem Sinn bzw. Unsinn gegebener Antworten auseinanderzusetzen und sich über Art und Umfang möglicher Rechnungen und Antworten in groben Zügen im Klaren zu sein.

Allerdings begegnet man diesem "Zauberlehrlingseffekt", den Weizenbaum in seinem inzwischen klassischen Werk [17] in Bezug auf den Computereinsatz erstmals umfassend artikulierte, im Zusammenhang mit dem Einsatz moderner Technik immer wieder. Dieser als "Janusköpfigkeit" bezeichnete Effekt, daß der unqualifizierte und insbesondere nicht genügend reflektierte Einsatz mächtiger technischer Mittel zu unkontrollierten, unkontrollierbaren und in ihrer Wirkung unbeherrschbaren Folgen führen kann, wissen wir nicht erst seit Tschernobyl. Daraus resultierende Fragen sind inzwischen Gegenstand eines eigenen Wissensgebiets, des "technology assessment", das ins Deutsche nur sehr unvollkommen als "Technologiefolgenabschätzung" zu übertragen ist, vgl. [11]. Bezogen auf den Computer kann die Beschäftigung mit Computeralgebra auch hier wertvolle Einsichten vermitteln, die helfen, dessen Wekzeugcharakter gegenüber heute oft anzutreffender Fetischisierung mehr in den Vordergrund zu rücken.

1.2 Die Voraussetzungen

Symbolische Rechnungen sind das wohl grundlegendste methodische Handwerkszeug in der Mathematik und durchdringen alle ihrer Gebiete. Dabei stellt es sich heraus, daß vom methodischen Gesichtspunkt her ein kleines, wiederkehrendes algorithmisches Grundinstrumentarium in den verschiedensten Situationen variierend zum Einsatz kommt. Dies mag nicht überraschen, ist doch ein ähnliches Universalitätsprinzip programmiersprachlicher Mittel zur Beschreibung von Algorithmen und Datenstrukturen gut bekannt und kann sogar theoretisch begründet werden.

Entsprechend sind Computeralgebrasysteme bereits heute Vielzweckwerkzeuge, die eine gehörige Portion symbolischer Rechnungen in kompliziertesten mathematischen Kontexten übernehmen

[1] Daß dies nach dem Vorreiter MATHEMATICA mit einer stärkeren Kommerzialisierung auch der meisten anderen großen Systeme verbunden ist, mag der eine oder andere bedauern. Eine wichtige Erkenntnis scheint mir aber zu sein, daß sich im Gegensatz zur unmittelbaren Forschung an Algorithmen hierfür eben keine öffentlichen Mittel allokieren lassen. Selbst MuPAD als letzte Ausnahme von dieser Regel unter den großen CAS hat inzwischen Kommerzialisierungstendenzen erkennen lassen. Erstaunlich dagegen ist, wie durch den Wettbewerb zwischen den großen Systemen deren (Lizenz)preis trotz umfangreicher Neuerungen auf derzeit meist unter 1000 DM für eine Vollizenz gefallen ist. Mit diesen Mittel kann man im wesentlichen in der Tat nur diese Supportleistungen, nicht aber tieferliegende algorithmische Untersuchungen refinanzieren.

Die Diskussion der Chancen und Risiken des Zusamenfließens öffentlich geförderter wissenschaftlicher Arbeit und privatwirtschaftlicher Supportleistung an diesem Beispiel erschiene mir auch deshalb wünschenswert, weil ähnliche Entwicklungen in anderen Bereichen der Informatik (Stichwort: LINUX-Distributionen) ebenfalls stattfinden.

können. Für einen Einführungskurs stellt dies eine doppelte Schwierigkeit dar, da einerseits die zu demonstrierenden Prinzipien erst in nichttrivialen Anwendungen zu überzeugender Entfaltung kommen, andererseits solche Anwendungen ohne Kenntnis ihres Kontextes nur schwer nachvollziehbar sind. Dies verlangt, eine wohlüberlegte Auswahl zu treffen.

Im folgenden wollen wir uns deshalb beschränken auf die Darstellung

- wichtiger Design-Aspekte eines CAS,
- der Simplikationsproblematik,
- der Rolle algebraischer Zahlen,
- von Nullstellen von Polynomen sowie
- der Differentiation und Integration elementarer bzw. rationaler Funktionen.

Insbesondere werden wir das Feld der *Differentialgleichungen* ob der Schwierigkeit der damit verbundenen Mathematik, deren Behandlung in keinem Verhältnis zu den dabei gewinnbaren neuen Einsichten in Zusammenhänge des symbolischen Rechnens steht, nicht berühren.

1.3 Demonstation einiger Möglichkeiten eines CAS

Wir wollen die Einleitung abschließen mit einer recht ausführlichen Demonstration der Möglichkeiten, die sich durch den Einsatz eines Computeralgebrasystems als Rechenhilfsmittel eröffnen.

1.3.1 Ein CAS als Taschenrechner

Zuerst einmal kann man ein CAS natürlich als Taschenrechner verwenden (alle Demonstrationen mit MuPAD):

```
1.23+2.25;
```

```
                                3.48
```

```
12+23;
```

```
                                35
```

Wie dieser beherrscht ein CAS auch eine Reihe mathematischer Funktionen, so etwa die Fakultät:

```
10!;
```

```
                             3628800
```

```
100!;
```

```
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

Am letzten Beispiel erkennen wir schon eine Besonderheit: CAS können im Gegensatz zu Taschenrechnern mit ganzen Zahlen mit *voller* Genauigkeit rechnen. Die in ihnen implementierte *Langzahlarithmetik* erlaubt es, die entsprechenden Umformungen *exakt* auszuführen.

Ein solcher Ansatz hat zur Folge, daß im Gegensatz zum Taschenrechner und klassischen Programmiersprachen auch Zahlen wie

```
sqrt(2);
```

```
                                  1/2
                                 2
```

```
PI;
```

```
                                  PI
```

nicht durch numerische Näherungswerte ersetzt werden, sondern als *symbolische Ausdrücke* in einer Form stehenbleiben, die uns aus einem streng mathematischen Kalkül wohlbekannt ist. Ein solches Symbol ist gleichwohl mehr als eine syntaktische Einheit; dahinter verbirgt sich eine wohldefinierte *mathematische Semantik*. So ist etwa

$\sqrt{2}$ diejenige (eindeutig bestimmte) positive reelle Zahl, deren Quadrat gleich 2 ist.

Diese Semantik ist dem CAS (bis zu einem gewissen Grade) bekannt, wie etwa die folgenden Eingaben belegen:

```
sin(PI);
```

```
                                   0
```

```
sin(PI/4);
```

```
                                  1/2
                                 2
                                 ----
                                  2
```

Auch gewisse Vereinfachungen solcher Ausdrücke werden automatisch vorgenommen:

```
sqrt(54);
```

```
                                    1/2
                                3 6
```

```
sqrt(288);
```

```
                                     1/2
                               12 2
```

Jedem Symbol ist also eine Menge von *Eigenschaften* zugeordnet, die dessen mathematischen Gehalt widerspiegeln. Eine dieser Eigenschaften kann insbesondere darin bestehen, daß dem CAS auch ein Verfahren zur Bestimmung eines numerischen Näherungswerts bekannt ist.

```
float(PI);
```

```
                                3.141592653
```

```
float(sqrt(2));
```

```
                                1.414213562
```

Dieser numerische Näherungswert kann ebenfalls mit einer beliebig vorzugebenden Präzision berechnet werden, die softwaremäßig auf der Basis der Langzahlarithmetik operiert.

```
DIGITS:=50: float(PI); DIGITS:=NIL:

                3.14159265358979323846264338327950288419716939937751
```

Damit ist die Numerik zwar oft deutlich langsamer als die auf Hardware-Operationen zurückgeführte Numerik klassischer Programmiersprachen, erlaubt aber genauere Approximationen als letztere.

Das algorithmische Wissen eines CAS beschränkt sich natürlich nicht auf Eigenschaften einzelner Symbole. Ähnlich wie auf dem Taschenrechner stehen auch wichtige mathematische Funktionen und Operationen zur Verfügung, allerdings in einem wesentlich größeren Umfang. So wissen CAS, wie man von ganzen Zahlen den größten gemeinsamen Teiler, die Primfaktorzerlegung oder die Teiler bestimmt, die Primzahleigenschaft testet, nächstgelegene Primzahlen ermittelt und vieles mehr.

```
gcd(2^30-1,3^20-1);

                                    11


Factor(12!);

                                     10  5  2
                                 11 2   3  5  7

isprime(12!-1);

                                   TRUE

nextprime(12!);

                                 479001629
```

1.3.2 CAS sind programmierfähig

Auf Taschenrechnern hat man außerdem meist die Möglichkeit, einige Werte zwischenspeichern zu können. Moderne Taschenrechner verfügen sogar über rudimentäre Möglichkeiten zur Programmierung. Diese beiden Punkte sind natürlich für ein CAS kein Problem, da Instrumente zum automatischen Ablauf von Rechnungen, also eine entsprechende *Programmiersprache*, Teil des jeweiligen Interpreters sind. Bis auf DERIVE, welches (gegenwärtig) nur über rudimentäre Sprachelemente verfügt, werden dabei alle gängigen Sprachkonstrukte einer imperativen Programmiersprache unterstützt, die noch um einige Spezifika, die aus der Natur des symbolischen Rechnens folgen und über die weiter unter zu sprechen sein wird, erweitert sind.

Diese Fähigkeiten eines CAS können unser Verständnis von Mathematik bereits grundlegend erweitern. So werden Aufgaben der Art:

Bestimmen Sie die letzte Ziffer der Zahl 2^{100}*,*

wie sie etwa in Schülerarbeitsgemeinschaften zum Kennenlernen des Rechnens mit Resten gern gestellt wurden, gegenstandslos, da man die ganze Zahl leicht ausrechnen kann.

```
2^100;

                1267650600228229401496703205376
```

Der ursprüngliche Sinn der Aufgabe bestand darin, zu erkennen, daß man zur Berechnung der letzten drei Ziffern nicht die gesamte Potenz, sondern nur deren Rest (*mod* 10) berechnen muß und daß Potenzreste 2^k (*mod* 10) eine periodische Folge bilden. Um dasselbe Ziel zu erreichen, müßte man schon eine höhere Potenz verwenden, die auch das CAS nicht in sinnvoller Zeit zu berechnen in der Lage ist.

Allerdings kann man den Rechner für eine wesentlich weitergehende Untersuchung derselben Problematik einsetzen. Da ein CAS (nachdem dieser Gegenstand genügend geübt worden ist) in der Lage ist, die ermüdende Berechnung der Potenzreste zu übernehmen, können wir nach Regelmäßigkeiten für die Potenzreste für beliebige Moduln fragen.

```
2^k mod 10 $ k=1..20;

        2, 4, 8, 6, 2, 4, 8, 6, 2, 4, 8, 6, 2, 4, 8, 6, 2, 4, 8, 6

2^k mod  3 $ k=1..20;

        2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1

2^k mod 11 $ k=1..20;

       2, 4, 8, 5, 10, 9, 7, 3, 6, 1, 2, 4, 8, 5, 10, 9, 7, 3, 6, 1

2^k mod 17 $ k=1..20;

     2, 4, 8, 16, 15, 13, 9, 1, 2, 4, 8, 16, 15, 13, 9, 1, 2, 4, 8, 16
```

Diese wenigen Kommandos liefern eine Fülle von Material, das uns schnell verschiedene Regelmäßigkeiten vermuten läßt, die man dann gezielter experimentell untersuchen kann. So taucht für primen Modul in der Folge stets eine 1 auf, wonach sich die Potenzreste offensichtlich wiederholen. Geht man dieser Eigenschaft auf den Grund, so erkennt man die Bedeutung primer Restklassen. Auch die Länge der Folge zwischen dem Auftreten der 1 folgt gewissen Gesetzmäßigkeiten, die ihre Verallgemeinerung in elementaren Aussagen über die Ordnung von Elementen in der Gruppentheorie finden.

Wir wollen diese auch aus didaktischen Gesichtspunkten interessanten Möglichkeiten eines CAS zur Erforschung von Eigenschaften ganzer Zahlen an einem weiteren Beispiel verdeutlichen:

Eine Zahl n heißt *perfekt*, wenn sie mit der Summe ihrer echten Teiler übereinstimmt, d.h. die Summe *aller* ihrer Teiler gerade $2\,n$ ergibt. Die Untersuchung solcher Zahlen kann man bis in die Antike zurückverfolgen. Bereits in der Pythagoräischen Schule im 6. Jh. vor Christi werden solche Zahlen betrachtet. EUKLID kannte eine Formel, nach der man alle gerade perfekte Zahlen findet, die erstmals von EULER exakt bewiesen wurde.

Wir wollen nun ebenfalls versuchen, uns einen Überblick über die perfekten Zahlen zu verschaffen. Das MuPAD-Paket `numlib` enthält die Funktion `sigma(n)`, mit der wir die Teilersumme der natürlichen Zahl n berechnen können. Mit der folgenden Schleife verschaffen wir uns erst einmal einen Überblick über die perfekten Zahlen bis 800:

```
export(numlib);
for i from 2 to 800 do
        if 2*i=sigma(i) then print(i) end_if
end_for;
```

```
6

28

496
```

Betrachten wir die gefundenen Zahlen näher:

$$6 = 2 \cdot 3, \quad 28 = 4 \cdot 7, \quad 496 = 16 \cdot 31.$$

Alle diese Zahlen haben die Gestalt $2^{k-1}(2^k - 1)$. Wir wollen deshalb versuchen, perfekte Zahlen dieser Gestalt zu finden:

```
for k from 2 to 40 do n:=2^(k-1)*(2^k-1):
        if 2*n=sigma(n) then print(k,n,yes)
        else print(k,n,no)
        end_if
end_for;
```

Die Ausgabe ist wegen ihrer Länge weggelassen, stellt aber umfangreiches experimentelles Material zur Verfügung, auf dessen Basis sich qualifizierte Vermutungen über bestehende Gesetzmäßigkeiten aufstellen lassen. Es scheint insbesondere so, als ob perfekte Zahlen nur für prime k auftreten. Jedoch liefert nicht jede Primzahl k eine perfekte Zahl n.

Derartige Beobachtung kann man nun versuchen, mathematisch zu untermauern, was in diesem Fall nur einfache kombinatorische Überlegungen erfordert: Die Teiler der Zahl $n = p_1^{a_1} \cdot \ldots \cdot p_m^{a_m}$ haben offensichtlich genau die Gestalt $t = p_1^{b_1} \cdot \ldots \cdot p_m^{b_m}$ mit $0 \le b_i \le a_i$. Ihre Summe beträgt also

$$\sigma(n) = (1 + p_1 + \ldots + p_1^{a_1}) \cdot \ldots \cdot (1 + p_m + \ldots + p_m^{a_m}).$$

Die Teilersumme ergibt sich also unmittelbar aus der Kenntnis der Primfaktorzerlegung der Zahl n.

Insbesondere ist die Teilersummenfunktion eine sogenannte *multiplikative* Funktion, d.h. es gilt $\sigma(a\,b) = \sigma(a)\sigma(b)$, wenn a, b zueinander teilerfremd sind. Ist $n = 2^{k-1}b$ eine gerade perfekte Zahl, so gilt also

$$2n = 2^k\, b = \sigma(n) = (2^k - 1)\sigma(b).$$

Damit muß b selbst ein Vielfaches von $2^k - 1$ sein, also $b = (2^k - 1)c$ gelten. Damit ist aber $\sigma(b) = 2^k\, c = b + c$. Da b und c beides Teiler von b sind, darf b keine weiteren Teiler haben, muß also eine Primzahl sein. Da auch umgekehrt jede solche Zahl eine perfekte Zahl ist haben wir damit folgenden Satz bewiesen:

Satz 1 *Jede gerade perfekte Zahl hat die Gestalt $n = 2^{k-1}(2^k - 1)$, wobei $P := 2^k - 1$ eine Primzahl sein muß.*

Ungerade perfekte Zahlen sind bis heute nicht bekannt, ein Beweis, daß es solche Zahlen nicht gibt, aber auch nicht gefunden worden.

Primzahlen der Gestalt $2^k - 1$ nennt man *Mersennesche Primzahlen.*

AUFGABE: Zeigen Sie, daß eine Zahl der Form $2^k - 1$ nur dann eine Primzahl sein kann, wenn k selbst prim ist.

Die größten heute bekannten Primzahlen sind Mersennesche Primzahlen.

1.3.3 Rechnen mit rationalen Zahlen und Funktionen

Der Grundgedanke des Einsatzes einer exakten Arithmetik verlangt es auch, daß rationale Zahlen als solche behandelt werden.

```
1/2+1/3;
```

$$5/6$$

```
sum(1/i,i=1..50);
```

$$13943237577224054960759/3099044504245996706400$$

```
float(%);
```

$$4.499205338$$

Die wichtigste Besonderheit eines CAS gegenüber Taschenrechner und klassischen Programmiersprachen ist jedoch die Fähigkeit zur Manipulation symbolischer Ausdrücke. Dies wollen wir zunächst an symbolischen Ausdrücken demonstrieren, die gewöhnliche Variablen enthalten, also Literale wie x, y, z ohne weitergehende mathematische Semantik. Aus solchen Symbolen kann man mit Hilfe der 4 Grundrechenarten *rationale Funktionen* zusammenstellen. Betrachten wir dazu einige Beispiele:

```
u:= a/((a-b)*(a-c)) + b/((b-c)*(b-a)) + c/((c-a)*(c-b));
```

$$\frac{a}{(a-b)\,(a-c)} + \frac{b}{(-a+b)\,(b-c)} + \frac{c}{(-a+c)\,(-b+c)}$$

Es ergibt sich die Frage, ob man diesen Ausdruck weiter vereinfachen kann. Am besten wäre es, wenn man einen gemeinsamen Zähler und Nenner bildet, welche zueinander teilerfremd sind, und diese in ihrer expandierten Form darstellt. Das ermöglicht die MuPAD-Funktion `normal`.

```
normal(u);
```

$$0$$

Das Ergebnis mag verblüffen; jedenfalls sieht man das dem ursprünglichen Ausdruck nicht ohne weiteres an.

Eine solche normalisierte Darstellung ist nicht immer zweckmäßig, weshalb diese Umformung von MuPAD nicht automatisch vorgenommen wurde. Betrachten wir etwa die Summe der Stammbrüche $\frac{1}{x-i}$ für $i = 1, \ldots, 6$:

```
u:=sum(1/(x-i) , i=1..6);
```

$$\frac{1}{x-1} + \frac{1}{x-2} + \frac{1}{x-3} + \frac{1}{x-4} + \frac{1}{x-5} + \frac{1}{x-6}$$

```
normal(u);
```

$$\frac{3248\,x^5 - 2205\,x^4 + 700\,x^3 - 105\,x^2 + 6\,x - 1764}{-1764\,x^6 + 1624\,x^5 - 735\,x^4 + 175\,x^3 - 21\,x^2 + x + 720}$$

```
Factor(u);
```

$$\frac{(2\,x - 7)\,(-392\,x^4 + 203\,x^3 - 42\,x^2 + 3\,x + 252)}{(x - 1)\,(x - 2)\,(x - 3)\,(x - 4)\,(x - 5)\,(x - 6)}$$

Hier sieht die erste der drei Darstellungen am einfachsten aus, die (im obigen Sinne) normalisierte am kompliziertesten. Die berühmteste Beispielklasse, die zeigt, daß eine Normalisierung kontraproduktiv sein kann, enthält das folgende Beispiel:

```
u:=(x^15-1)/(x-1);
```

$$\frac{x^{15} - 1}{x - 1}$$

```
normal(u);
```

$$x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$$

Betrachten wir allgemein Ausdrücke der Form

$$u_n := \frac{a^n}{(a-b)*(a-c)} + \frac{b^n}{(b-c)*(b-a)} + \frac{c^n}{(c-a)*(c-b)}$$

und untersuchen, wie sie sich unter Normalformbildung verhalten. Wir verwenden dazu die Funktion `subs`, die lokal eine Variable durch einen anderen Ausdruck, in diesem Fall eine Zahl, ersetzt.

```
normal(subs(u,n=2));
```

$$1$$

```
normal(subs(u,n=3));
```

$$a + b + c$$

```
normal(subs(u,n=4));
```

$$a\,b + a\,c + b\,c + a^2 + b^2 + c^2$$

```
normal(subs(u,n=5));
```

```
              3    3    3      2    2        2    2        2    2
     a b c + a  + b  + c  + a b  + a  b + a c  + a  c + b c  + b  c
```

Wir sehen, daß sich in jedem der betrachteten Fälle die rationale Funktion u_n zu einem Polynom vereinfacht.

AUFGABE: Finden Sie eine allgemeine Formel für u_n. Zeigen Sie insbesondere, daß u_n immer ein Polynom ist.

Die Möglichkeit zur Manipulation derartiger symbolischer Ausdrücke kann natürlich mit den programmiersprachlichen Mitteln, die wir weiter oben im Zusammenhang mit ganzen Zahlen kennengelernt haben, zu einem leistungsfähigen *symbolischen Taschenrechner* kombiniert werden.

1.3.4 Rechnen mit Wurzelausdrücken und komplexen Zahlen

Im letzten Abschnitt hatten wir gesehen, daß CAS symbolische Ausdrücke, in denen Variablen (ohne weitere Eigenschaften) durch die Grundrechenoperationen verbunden sind, manipulieren können. Dies gilt ebenfalls für solche Ausdrücke, die Symbole mit Eigensschaften wie etwa Wurzelausdrücke enthalten. Eines der besonderen Symbole ist dabei die imaginäre Einheit I. CAS können also auch mit komplexen Zahlen exakt rechnen:

```
(1+I)^n $ n=1..10;
```

```
 1 + I, 2 I, - 2 + 2 I, -4, - 4 - 4 I, - 8 I, 8 - 8 I, 16, 16 + 16 I, 32 I
```

```
(3+I)/(2-I);
```

```
                                   1 + I
```

Dasselbe gilt natürlich auch für andere Wurzelsymbole.

```
expand((sqrt(2)+sqrt(3))^n) $ n=2..4;
```

```
              1/2  1/2          1/2      1/2      1/2  1/2
          2 2    3    + 5, 11 2    + 9 3   , 20 2    3    + 49
```

```
radsimp(1/(sqrt(2)+sqrt(3)));
```

```
                                1/2    1/2
                             - 2    + 3
```

Allerdings wurden die letzten Umformungen (Expandieren eines Ausdrucks, Rationalmachen eines Nenners) nicht automatisch ausgeführt, sondern mußten vom System MuPAD explizit angefordert werden.

1.3.5 Differenzieren und Integrieren

Obwohl dem Namen nach eigentlich auf *algebraische* Manipulationen ausgerichtet, enthält ein CAS auch umfangreiches Wissen aus der Analysis. Dies ist nicht weiter verwunderlich, da die konkrete Berechnung von Ableitungen und Stammfunktionen elementarer Funktionen nicht nach der Definition

erfolgt, sondern durch geschicktes Kombinieren verschiedener *Regeln* auf entsprechende Grundintegrale und -Ableitungen zurückgeführt wird. Solche Umformungen haben aber eher algebraischen als analytischen Charakter.

```
diff(x^n,x);
```

$$\frac{n\, x^{n}}{x}$$

```
diff(x^2*sin(x)*ln(x+a),x$2); # Zweite Ableitung #
```

$$\frac{4\, x \sin(x)}{a + x} + 2 \sin(x) \ln(a + x) + 4\, x \cos(x) \ln(a + x) + \frac{2\, x^{2} \cos(x)}{a + x}$$

$$- x^{2} \sin(x) \ln(a + x) - \frac{x^{2} \sin(x)}{(a + x)^{2}}$$

Zur Berechnung von Stammfunktionen verfügen CAS über das Wissen guter Integraltafeln, allerdings in algorithmischer Form, d.h. sie kennen die wichtigsten Substitutionsregeln und wann diese erfolgreich anzuwenden sind.

```
p:=(2*x^3+2*x^2+2*x-2)/(x^2+1)^2;
```

$$\frac{2 x + 2 x^{2} + 2 x^{3} - 2}{(x^{2} + 1)^{2}}$$

```
int(p,x);
```

$$- \frac{2 x}{x^{2} + 1} + \ln(x^{2} + 1)$$

```
diff(%,x);
```

$$- \frac{2}{x^{2} + 1} + \frac{2 x}{x^{2} + 1} + \frac{4 x^{2}}{(x^{2} + 1)^{2}}$$

```
normal(%-p);
```

```
                                0
```

```
int(1/(x^4-1),x);
```

```
                  / 1 \
              atan| - |                              2     1/2
                  \ x /   ln(x + 1)   ln((- 2 x + x  + 1)   )
              --------- - --------- + -----------------------
                  2           4                  4
```

Dasselbe gilt für Integranden, die trigonometrische Funktionen enthalten.

```
p:=sin(x)^2*cos(3*x-2): int(p,x);
```

```
                  sin(x - 2)   sin(3 x - 2)   sin(5 x - 2)
                - ---------- + ------------ - ------------
                      4             6              20
```

```
q:=diff(%,x);
```

```
                  cos(x - 2)   cos(3 x - 2)   cos(5 x - 2)
                - ---------- + ------------ - ------------
                      4             2              4
```

```
simplify(q-p);
```

```
                        cos(x - 2)   cos(- x + 2)
                      - ---------- + ------------
                            4             4
```

```
expand(%);
```

```
                                0
```

Wir sehen an diesem Beispiel zugleich, daß es oft nicht einfach ist, Simplifikationen so vorzunehmen, daß sie das einfachste Ergebnis liefern.

Die Ergebnisse der verschiedenen Systeme können sich dabei auch voneinander unterscheiden. Berechnet man etwa

$$\int \frac{x}{(a^2 + x^4)}$$

so liefert Maple

```
                                 2
                                x
                           arctan(----)
                                  a
                        1/2 ------------
                                a
```

MuPAD dagegen

```
                 /  2      2 /     1   \1/2 \
/     1   \1/2   | x  + 4 a  | - ----- |    |
| - ----- |    ln|           |       2 |    |
|       2 |      \           \   16 a  /    /
\   16 a  /
```

```
                   /  2      2 /     1   \1/2 \
  /     1   \1/2   | x  - 4 a  | - ----- |    |
- | - ----- |    ln|           |       2 |    |
  |       2 |      \           \   16 a  /    /
  \   16 a  /
```

und Reduce schließlich

```
          sqrt(a)*sqrt(2) - 2*x           sqrt(a)*sqrt(2) + 2*x
  - (atan(----------------------) + atan(----------------------))
             sqrt(a)*sqrt(2)                 sqrt(a)*sqrt(2)
------------------------------------------------------------------
                              2*a
```

Alle drei Ergebnisse sind korrekt, jedoch ist es hier schon schwieriger, das auch zu überprüfen.

Schließlich sind die Systeme auch in der Lage, Reihenentwicklungen verschiedener Funktionen zu berechnen, so etwa die Taylorentwicklungen folgender Funktionen bis zur Ordnung 10:

```
taylor(sin(x),x=0,10);
```

```
                      3    5      7       9
                     x    x      x       x         10
                x - -- + --- - ---- + ------ + O(x  )
                     6    120   5040   362880
```

```
taylor(cos(x),x=0,10);
```

```
                       2    4    6      8
                      x    x    x      x         10
                 1 - -- + -- - --- + ----- + O(x  )
                      2    24   720   40320
```

```
taylor(cos(sin(x)),x=0,10);
```

```
                       2      4       6        8
                      x    5 x    37 x    457 x       10
                 1 - -- + ---- - ----- + ------ + O(x  )
                      2     24     720    40320
```

```
taylor(sin(tan(x))-tan(sin(x)),x=0,10);
```

$$-\frac{x^7}{30}-\frac{29\,x^9}{756}+O(x^{10})$$

1.3.6 CAS als Expertensysteme

Bereits die Rechnungen im letzten Abschnitt zeigen, daß CAS neben den Taschenrechnerfunktionen auch eine Quelle umfangreichen mathematischen Wissens in algorithmischer oder assoziativer Form zu den verschiedensten mathematischen Objekten darstellen.

Dies beschränkt sich aber nicht auf die bisherigen klassischen Anwendungen. So kennt MuPAD etwa ebenfalls verschiedene wohlbekannte Summenformeln:

```
sum(i,i=1..n);
```

$$\frac{n}{2}+\frac{n^2}{2}$$

```
Factor(%);
```

$$1/2\ n\ (n+1)$$

```
sum(i^2,i=1..n);
```

$$\frac{n}{6}+\frac{n^2}{2}+\frac{n^3}{3}$$

```
Factor(%);
```

$$1/6\ n\ (n+1)\ (2\,n+1)$$

Es kennt verschiedene Reihen- und Grenzwerte bzw. kann sie berechnen.

```
limit((1 - cos(x))/x^2, x=0);
```

$$1/2$$

```
limit(cos(x)/sin(x)-1/x,x=0);
```

$$0$$

```
sum(1/i^2,i=1..infinity);
```

$$\frac{PI^2}{6}$$

```
limit((1 + 1/n)^n, n = infinity);
```

$$\exp(1)$$

Ausdrücke, die sich nicht elementar vereinfachen lassen, kann ein CAS oft mit anderen mathematischen Begriffen verbinden, die in einer gewissen Begriffswelt zum Standard gehören. Zum Beispiel liefert es für die Partialsummen $H(n) = \sum_{i_1}^{n} \frac{1}{i}$ der harmonischen Reihe

```
sum(1/i,i=1..n);
```

$$\mathtt{EULER} + \mathtt{psi}(n + 1)$$

Hier ist `EULER` die *Eulersche Konstante* $\lim_{n\to\infty}(H(n) - ln(n)) \approx 0.5772156649$) und `psi` eine der vielen speziellen hypergeometrischen Funktionen, die in der Analysis eine wichtige Rolle spielen.

Neben diesen Beispielen algorithmischen Wissens, das in Form von verschiedenen Funktionen implementiert ist, verfügen CAS auch über assoziatives Wissen, d.h. Kenntnisse über einzelne mathematische Symbole. Wir hatten das weiter oben bereits für das Wurzelsymbol und die Sinus-Funktion kennengelernt. Das Wissen über spezielle Werte der Winkelfunktionen übersteigt dabei durchaus an einigen Stellen das aus der Schule bekannte:

```
sin(PI/k) $ k=2..9;
```

$$1, \frac{3^{1/2}}{2}, \frac{2^{1/2}}{2}, \sin\left(\frac{PI}{5}\right), 1/2, \sin\left(\frac{PI}{7}\right), \frac{(-2^{1/2}+2)^{1/2}}{2}, \sin\left(\frac{PI}{9}\right)$$

```
cos(PI/k) $ k=2..9;
```

$$0, 1/2, \frac{2^{1/2}}{2}, \cos\left(\frac{PI}{5}\right), \frac{3^{1/2}}{2}, \cos\left(\frac{PI}{7}\right), \frac{(2+2^{1/2})^{1/2}}{2}, \cos\left(\frac{PI}{9}\right)$$

Funktionsausdrücke, für die keine Vereinfachungen bekannt sind, bleiben als symbolische Ausdrücke stehen.

1.3.7 Rechnen mit Matrizen

Es stehen natürlich auch gängige mathematische Strukturen komplexerer Bauart zur Verfügung. Hier sind vor allem Matrizen und Vektoren zu nennen. In Maple etwa definiert man mit

```
m:=matrix(2,2,[[1,2],[3,4]]);
```

$$m := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

eine Matrix, mit der man dann verschiedene Operationen ausführen kann.

```
evalm(m^2); # Matrixpotenz #
```

$$\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

```
evalm(m^(-1)); # Inverse einer Matrix #
```

$$\begin{bmatrix} -2 & 1 \\ 3/2 & -1/2 \end{bmatrix}$$

```
m2:=matrix(2,2,[[x+1,x-1],[x-1,x+1]]);
```

$$m2 := \begin{bmatrix} x + 1 & x - 1 \\ x - 1 & x + 1 \end{bmatrix}$$

```
evalm(m&*m2); # Matrixprodukt
```

$$\begin{bmatrix} 3\,x - 1 & 3\,x + 1 \\ 7\,x - 1 & 7\,x + 1 \end{bmatrix}$$

```
m3:=evalm(m2^2);
```

$$\begin{bmatrix} (x + 1)^2 + (x - 1)^2 & 2\,(x + 1)\,(x - 1) \\ 2\,(x + 1)\,(x - 1) & (x + 1)^2 + (x - 1)^2 \end{bmatrix}$$

```
map(expand,m3);
```

$$\begin{bmatrix} 2\,x^2 + 2 & 2\,x^2 - 2 \\ 2\,x^2 - 2 & 2\,x^2 + 2 \end{bmatrix}$$

Die Vorgehensweise ist hier schon recht kryptisch, was allerdings damit zusammenhängt, daß die Matrizenmultiplikation nicht kommutativ ist. Die letzte Eingabe zeigt, daß das Simplifizieren der einzelnen Ausdrücke einer Matrix schon detaillierte Kenntnisse über den innerern Aufbau dieser (Maple-spezifischen) Datenstruktur benötigt. In anderen Systemen werden andere Mechanismen zur Matrizendefinition verwendet.

Um z.B. in MuPAD Matrizen zu definieren, muß zunächst ein entsprechender Datentyp vereinbart werden.

```
M:=Dom::Matrix();
```

```
                Dom::Matrix(Dom::ExpressionField(id, iszero))
```

Dies definiert einen Bereich M, dessen Elemente Matrizen mit beliebigen rationalen Ausdrücken als Einträge sein können. Die syntaktische Analogie zu Member-Definitionen für Klassen in C++ ist nicht zufällig. Ungewöhnlich ist nur der Rückgabewert: es wird ein neuer *Datentyp* zurückgegeben. MuPAD behandelt auf diese Weise Datentypen wie andere symbolische Ausdrücke, was es ermöglicht, Funktionen und Datentypkonstruktoren auf eine einheitliche Weise zu definieren. Dies führt zu einem Konzept parametrisierter Datentypen, das weit über die Templates in C++ hinausreicht, was an dieser Stelle aber nicht weiter ausgeführt werden soll.

```
A:=M([[x+1,x-1],[x-1,x+1]]);
```

```
                              +-              -+
                              |  x + 1, x - 1  |
                              |                |
                              |  x - 1, x + 1  |
                              +-              -+
```

```
A1:=M([[a,b],[c,d]]);
```

```
                                  +-      -+
                                  |  a, b  |
                                  |        |
                                  |  c, d  |
                                  +-      -+
```

erzeugt dann Matrizen mit polynomialen Einträgen, mit denen man nun die üblichen Operationen wie Multiplikation oder Inversenberechnung mit den üblichen (objektorientiert überladenen) Operationszeichen ausführen kann.

```
A2:=A*A1; B:=A^2;
```

```
            +-                                                  -+
            |  a (x + 1) + c (x - 1), b (x + 1) + d (x - 1)     |
            |                                                    |
            |  a (x - 1) + c (x + 1), b (x - 1) + d (x + 1)     |
            +-                                                  -+
```

```
              +-                                          -+
              |         2          2                      |
              |  (x + 1)  + (x - 1) ,  2 (x + 1) (x - 1)   |
              |                                            |
              |                              2          2  |
              |   2 (x + 1) (x - 1),  (x + 1)  + (x - 1)   |
              +-                                          -+
```

```
map(B,expand);
```

$$\begin{bmatrix} 2 & 2 \\ 2\,x^2 + 2, & 2\,x^2 - 2 \\ 2 & 2 \\ 2\,x^2 - 2, & 2\,x^2 + 2 \end{bmatrix}$$

```
C:=A^(-1);
```

$$\begin{bmatrix} \dfrac{(x-1)\,(-x+1)}{\left(\dfrac{x-1}{x+1}\right)^2 + 1} & -x+1 \\ \dfrac{x-1}{\left(\dfrac{x-1}{x+1}\right)^2 + 1} & \dfrac{1}{\left(\dfrac{x-1}{x+1}\right)^2 + 1} \end{bmatrix}$$

```
map(C,normal);
```

$$\begin{bmatrix} \dfrac{x+1}{4\,x^2}, & \dfrac{-x+1}{4\,x^2} \\ \dfrac{-x+1}{4\,x^2}, & \dfrac{x+1}{4\,x^2} \end{bmatrix}$$

1.3.8 Der `Solve`-Operator

Gewöhnlich sind die algorithmischen Fähigkeiten zum Auflösen gewisser Sachverhalte in einem oder mehreren `solve`-Operatoren gebündelt, der den Typ eines Problems erkennt und je nachdem verschiedene Lösungsalgorithmen aufruft.

```
solve(x^2+x+1,x);
```

```
                              1/2                 1/2
                  {(- 1/2 I) 3    - 1/2, 1/2 I 3    - 1/2}
```

```
solve({x+y=2,x-y=1},{x,y});
```

```
                             {{y = 1/2, x = 3/2}}
```

```
solve({x^2+y=2,3*x-y^2=2},{x,y});
```

```
{                                    --     y                     1/2       --
{ [x = 1, y = 1], [x = 2, y = -2], |  x = - - 5/3, y = 3/2 I 3    + 1/2  |,
{                                    --     3                              --

--     y                         1/2       -- }
|  x = - - 5/3, y = (- 3/2 I) 3    + 1/2  | }
--     3                                   -- }
```

```
solve(sin(x)=1/2);
```

```
       { / PI  13 PI    11 PI      \  / 5 PI  17 PI    7 PI      \ }
       { | --, -----, - -----, ... |, | ----, -----, - ----, ... | }
       { \ 6     6        6       /  \  6      6       6        / }
```

```
solve(sin(x)+cos(x)=1/2,x);
```

```
{ /       /  1/2       \               /  1/2       \
{ |       | 7          |               | 7          |
{ | 2 atan| ---- + 2/3 |, 2 PI + 2 atan| ---- + 2/3 |,
{ \       \  3         /               \  3         /

                 /  1/2       \      \  /       /    1/2       \
                 | 7          |      |  |       |   7          |
   - 2 PI + 2 atan| ---- + 2/3 |, ... |, | 2 atan| - ---- + 2/3 |,
                 \  3         /      /  \       \    3         /

               /    1/2       \                 /    1/2       \      \ }
               |   7          |                 |   7          |      | }
   2 PI + 2 atan| - ---- + 2/3 |, - 2 PI + 2 atan| - ---- + 2/3 |, ... | }
               \    3         /                 \    3         /      / }
```

Für die letzte Gleichung dürfte es vielen schon schwerfallen, selbst einen Lösungsweg zu finden. Eine mögliche Umformung wäre

$$\sin(x) + \cos(x) = \sin(x) + \sin(\frac{\pi}{2} - x) = 2\sin(\frac{\pi}{4})\cos(x - \frac{\pi}{4}),$$

womit die Gleichung zu $\cos(x - \frac{\pi}{4}) = \frac{1}{2\sqrt{2}}$ umgeformt wurde. Als Ergebnis erhalten wir $\frac{\pi}{4} + \arccos(\frac{1}{2\sqrt{2}}) + 2k\pi$ und $\frac{\pi}{4} - \arccos(\frac{1}{2\sqrt{2}}) + 2k\pi$ mit $k \in \mathbf{Z}$. Ebenso ist es nicht einfach, die Äquivalenz beider Ergebnisse zu prüfen, da MuPAD bei seiner Antwort offensichtlich einen anderen Lösungsweg gegangen ist.

An dieser Stelle wird bereits deutlich, daß die Komplexität eines CAS es oftmals notwendig macht, Ergebnisse in einem gegenüber dem Taschenrechnereinsatz vollkommen neuen Umfang nach- und umzuinterpretieren, um sie an die eigenen Erwartungen an deren Gestalt anzupassen.

1.3.9 Mathematisches Wissen und Pakete

Das mathematische Wissen, das in den verschiedenen CAS enthalten ist, hat inzwischen einen solchen Umfang erreicht, daß es in seiner Gesamtheit nicht unmittelbar im Hauptspeicher gehalten wird. Dies ist auch unter dem Aspekt sinnvoll, daß man für die meisten konkreten Anwendungen nur einen Bruchteil desselben wirklich benötigt. Das Wissen ist deshalb in verschiedenen Paketen abgelegt, die bei Bedarf aktiviert werden können.

So spielen etwa bei der Analyse von Zeitreihendaten die Begriffe *Phase* und *Frequenz* eine wichtige Rolle. Die Transformation von der Zeitskala, in der die Phasenlage sichtbar ist, zur Frequenzskala, die das in der Zeitreihe verborgene Frequenzspektrum darstellt, erfolgt durch die *Fouriertransformation*. Diese kann man ausrechnen, indem man das Paket `transform` lädt. Für klassische Funktionen ergeben sich als Transformation *Dirac*- und davon abgeleitete Funktionen.

```
export(transform);
fourier(exp(-t^2),t,x);
```

$$\mathrm{PI}^{1/2}\,\exp\left(-\frac{x^2}{4}\right)$$

```
fourier(sin(t),t,x);
```

$$I\,\mathrm{PI}\,(-\mathrm{dirac}(x+1)+\mathrm{dirac}(x-1))$$

Andere Pakete existieren für Funktionen aus der Zahlentheorie, der Kombinatorik, der linearen Optimierung und vieler anderer Gebiete der Mathematik.

```
?numlib
Library 'numlib':     the package for elementary number theory
Interface:
numlib::decimal,     numlib::divisors,         numlib::fibonacci,
numlib::fromAscii,   numlib::g_adic,           numlib::ichrem,
numlib::igcdmult,    numlib::ispower,          numlib::isquadres,
numlib::issqr,       numlib::jacobi,           numlib::lambda,
numlib::legendre,    numlib::lincongruence,    numlib::mersenne,
numlib::moebius,     numlib::mroots,           numlib::msqrts,
numlib::numdivisors, numlib::numprimedivisors, numlib::order,
numlib::pollard,     numlib::prevprime,        numlib::primedivisors,
numlib::primroot,    numlib::proveprime,       numlib::sigma,
numlib::sumdivisors, numlib::tau,              numlib::toAscii

?combinat
Library 'combinat':  combinatorial functions
Interface:
combinat::bell,       combinat::cartesian, combinat::composition,
combinat::partitions, combinat::permute,   combinat::powerset,
combinat::stirling1,  combinat::stirling2
```

```
?linopt
Library 'linopt':  Linear and Integer Optimization / Programming
Interface:
linopt::feasible, linopt::findMinimum, linopt::maximize,
linopt::minimize, linopt::setup
```

Dieses sprunghaft wachsende Expertenwissen ist der eigentliche Kern eines CAS als Expertensystem. Man kann mit gutem Gewissen bahaupten, daß heute bereits große Teile der algorithmischen Mathematik in dieser Form verfügbar sind. Mehr noch, kann ein solches Expertensystem auf derselben Basis auch algorithmisches Wissen aus anderen Wissensgebieten erschließen und in seine Informationsbasis aufnehmen. In dieser Richtung spielt das System MATHEMATICA eine Vorreiterrolle, welches insbesondere auf dem Gebiet der Elektrotechnik bereits über eine umfangreiche Bibliothek verfügt.

Die Nutzung dieses Teils der Systeme bleibt allerdings Experten überlassen, wobei neben dem notwendigen eigenen Expertenwissen die Kenntnis wichtiger allgemeiner Arbeitstechniken, um die es in diesem Kurs gehen wird, auch dort eine zentrale Rolle für den Benutzer spielen.

1.3.10 Erweiterung eines CAS

Um dieses Wachstum der Computeralgebra-Systeme zu ermöglichen, besteht (in unterschiedlichem Umfang) die Möglichkeit, diese mit den gegebenen programmiersprachlichen Mitteln zu erweitern oder zu modifizieren, indem eigene Funktionen definiert werden können. Neben detaillierten Kenntnissen dieser Möglichkeiten für Experten, die an solchen Erweiterungen arbeiten, ist das auch für den Nutzer von Interesse, kann er doch auf diese Weise Funktionalität nach eigenem Geschmack hinzufügen.

Betrachten wir etwa die Vereinfachungen, die sich bei der Untersuchung des rationalen Ausdruck u_n ergeben haben. Die dabei entstandenen Polynome, wie übrigens u_n auch, ändern sich bei Vertauschungen der Variablen nicht. Solche Ausdrücke nennt man deshalb *symmetrisch*. Insbesondere spielen symmetrische Polynome in vielen Anwendungen eine wichtige Rolle. Nehmen wir an, daß wir solche symmetrischen Polynome untersuchen wollen, diese Funktionalität aber vom System nicht gegeben wird.

Dazu erst einige Definitionen. Sei $X = (x_1, \ldots, x_n)$ eine endliche Menge von Variablen. Die bekanntesten symmetrischen Polynome sind die *elementarsymmetrischen Polynome* $e_k(X)$, die alle verschiedenen Terme aus k paarweise verschiedenen Faktoren x_i aufsummieren, die *Potenzsummen* $p_k(X) = \sum_i x_i^k$ und die *vollen symmetrischen Polynome* $h_k(X)$, die *alle* Terme in den gegebenen Variablen vom Grad k aufsummieren. So gilt etwa für $n = 4$

$$
\begin{aligned}
e_2 &= x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_3x_4 \\
p_2 &= {x_1}^2 + {x_2}^2 + {x_3}^2 + {x_4}^2 \\
h_2 &= x_1x_2 + x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_3x_4 + {x_1}^2 + {x_2}^2 + {x_3}^2 + {x_4}^2
\end{aligned}
$$

Wir wollen das System MuPAD so erweitern, daß diese Polynome durch die Funktionen `e(d,vars)`, `p(d,vars)` und `h(d,vars)` zu einer natürlichen Zahl d und einer Liste *vars* von Variablen erzeugt werden. Statt der angegebenen Definition wollen wir dazu die rekursiven Relationen

$$e(d, (x_1, \ldots, x_n)) = e(d, (x_2, \ldots, x_n)) + x_1 * e(d-1, (x_2, \ldots, x_n))$$

und

$$h(d, (x_1, \ldots, x_n)) = h(d, (x_2, \ldots, x_n)) + x_1 * h(d-1, (x_1, \ldots, x_n))$$

verwenden. Von der Richtigkeit dieser Formeln überzeugt man sich schnell, wenn man die in der Summe auftretenden Terme in zwei Gruppen einteilt, wobei die erste Gruppe x_1 nicht enthält, die

Teilsumme also gerade das entsprechende symmetrische Polynom in $(x_2, \ldots, x_n)$ ist. In der zweiten Gruppe kann man x_1 ausklammern.

Die entsprechenden Funktionsdefinitionen in MuPAD lauten (ohne Typprüfungen der Eingabeparameter)

```
e:=proc(d,vars) local u; begin
  if nops(vars)<d then 0
  elif d=0 then 1
  else u:=[op(vars,2..nops(vars))];
      expand(e(d,u)+op(vars,1)*e(d-1,u))
end_if end_proc;

h:=proc(d,vars) local u; begin
  if d=0 then 1
  elif nops(vars)=1 then op(vars,1)^d
  else u:=[op(vars,2..nops(vars))];
      expand(h(d,u)+op(vars,1)*h(d-1,vars))
end_if end_proc;

p:=proc(d,vars) begin
  _plus(map(op(vars),func(x^d,x)))
end_proc;
```

Wir sehen an diesem kleinen Beispiel bereits, daß die komplexen Datenstrukturen, die in symbolischen Rechnungen auf natürliche Weise auftreten, den Einsatz verschiedener Programmierparadigmen und -praktiken ermöglichen.

Die ersten beiden Blöcke ergänzen die jeweilige Rekursionsrelation durch geeignete Abbruchbedingungen zu einer korrekten Funktionsdefinition für e_d und h_d. Im dritten Block werden intensiv verschiedene Operationen auf Listen in einem funktionalen Programmierstil verwendet. Für

```
vars:=[x,y,z];
```

erhalten wir damit

```
h(2,vars);
```

$$x\,y + x\,z + y\,z + x^2 + y^2 + z^2$$

```
e(3,vars);
```

$$x\,y\,z$$

```
h(3,vars);
```

$$x\,y\,z + x^3 + y^3 + z^3 + x^2\,y + x\,y^2 + x^2\,z + x\,z^2 + y^2\,z + y\,z^2$$

Ein Vergleich mit unseren weiter oben vorgenommenen Vereinfachungen zeigt, daß die rationale Funktion u_d offensichtlich gerade mit dem vollen symmetrischen Polynom h_{d-2} zusammenfällt.

Kapitel 2

Die Stellung der Computeralgebra im Wissenschaftsgebäude

2.1 Symbolisches Rechnen als Kulturtechnik

Mathematik besteht zu 90 % aus Rechnen

Mathematik ist ebenso wie die anderen Naturwissenschaften eine sehr experimentelle Wissenschaft, wenn auch ihre Experimente mit "Bleistift und Papier" von besonderer Art sind. Jedoch auch für sie gilt, daß nur das Aufspüren von Gesetzmäßigkeiten in einer Fülle empirischen Materials den Weg zu neuen Einsichten weist. Es kann deshalb nicht verwundern, daß die überwiegende Zahl mathematischer Problemlösungen von aufwendigen Berechnungen begleitet sind, die nur selten darin gipfeln, daß sich neue Erkenntnisse in Form von mathematischen Sätzen formulieren (und beweisen) lassen.

Mehr noch, auch in anderen Wissenschaftszweigen und ingenieurtechnischen Anwendungen, die quantisierbare Effekte untersuchen, tritt die Mathematik als Werkzeug in der Regel eher in ebendieser unspektakulären, alltäglichen Weise in Erscheinung, statt durch unerwartete Vereinfachungen und zwingend logische Schlußketten zu glänzen.

Berühmte Mathematiker, ob nun Euler, Gauß, Fermat, die Gebrüder Bernoulli oder andere, waren nicht nur geniale Zeitgenossen, sondern stets auch hervorragende Rechenkünstler. Mittelalterliche Rechenmeister wie Adam Ries genossen ob ihres Wissens und ihrer Fertigkeiten großes Ansehen, obwohl man sie kaum als (bedeutende) Mathematiker bezeichnen kann. Eine Verschiebung dieser Situation hin zu einer eher "trockenen" Mathematik, wie sie sich heute vielen Menschen darstellt, tritt erst mit Beginn dieses Jahrhunderts ein, als stärker qualitative Aussagen in den Vordergrund rückten. Damit sind mathematische Existenzbeweise gemeint, die kein Verfahren liefern, die Größen, deren Existenz behauptet wird, auch zu konstruieren. Man begann also, über Dinge zu reden, deren Existenz wohl klar war, aber die man nicht in der Hand hielt. Ausgelöst wurde dieser Wechsel durch die zunehmende Schwierigkeit, immer komplexere Rechnungen selbst an einfachen Beispielen auch wirklich auszuführen. In einer ersten Etappe gab man sich mit Beweisen zufrieden, wenn die Konstruktionen als prinzipiell ausführbar dargestellt werden konnten, genügend Fertigkeiten im Umgang mit elementaren mathematischen Objekten vorausgesetzt. In einer zweiten Etappe wurden auch reine Existenzbeweise, wenn auch zähneknirschend, akzeptiert. So ist von Paul Gordan, dem damaligen Papst der Invariantentheorie, auf Hilberts (Existenz-)Beweis der Endlichkeit der Basisinvarianten die abschätzige Bemerkung: "Das ist Teufelszeug, aber kein Beweis" überliefert.

Im Zusammenhang mit der hierdurch ausgelösten Krise der Mathematik wurden generelle Fragen der Beweisbarkeit und Konstruierbarkeit näher untersucht. Heute kennt man die Antwort der Logiker, daß es wahre, aber prinzipiell unbeweisbare Tatsachen ebenso gibt wie beweisbare, aber

prinzipiell nicht konstruierbare Objekte.

Für uns ist jedoch interessant, daß mit dem Aufkommen des Computers die Grenze der Ausführbarkeit mathematischer Konstruktionen weit verschoben wurde, so daß seither auch konstruktive Aspekte der Mathematik wieder einen Aufschwung erlebt haben. Welche Möglichkeiten in dieser Richtung erst Systeme bieten, die bereits über umfangreiche "Fertigkeiten im Umgang mit elementaren mathematischen Objekten" verfügen, kann man nach den einführenden Demonstrationen wohl ermessen.

Mathematik als Element der Allgemeinbildung

Für den Einsatz der Kulturtechnik "Mathematik" in einem Umfang, wie ihn der heutige Entwicklungsstand der menschlichen Gemeinschaft zwingend erfordert, ist es notwendig, diese breiten Kreisen zu erschließen. Nachdem noch vor 500 Jahren selbst elementare Rechenfertigkeiten nicht zur Allgemeinbildung gehörten, kann man seitdem auf eine rasante Entwicklung blicken. Während damals breite Kreise der Bevölkerung als Analphabeten weden lesen noch schreiben konnten, ist heute ein erfolgreicher Schulabschluß nicht möglich, wenn man nicht auch über Grundkenntnisse einiger wichtiger mathematischer Arbeitsmethoden verfügt. Dazwischen liegen 400 Jahre, die mit der Erfindung des Buchdrucks und den ingenieurtechnischen Innovationen der industriellen Revolution dazu führten, daß heute aktive Teilhabe am gesellschaftlichen Leben ohne solche Grundkenntnisse einfach nicht mehr möglich ist. Im übrigen ist mit dem Übergang zu stärker wissensbasierten Produktionsmethoden, dessen Zeuge wir gerade sind, noch einmal mit einer wesentlichen Steigerung dieser Anforderungen zu rechnen.

Symbolisches Rechnen als die höhere Form der Mathematik

Grundstock dieser mathematischen Fertigkeiten sind zuerst einmal *Rechenfertigkeiten*, wie sie Rechenmeister wie Adam Ries verbreiteten und wie sie heute bereits in der Grundschule vermittelt werden. Solche arithmetischen Fähigkeiten sind für uns heute selbstverständlich. Allerdings öffnet erst der souveräne Gebrauch dieser Methoden *direkter* Berechnung, d.h. eine befriedigende Antwort auf die Frage nach Effekten, die von gewissen Ursachen hervorgebracht werden, die Perspektive für *inverse* Fragestellungen, die mit der gezielten Manipulation von Ursachen, um gewünschte Effekte hervorzubringen, verbunden sind. In dieser generellen, paradigmatischen Fragestellung, die eine skalierbare Vorwegnahme des gewünschten Ergebnisses als zentrale Komponente erfordert, finden wir auch die Wurzeln des symbolischen Rechnens.

So kann man etwa, die numerische Fähigkeit zum Berechnen von Nullstellen eines Polynoms, wie sie heute selbst Taschenrechner besitzen, vorausgesetzt, eine große Menge von Daten erzeugen, um eine Beziehung zwischen den Nullstellen und den Koeffizienten eines Polynoms aufzuspüren, indem wir diese Koeffizienten jeweils mit der Summe und dem Produkt der Nullstellen vergleichen. Simulieren wir diese Rechnungen einmal mit Maple:

```
print('Nullstellen', 'a','b','Summe','Produkt');
for a from 2.0 to 4.0 do for b from -12.0 to -9.0 do
     u:=solve(x^2+a*x+b=0,x);
     print([u], a,b, u[1]+u[2], u[1]*u[2])
od od;
```

```
    Nullstellen,          a,    b,       Summe,       Produkt

   [-4.605551275, 2.605551275], 2.0, -12.0, -2.000000000, -12.00000000
   [-4.464101615, 2.464101615], 2.0, -11.0, -2.000000000, -11.00000000
   [-4.316624790, 2.316624790], 2.0, -10.0, -2.000000000, -9.999999998
   [-4.162277660, 2.162277660], 2.0,  -9.0, -2.000000000, -8.999999999
```

```
   [-5.274917218, 2.274917218], 3.0, -12.0, -3.000000000, -12.00000000
   [-5.140054945, 2.140054945], 3.0, -11.0, -3.000000000, -11.00000000
            [-5., 2.],          3.0, -10.0, -3.,          -10.
   [-4.854101966, 1.854101966], 3.0,  -9.0, -3.000000000, -8.999999998
            [-6., 2.],          4.0, -12.0, -4.,          -12.
   [-5.872983346, 1.872983346], 4.0, -11.0, -4.000000000, -11.00000000
   [-5.741657387, 1.741657387], 4.0, -10.0, -4.000000000, -10.00000000
   [-5.605551276, 1.605551276], 4.0,  -9.0, -4.000000000, -9.000000004
```

Wir erkennen unschwer die aus dem Vietaschen Wurzelsatz bekannte Relation, daß für die Nullstellen x_1 und x_2 gerade $a = -(x_1 + x_2)$ und $b = x_1\,x_2$ gilt. Allerdings ist eine solche Liste von experimentellen Ergebnissen, so lang sie auch sein mag, noch kein Beweis im streng deduktiven Sinn. Diesen wird man für den Fall eines Polynoms zweiten Grades erst erbringen können, wenn man in der Lage ist, die Nullstellen der *allgemeinen Gleichung 2. Grades*, also von

$$x^2 + a\,x + b = 0,$$

symbolisch durch die Koeffizienten a, b darzustellen, also die allgemeine Lösungsformel

$$x_1 = -1/2\,a + 1/2\,\sqrt{a^2 - 4\,b}, \quad x_2 = -1/2\,a - 1/2\,\sqrt{a^2 - 4\,b}$$

kennt. Aus dieser kann man unschwer einen *Beweis* für den behaupteten Zusammenhang herleiten. Mehr noch sind die symbolischen Rechnungen mit einem der Formelmanipulation fähigen System auch *automatisch* ausführbar, wie in unserem Beispiel mit Maple:

```
a:='a': b:='b': u:=solve(x^2+a*x+b=0,x);

                              2       1/2                   2       1/2
        u := - 1/2 a + 1/2 (a  - 4 b)   , - 1/2 a - 1/2 (a  - 4 b)

u[1]+u[2];
                                     -a

expand(u[1]*u[2]);
                                      b
```

Dieselben Untersuchungen für ein Polynom dritten Grades lassen einen ähnlichen Zusammenhang vermuten. Die entsprechenden numerischen Rechnungen sind wieder mit Maple simuliert, diesmal aber auf drei Stellen gekürzt:

```
Digits:=3;
print('Nullstellen', 'a','b','c','Summe','Produkt');
for a from 2.0 to 3.0 do
   for b from -12.0 to -11.0 do
   for c from 3.0 to 5.0 do
     u:=solve(x^3+a*x^2+b*x+c=0,x);
     print([u], a,b,c, u[1]+u[2]+u[3], u[1]*u[2]*u[3] )
od od od;

        Nullstellen,       a,     b,   c,  Summe, Produkt

     [-4.69, .263, 2.43], 2.0, -12.0, 3.0, -2.00, -2.99
     [-4.72, .359, 2.36], 2.0, -12.0, 4.0, -2.00, -3.99
```

```
[-4.75, .460, 2.29], 2.0, -12.0, 5.0, -2.00, -5.02
[-4.56, .290, 2.27], 2.0, -11.0, 3.0, -2.00, -3.00
[-4.59, .398, 2.19], 2.0, -11.0, 4.0, -2.00, -4.01
[-4.62, .515, 2.10], 2.0, -11.0, 5.0, -2.01, -5.00
[-5.35, .270, 2.08], 3.0, -12.0, 3.0, -3.00, -3.00
  [2., -5.37, .373], 3.0, -12.0, 4.0, -3.00, -3.99
[-5.40, .485, 1.91], 3.0, -12.0, 5.0, -3.01, -5.00
[-5.22, .300, 1.92], 3.0, -11.0, 3.0, -3.00, -3.01
[-5.24, .418, 1.83], 3.0, -11.0, 4.0, -2.99, -4.01
[-5.27, .554, 1.71], 3.0, -11.0, 5.0, -3.01, -4.99
```

Hier ist der Zusammenhang zur expliziten Lösungsformel zum Beweis der angetroffenen Regelmäßigkeit bereits schwieriger auszunutzen. Für Polynome höheren Grades, wo es solche Lösungsformeln gar nicht gibt, muß man schließlich tieferliegende Zusammenhänge zwischen den Nullstellen und den Koeffizienten benutzen, die sich *nur* symbolisch formulieren lassen: Ein Polynom

$$f := x^n + a_1\, x^{n-1} + a_2\, x^{n-2} + \ldots + a_{n-1}\, x + a_n$$

hat stets n komplexe Nullstellen $x_1, x_2, \ldots, x_n$, wenn man diese mit der entsprechenden Vielfachheit zählt, und es gilt

$$f = (x - x_1) \cdot (x - x_2) \cdot \ldots \cdot (x - x_n).$$

Nun ist es leicht, die Vietaschen Formeln allgemein herzuleiten. Man muß einfach diese Produktdarstellung ausmultiplizieren und die Koeffizienten mit denen in der Ausgangsdarstellung vergleichen.

Satz 2 *(Vietascher Wurzelsatz)*
Zwischen den Koeffizienten des Polynoms

$$f := x^n + a_1\, x^{n-1} + a_2\, x^{n-2} + \ldots + a_{n-1}\, x + a_n$$

und seinen Nullstellen $x_1, x_2, \ldots, x_n$ besteht folgender Zusammenhang:

$$a_i = (-1)^i\, e_i(x_1, x_2, \ldots, x_n), \quad i = 1, 2, \ldots, n,$$

wobei e_i das im letzten Kapitel eingeführte i-te elementarsymmetrische Polynom ist.

Evidenz für diesen Sachverhalt kann man sich wiederum mit dem Computer verschaffen, diesmal unter Ausnutzung der *symbolischen* Fähigkeiten von Maple:

```
for n from 2 to 5 do print(collect(expand(mul(x-x.i,i=1..n)),x)) od;
                            2
                           x  + (-x2 - x1) x + x1 x2

         3                     2
        x  + (-x1 - x3 - x2) x  + (x1 x3 + x1 x2 + x2 x3) x - x1 x2 x3

 4                         3
x  + (-x2 - x1 - x4 - x3) x

                                                              2
     + (x1 x2 + x2 x4 + x2 x3 + x1 x4 + x1 x3 + x3 x4) x

     + (-x1 x2 x4 - x1 x2 x3 - x2 x3 x4 - x1 x3 x4) x + x1 x2 x3 x4
```

```
 5                                4
x  + (-x2 - x5 - x4 - x3 - x1) x  + (x1 x5 + x3 x5 + x2 x5 + x3 x4 + x2 x3

                                                    3
     + x1 x4 + x1 x3 + x1 x2 + x2 x4 + x4 x5) x  + (-x1 x2 x5 - x2 x3 x4

     - x1 x3 x4 - x1 x2 x4 - x3 x4 x5 - x1 x4 x5 - x1 x3 x5 - x2 x4 x5

                              2
     - x2 x3 x5 - x1 x2 x3) x

     + (x1 x3 x4 x5 + x1 x2 x4 x5 + x1 x2 x3 x5 + x1 x2 x3 x4 + x2 x3 x4 x5) x

     - x1 x2 x3 x4 x5
```

Zum *Beweis* dieses Satzes ist eine weitere Abstraktionsstufe zu erklimmen, indem man etwa die zur Definition der elementarsymmetrischen Polynome im letzten Kapitel herangezogenen Rekursionsrelationen verwendet.

Numerische Ergebnisse ermöglichen es, größere Datenmengen zu erzeugen, die man nach *Regelmäßigkeiten* untersuchen kann. Jedoch erst eine symbolische Analyse in einem streng deduktiven Verständnis von Mathematik verwandelt diese in *Gesetzmäßigkeiten*, die unserem *Pool gesicherten Wissens* hinzugefügt werden können.

Symbolische Ergebnisse vermitteln also einen wesentlich tieferen Einblick in Zusammenhänge als arithmetische, erfordern aber zugleich einen komplizierteren mathematischen Apparat. Sie sind jedoch das eigentliche Ziel matehmatischer Erkenntnis. [13] zitiert dafür als Beleg R.W.HAMMING mit dem Ausspruch:

The purpose of computing is insight, not numbers.

Dies gilt insbesondere für Prozesse, die nicht nur *analysiert*, sondern auch *gesteuert* werden sollen.

Symbolisches Rechnen und der Computer als Universalmaschine

Interessanterweise wiederholt sich diese Entwicklung von der Arithmetik hin zur Symbolik in der Geschichte des Einsatzes des Computers als Hilfsmittel geistiger Arbeit. Historisch wurde das Wort Computer bekanntlich zuerst mit einer Maschine zur schnellen Ausführung numerischer Rechnungen verbunden. Nachdem dies in der Anfangszeit ebenfalls auf einfache arithmetische Operationen beschränkt war (und für Taschenrechner lange so beschränkt blieb), können auf entsprechend leistungsfähigen Maschinen heute auch kompliziertere Anwendungen wie das Berechnen numerischer Werte mathematischer Funktionen, die Approximation von Nullstellen gegebener Polynome oder von Eigenwerten gegebener Matrizen realisiert werden.

All diesen Anwendungen ist gemein, daß sie sich, unter Verwendung ausgefeilter Programmiersprachen die Programmierfähigkeit eines Computers ausnutzend, letztlich allein auf das Rechnen mit (Computer)zahlen zurückführen lassen. Sowohl von der Aufgabenstellung her als auch der Methodik wird dabei mit Näherungswerten gerechnet und die Frage der Relevanz der erzielten Ergebnisse (Fehlertoleranzen) ist gesondert zu untersuchen. Solche *numerischen Anwendungen*, insbesondere aus der Hochenergiephysik, der Quantenchemie und anderen stark mathematisierten Gebieten der Naturwissenschaften sind auch heute noch der Hauptposten in der Auslastung der "mainframes", großer und moderner Rechensysteme.

Mehr noch verbindet heute ein weiter Kreis von Anwendern mit dem Terminus "wissenschaftliches Rechnen" im Zusammenhang mit Computern gerade einen solchen Einsatz, obwohl dies sowohl aus innermathematischen, wie oben dargelegt, als auch informatik-theoretischen Überlegungen

heraus eher einer künstlichen Beschränkung der Einsatzmöglichkeiten eines modernen Computers gleichkommt. Zeigt uns doch die Berechenbarkeitstheorie in Gestalt der Church'schen These, daß der Computer eine *Universalmaschine* ist, die, mit einem geeigneten Programm versehen, prinzipiell in der Lage versetzt werden kann, jede nur denkbare algorithmische Tätigkeit auszuüben. Also insbesondere auch in der Lage sein sollte, Symbole und nicht nur Zahlen nach wohlbestimmten Regeln zu verarbeiten.

Diese Eigenschaft war lange vor dem Bau des ersten "echten" (von-Neumann-)Rechners bekannt, rückte aber durch die Spezifik bisheriger Einsatzgebiete in den Hintergrund. Bereits Charles Babbage (1792 - 1838), der mit seiner "Analytical Engine" 1838 ein dem heutigen Computer ähnliches Konzept entwickelte, ohne es aber je realisieren zu können, hatte die Fähigkeiten einer solchen Maschine zur symbolischen Informationsverarbeitung im Blick. Seine Assistentin, Freundin und Mäzenin, Lady Lovelace, schreibt (zitiert nach [10, S. 1]) :

> Viele Menschen, die nicht mit entsprechenden mathematischen Studien vertraut sind, glauben, daß mit dem *Ziel* von Babbage's Analytical Engine, Ergebnisse in Zahlennotation auszugeben, auch deren *Inneres* arithmetisch-numerischer Natur sein müsse statt algebraisch-analytischer. Das ist ein Irrtum. Die Maschine kann ihre numerischen Eingaben genau so anordnen und kombinieren, als wären es Buchstaben oder andere allgemeine Symbole; und könnte sie sogar *in einer solchen Form ausgeben*, wenn nur entsprechende Vorkehrungen getroffen würden.

Ein solches Computerverständnis ist uns, im Gegensatz zu den Pionieren des Computer-Zeitalters, im Lichte von ASCII-Code und Textverarbeitungssystemen heute allgemein geläufig, wenigstens was den Computer als intelligente Schreibmaschine betrifft. Mit dem Siegeszug der Kleinrechentechnik in den letzten 20 Jahren entwickelte er sich dabei vom Spielzeug und der erweiterten Schreibmaschine hin zu einem unentbehrlichen Werkzeug der Büroorganisation, wobei vor allem seine Fähigkeit, (geschriebene) Information speichern, umordnen und verändern zu können, eine zentrale Rolle spielt. In diesem Anwendungssektor kommt also die Fähigkeit des Computers, *symbolische Information* verarbeiten zu können, bereits unmittelbar auch für den Umgang mit Daten zum Einsatz. Dabei verwischt sich, genau wie von Lady Lovelace vorausgesehen, durch die binäre Kodierung symbolischer Information die Grenze zwischen Zahlen und Zeichen, die zuerst so absolut schien.

Auf dieser Abstraktionsebene ist es auch möglich, die verschiedensten Nachschlagewerke und Formelsammlungen, also in symbolischer Form kodiertes Wissen, mit dem Computer aufzubereiten, in datenbankähnlichen Strukturen vorzuhalten und mit entsprechenden Textanalyseinstrumenten zu erschließen. Es wird sogar möglich, auf verschiedene Weise symbolisch kodierte Informationen in multimedialen Produkten zu verknüpfen, was das ungeheure innovative Potential dieser Entwicklungen verdeutlicht. In der Hand des Ingenieurs und Wissenschaftlers entwickelt sich damit der Computer zu einem sehr effektiven Instrument, das nicht nur den Rechenschieber, sondern auch zunehmend Formelsammlungen abzulösen in der Lage ist. Auf diesem Niveau handelt es sich allerdings noch immer um eine *syntaktische* Verarbeitung von Information, wo der Computer deren *Sinn* noch nicht in die Verarbeitung einzubeziehen vermag.

Aber auch der Einsatz des Computers zu *numerischen* Zwecken enthält bereits eine wichtige symbolische Komponente: Er hat das Programm für die Rechnungen in adäquater Form in seinen Speicher zu bringen und von dort wieder zu extrahieren. Diese Art symbolischer Information ist bereits *semantischer Art*, da die auf diese Weise dargestellten *Algorithmen* inhaltliche Aspekte der verarbeiteten Zahlengrößen erschließen. Daß dies vom Nutzer nicht in gebührender Form wahrgenommen wird, hängt in erster Linie mit der strikten Trennung von (numerischen) Daten und (symbolischem) Programm sowie der Betrachtung des Computers als virtuelle Maschine ("Das Programm macht der Programmierer, die Rechnung der Computer") zusammen.

Bringt man beide Ebenen, die Daten und die Programme, zusammen, ermöglicht also algorithmische Operationen auch auf symbolischen Daten, geht man einen großen Schritt in die Richtung,

auch semantische Aspekte der Information zu berücksichtigen. In diesem Schnittpunkt moderner Entwicklungen im Multimediabereich und einer Erweiterung des klassischen Selbstverständnisses des Wissenschaftlichen Rechnens steht heute die *Computeralgebra*. Obwohl sie ob ihrer *Inhalte* traditionell stärker dem klassischen Gegenstand des wissenschaftlichen Rechnen nahesteht und lange nur als Verlängerung dieser sich in jahrzehntelanger Entwicklung aus der Numerik heraus etablierten wissenschaftlichen Disziplin verstanden wurde, wird heute die Vereinigung algorithmischer Methoden mit Wissensrepräsentationssystemen zu einer *Computermathematik* als neuer Inhalt wissenschaftlichen Rechnens thematisiert, wie etwa in [6], wo sogar von der Computeralgebra als "einer Säule des wissenschaftlichen Rechnens" gesprochen wird.

2.2 Was ist Computeralgebra ?

Was ist nun Computeralgebra ? Mathematische Berechnungen haben gewöhnlich neben der rein numerischen Komponente eine weitere wichtige Seite, die wir *symbolisch-algebraische Berechnungen* nennen wollen. Die Spezifik eines solchen Zugangs demonstriert das folgende Beispiel aus [13]:

> Betrachten wir den einfachen Ausdruck $\frac{3\pi^2}{\pi}$. Jeder Algebrastudent weiß, daß sich dieser Bruch zu 3π vereinfachen läßt, wenn man Zähler und Nenner durch π kürzt. Der numerische Wert von 3π kann interessieren, es kann aber auch sein, daß es ausreicht oder vielleicht sogar günstiger ist, diesen Ausdruck in seiner symbolischen, nichtnumerischen Form zu belassen. Mit einem nur auf arithmetische Operationen getrimmten Computer *muß* der Ausdruck $\frac{3\pi^2}{\pi}$ ausgewertet werden; wird dies mit einer Präzision von 10 Stellen ausgeführt, erhält man als Wert 9.424777958. Diese Zahl, abgesehen von der Tatsache, daß es sich dabei um eine eher uninformative Folge von Ziffern handelt, ist nicht dasselbe wie die Zahl, die man auf demselben Wege (d.h. durch Auswertung mit einer Genauigkeit von 10 Stellen) aus 3π bekommt. Letzteres liefert nämlich die Zahl 9.424777962, wobei die Differenz in den letzten beiden Stellen aus unumgänglichen Rundungsfehlern des Computers herrührt. Die Äquivalenz von $\frac{3\pi^2}{\pi}$ und 3π würde von solch einem Computer wohl nicht einmal festgestellt werden.

Im weiteren verweisen die Autoren von [13] auf die folgenden Vorteile, die eine solche *strukturelle* Sicht auf mathematische Untersuchungen gegenüber rein quantitativen Zugängen bietet:

> *Erstens* gestattet eine solche präventive symbolische Analyse es oftmals, einen Ausdruck effektiver numerisch auszuwerten.
>
> *Zweitens* sind die gewonnenen Aussagen durch das prinzipielle Vermeiden von numerischen Approximationen exakt in einem streng mathematisch-deduktiven Sinn. Damit werden Fehler- und Rundungsanalysen, die bei der Ergebnisverifikation numerischer Verfahren oftmals den Löwenanteil des Aufwands ausmachen, unnötig.

Der wohl zentrale Vorteil liegt jedoch darin, daß, wie im Abschnitt 2.1 bereits ausgeführt,

> *drittens* Ergebnisse in einer solchen algebraischen Form einer höheren Form mathematischen Verständnisses entsprechen als selbst eine große Menge rein numerischer Daten.

Es geht dabei wie in der Textverarbeitung um die Umformung symbolischer Information nach mehr oder minder komplizierten Regeln, aber unter Einbeziehung eines gewissen Verständnisses der damit verbundenen Semantik. Kurz gesprochen kann man dieses Gebiet also als

> *das Rechnen mit Symbolen, die mathematische Objekte repräsentieren,*

bezeichnen. Diese Objekte können neben ganzen, rationalen, reellen oder komplexen Zahlen (beliebiger Genauigkeit) auch algebraische Ausdrücke, Polynome, rationale Funktionen, Gleichungssysteme

oder sogar noch abstraktere mathematische Objekte wie Gruppen, Ringe, Algebren und deren Elemente sein. Mehr noch, das Adjektiv *symbolisch* bedeutet dabei, daß das Ziel der mathematischen Problemstellung nicht die Suche nach numerischen Werten, sondern nach einer geschlossenen oder approximativen Formel im Sinne des deduktiven Mathematikverständnisses ist. *Algebraisch* bedeutet, daß eine *exakte* mathematische Ableitung aus den Ausgangsgrößen durchgeführt wird, anstatt näherungsweise Fließkommaarithmetik einzusetzen[1]. Beispiele für solche algebraisch-symbolischen Umformungen sind die Polynomfaktorisierung, die (unbestimmte) Differentiation und Integration, die Reihenentwicklung von Funktionen, analytische Lösungen von Differentialgleichungen, exakte Lösungen polynomialer Gleichungssysteme oder die Simplifikation mathematischer Formeln.

In den letzten 30 Jahren hat man große Fortschritte in der theoretischen Fundierung dieser symbolischen Methoden erreicht. Es sind Werkzeuge entstanden, die es erlauben, ein großes Stück mathematischen und zunehmend auch naturwissenschaftlichen und ingenieurtechnischen Know-hows Anwendern als black box unmittelbar zugänglich zu machen. Diese junge, sich stark entwickelnde Disziplin im Grenzgebiet zwischen Mathematik und Informatik hat noch keinen traditionellen Namen: symbolisches Rechnen, Formelmanipulation oder Computeralgebra sind einige im deutschen Sprachraum gängige Bezeichnungen. Wir wollen im folgenden zur Bezeichnung dieses Gebiets die letztere bemühen.

Historisch stand und steht dabei das effiziente Rechnen in durch Erzeugende und Relationen gegebenen algebraischen Strukturen am Ausgangspunkt, d.h. polynomiale und rationale Ausdrücke in einer Menge mathematischer Symbole, wobei die wichtigste Problemstellung im Erkennen semantischer Gleichwertigkeit syntaktisch unterschiedlicher Ausdrücke liegt. Die hierbei verwendeten Methoden unterscheiden sich dabei oftmals sehr von der durch "mathematische Intuition" gelenkten und stärker heuristisch geprägten Schlußweise des Menschen und greifen damit auch verstärkt die Entwicklungslinien der konstruktiven Mathematik mit ihrer vorerst letzten Blüte in den 20er Jahren dieses Jahrhunderts wieder auf, vgl. R.Loos in [2].

In diesem Sinne beschreibt J.Grabmeier in [6], auf R.Loos zurückgehend,

> *Computeralgebra als den Teil der Informatik und Mathematik, der algebraische Algorithmen entwickelt, analysiert, implementiert und anwendet.*

Einzubeziehen sind dabei neben analytischen Algorithmen, insofern sie auf *algebraische* Umformungen hinauslaufen, auch

> *die Entwicklung des zu Implementierung und Management solcher Systeme notwendigen informatik-theoretischen Instrumentariums.*

Die Computeralgebra befindet sich damit an der Schnittstelle zentraler Entwicklungen verschiedener Gebiete sowohl der Mathematik als auch der Informatik.

In ihrem Computeralgebra-Report [7] definiert die Fachgruppe Computeralgebra der Gesellschaft für Informatik (GI) ihr eigenes Fachgebiet etwas ausführlicher so:

> *Die Computeralgebra ist ein Wissenschaftsgebiet, das sich mit Methoden zum Lösen mathematisch formulierter Probleme durch symbolische Algorithmen und deren Umsetzung in Soft- und Hardware beschäftigt. Sie beruht auf der exakten endlichen Darstellung endlicher oder unendlicher mathematischer Objekte und Strukturen und ermöglicht deren symbolische und formelmäßige Behandlung durch eine Maschine. Strukturelles mathematisches Wissen wird dabei sowohl beim Entwurf als auch bei der Verifikation und Aufwandsanalyse der betreffenden Algorithmen verwendet. Die Computeralgebra kann damit*

[1] Dies gilt auch für Anwendungen aus der Analysis, die, wie z.B. Grenzwert- oder Integralbegriff, per definitionem Näherungsprozesse untersuchen. In der Tat zeigt der Unterschied etwa zwischen der Ableitungsdefinition und dem Vorgehen bei der praktischen Bestimmung einer solchen Ableitung bereits den ebenfalls *algebraischen* Charakter der verwendeten Vorgehensweise, wo aus einem kleinen Arsenal von Ableitungen elementarer Funktionen nach *algebraischen* Regeln die einer daraus zusammengesetzten Funktion *deduziert* wird. Vgl. auch R.Loos in [2]

wirkungsvoll eingesetzt werden bei der Lösung von mathematisch modellierten Fragestellungen in zum Teil sehr verschiedenen Gebieten der Informatik und Mathematik sowie in den Natur- und Ingenieurwissenschaften.

In diesem Spannungsfeld zwischen Mathematik und Informatik, das R.Loos in [2] etwas pessimistisch als einen Bereich umreißt, der solche Probleme zum Gegenstand hat,

> die zu rechenintensiv sind, um in einem Algebrabeitrag zu erscheinen
>
> und die zu algebraisch sind, um sie in einem Informatikbuch darzulegen,

findet die Computeralgebra zunehmend ihren eigenen Platz. Sie nimmt damit zugleich eine wichtige Stellung für Entwicklungsimpulse aus beiden Gebieten ein. So mag es nicht verwundern, daß große Durchbrüche der letzten Jahre sowohl in der Mathematik als auch in der Informatik die von der Computeralgebra produzierten Werkzeuge wesentlich beeinflußt haben und umgekehrt.

Hier kommt zugleich ein weiterer Vorteil des Computereinsatzes beim symbolischen Rechnen zum Ausdruck, der starken Einfluß auf pragmatische Aspekte der Akkumulation wissenschaftlicher Kenntnisse hat. In [13] wird er wie folgt beschrieben:

> Eine wissenschaftliche Theorie wird oftmals in einer kompakten und sehr allgemeinen Sprache formuliert, die die Form, die diese allgemeine Theorie unter bestimmten Voraussetzungen annimmt, anschaulich zu suggerieren vermag. Z.B. kann ein Teil der allgemeinen Relativitätstheorie, bekannt als die *Einsteinsche Feldgleichung*, in der einzigen Zeile
>
> $$R_{\mu\nu} - \frac{1}{2} g_{\mu\nu} R = -\kappa T_{\mu\nu} \quad \text{mit} \quad R = g^{\mu\nu} R_{\mu\nu}$$
>
> aufgeschrieben werden, gewisse weithin akzeptierte Bezeichnungen als bekannt vorausgesetzt. Die physikalischen Folgerungen dieser Theorie zu erforschen bedeutet, die mathematischen Konsequenzen dieser einen Gleichung zu untersuchen. Diese Untersuchungen erfordern aber, von einigen einfachen Fällen abgesehen, eine solch ungeheure Anzahl von algebraischen Operationen, daß dafür Handrechnungen mit Bleistift und Papier ausscheiden. Selbst unter solchen Annahmen, die das Problem enorm vereinfachen, ist die Algebra so anspruchsvoll, daß nur wenige Forscher bereit sind, ihre Gesundheit, ihren Optimismus und ihr berufliches Fortkommen bei einem Lösungsversuch riskieren[2].
>
> Selbst wenn ein solches Ergebnis dann in den Kreis der gesicherten wissenschaftlichen Erkenntnis aufgenommen werden sollte, steht es doch isoliert auf einem intellektuellen Außenposten, nicht überprüft und nicht überprüfbar durch andere Wissenschaftler und akzeptiert hauptsächlich wegen der Reputation des Forschers und weniger, weil es weiterer kritischer Examination standgehalten hat.

Der Einsatz des Computers erlaubt es nun, hier zu vollkommen neuen Ufern aufzubrechen. Eines der in diesem Zusammenhang oft genannten Beispiele der "Macht der Agentien" (Marx), die eine solche Entwicklung konstituiert, ist die Wiederholung der Mondbahnberechnungen, für die der französische Astronom C.Delaunay im 19. Jahrhundert 20 Jahre seines Lebens opferte und die mit modernen Computern in wenigen Stunden nachvollziehbar sind, vgl. [13]. Die mathematische Glaubwürdigkeit des Computerergebnisses ist zudem unvergleichlich höher, wie in der anerkennenden Feststellung zum Ausdruck kommt, daß sich Delaunay nur an drei Stellen in Termen untergeordneter Bedeutung verrechnet habe.

Die von der Computeralgebra produzierten Werkzeuge spielen eine stark zunehmende Rolle sowohl in den Natur- als auch in den Ingenieurwissenschaften. Dies dokumentiert sowohl die wachsende Zahl von Anwenderpaketen der verschiedenen großen Systeme als auch eine beeindruckende Zahl von Büchern zu dieser Thematik. Für einen vollständigeren Überblick über Tendenzen und Anwendungen der Computeralgebra siehe den schon erwähnten Computeralgebra-Report [7].

[2] Im Original: ... the algebra will be so taxing that few investigators will risk their health, happiness and professional well-being attempting a solution.

2.3 Computeralgebrasysteme (CAS)

Die ersten Anfänge der Entwicklung von Programmen der Computeralgebra reichen bis in die frühen 50er Jahre zurück. So hat etwa H.G. Kahrimanian 1953 an der Temple University in Philadelphia eine Master Thesis zur analytischen Differentiation geschrieben, in deren Rahmen er auch ein Assemblerprogramm für die Univac I erstellte. Ende der 50er und Anfang der 60er Jahre wurden vor allem am MIT (Massachusetts Institute of Technology) große Forschungsanstrengungen unternommen, symbolisches Rechnen mit eigenen Hochsprachen zu entwickeln (Formula ALGOL, ABC ALGOL, ALADIN, ...). Von diesen Sprachen hat sich bis heute vor allem LISP als die Grundlage für die meisten Computeralgebrasysteme erhalten, weil in dessen Sprachkonzept die strenge Trennung von Daten- und Programmteil, wie wir sie als typisch für die Computeralgebra herausgearbeitet hatten, bereits aufgehoben ist.

Die Wurzeln der ersten allgemeinen Systeme liegen in Versuchen verschiedener Fachwissenschaften, die jeweils typischen umfangreichen symbolischen Rechnungen mit einem Computer als Hifsmittel zu vollziehen. Schwardmann [15] weist auf die Programme CAYLEY für Algorithmen in der Gruppentheorie und SAC zum Rechnen mit multivariaten Polynomen und rationalen Funktionen (Mathematik) sowie SHEEP für die Relativitätstheorie und MOA für die Himmelsmechanik (Physik) hin.

Die ersten Systeme allgemeiner Ausrichtung, REDUCE und MACSYMA, basieren auf LISP und sind seit Mitte der 60er Jahre im Einsatz. REDUCE wurde von A. Hearn aus einem System für Berechnungen in der Hochenergiephysik (Feynman-Diagramme, Wirkungsquerschnitte) entwickelt, während MACSYMA am MIT im Zusammenhang mit Untersuchungen zur künstlichen Intelligenz im Rahmen des DARPA-Programms entstand. GRABMEIER klassifiziert diese in [6] als *Allzwecksysteme der ersten Generation* und weist darauf hin, daß die programmiertechnischen Restriktionen jener Zeit (Lochstreifen, Stapelbetrieb) für symbolische Rechnungen, die aus noch darzulegenden Gründen meist stärker dialogorientiert sind, denkbar ungeeignete Bedingungen boten.

Mit der Entwicklung stärker dialogorientierter Rechentechnik in der 70er und 80er Jahren erhielten diese Entwicklungen neue Impulse. Sie beginnen ebenfalls, wie auch die des Computers als "number cruncher", bei der Arithmetik, hier allerdings der *Arithmetik symbolischer Ausdrücke*. Entsprechend bilden bei den Computeralgebrasystemen der zweiten Generation eine interaktiv zugängliche Polynomarithmetik zusammen mit einem regelbasierten Simplifikationssystem den Kern des Systemdesigns, um den herum mathematisches Wissen in Anwenderbibliotheken gegossen wurde und wird. Diese Systeme sind durch ein Zwei-Ebenen-Modell gekennzeichnet, in dem die Interpreterebene zwar gängige Programmablaufstrukturen unterstützt, jedoch keine Datentypen kennt, sondern es prinzipiell erlaubt, alle im Kernbereich, der *ersten Ebene*, implementierten symbolischen Algorithmen mit allen möglichen Daten zu kombinieren. Weitergehende programmiersprachliche Elemente wie insbesondere Typisierung werden nur rudimentar unterstützt.

Neben neuen Versionen der "alten" Systeme REDUCE und MACSYMA, entstanden dabei eine Reihe neuer Systeme, von denen besonders MAPLE, MATHEMATICA und DERIVE zu nennen sind.

MATHEMATICA entwickelte sich aus dem von Steven Wolfram Ende der 70er Jahre entworfenen System SMP, das wiederum aus der Notwendigkeit geboren wurde, komplizierte algebraische Manipulationen in bestimmten Bereichen der theoretischen Physik effektiv und zuverlässig auszuführen. Es ist das erste System, das unmittelbar in der sich zu dieser Zeit im Compilerbereich durchsetzenden Sprache C geschrieben wurde. In der weiteren Entwicklung spielte dieses System eine Vorreiterrolle in der konsequenten Vermarktung von Software aus dem symbolischen Bereich, was bis heute in einer äußerst restriktiven Lizenzpolitik der inzwischen speziell für die Weiterentwicklung und den Vertrieb gegründeten Firma Wolfram Research, Inc. zum Ausdruck kommt. Der Versuch, schwerpunktmäßig über rein marktorientierte Mechanismen die Mittel für die Weiterentwicklung der algorithmischen und softwaretechnischen Seite eines solch komplexen Produkts aufzubringen, hat MATHEMATICA, besonders mit seiner neuen Version 3.0 (Sommer 1996), in die vorderste Front

der "Großen" gebracht. Gleichwohl werden diese Aktivitäten, ähnlich derer von Bill Gates, von der weltweiten Gemeinschaft der Computeralgebraiker teilweise mit großen Vorbehalten verfolgt.

Einen anderen Weg der Kommerzialisierung gingen A.D. Rich und D.R. Stoutemyer mit der Gründung von Soft Warehouse, Inc. in Honolulu (Hawaii) ebenfalls im Jahre 1979. Neben Mainframes begannen zu dieser Zeit Arbeitsplatzcomputer mit geringen technischen Ressourcen eine zunehmend wichtige Rolle zu spielen, so daß die Frage entstand, ob man CAS mit ihren traditionell hohen Hardware-Anforderungen auch für solche Plattformen "zuschneiden" kann. Mit dem System muMATH-79 und der (wiederum LISP-basierten) Sprache muLISP-79 wurde darauf eine überzeugende Antwort gefunden. Nach fast zehnjähriger "Ehe" mit Microsoft nahm die Firma die weitere Entwicklung in die eigenen Hände und brachte 1988 ein Nachfolgeprodukt unter dem Namen *DERIVE – A Mathematical Assistant* auf den Markt, das mit minimalen Hardware-Anforderungen unter dem Betriebssystem DOS gute symbolische Fähigkeiten entwickelt. Nachdem in den letzten Jahren durch die rasante Erweiterung der Hardware-Ressourcen von Arbeitsplatzrechnern dieser Anwendungsbereich auch von den anderen CAS zunehmend erobert wird, konzentriert sich die Firma auf das Taschenrechner-Geschäft und hat in Zusammenarbeit mit HP und TI zwei interessante Kleinstrechner mit symbolischen Fähigkeiten mitentwickelt, den HP-486X (1991) und den TI-92 (1995).

Ähnliche Überlegungen des "Downsizing" der bis dahin nur auf Mainframes laufenden großen Systeme lagen der Entwicklung von MAPLE an der University of Waterloo zugrunde. In nur drei Wochen wurde Ende 1980 eine erste Version (mit beschänkten Fähigkeiten) entwickelt, die auf *B*, einer ressourcen- und laufzeitfreundlichen Sprache aus der BCPL-Familie aufsetzte, aus der sich C als heutiger Standard entwickelt hat. Seit 1983 sind Versionen von Maple auch außerhalb der University of Waterloo in Gebrauch. Zur Gewährleistung einer effizienten Portabilität auf immer neue Computergenerationen wurde im Gegensatz zu den großen LISP-Systemen Wert auf einen kleinen, heute in C geschriebenen Systemkern gelegt, der die erforderlichen Elemente einer symbolischen Hochsprache implementiert, in der seinerseits die verschiedenen symbolischen Algorithmen geschrieben werden können.

Auch hier stellte sich schnell heraus, daß die Anforderungen, die Vertrieb, Support einer Nutzergemeinde sowie die Portierung auf immer neue Rechnergenerationen stellen, die Möglichkeiten einer akademischen Anbindung sprengen und nur über eine entsprechende Firma möglich ist. Diese Rolle spielt seit Ende 1987 Waterloo Maple Inc., die im Gegensatz zu Wolfram Research aber nach wie vor mit einer sehr engen Bindung an die universitäte Gruppe um K. Geddes und G. Labahn über ein ausgebautes akademisches Hinterland verfügt, das ein arbeitsteiliges Vorgehen in der softwaretechnischen und algorithmischen Weiterentwicklung des Systems ermöglicht.

Auch IBM als eine der großen Softwarefirmen hat sich seit den Anfangstagen der Computeralgebra mit eigenen Aktivitäten an den wichtigsten Entwicklungen beteiligt. Das betrifft insbesondere nach einigen Sprachentwicklungen in den 60er Jahren das System SCRATCHPAD, mit dem im *IBM T.J. Watson Research Center at Yorktown, NY,* seit den 70er Jahren diese Untersuchungen fortgesetzt wurden. SCRATCHPAD verwendet als bisher einziges System im Design ein strenges Typsystem. Bis zum Beginn der 90er Jahre standen diese Entwicklungen ausschließlich auf IBM-Rechnerplattformen und nur in experimentellen Versionen zur Verfügung und fanden damit trotz ihrer Exklusivität keine weite Verbreitung. Auch mit der ersten offizielle Version von AXIOM im Jahre 1991 änderte sich daran wenig. Das mag IBM bewogen haben, 1992 im Zuge einer generellen "Flurbereinigung" des Firmenprofils die Rechte an AXIOM der Firma NAG Ltd. zu übertragen, einer Non-profit-Firma, die sich bereits auf dem Gebiet wissenschaftlicher Software ausgewiesen hatte.

AXIOM ist mit diesem streng getypten Ansatz ein

Computeralgebrasystem der dritten Generation,

mit denen versucht wird, ins Zentrum des Designs moderne programmiertechnische Ansätze der Typisierung, Modularisierung und Datenkapselung zu setzen und diese konsequent in einem sym-

System Neueste Version	PC-OS PC-Ressourcen	Preis (E)inzellizenz (S)tudentenversion
AXIOM 2.1	Win95, NT, Linux 8..16 MB RAM	E: 1710 DM S: —
Derive 3.0	DOS 512 kB RAM, $<$ 1 MB disk	E: 198 DM S: 90 DM
	Win (8 MB RAM)	E: 398 DM
Macsyma 2.2.	Win3.x, 95, NT 16-32 MB RAM, 27 MB disk	E: 450 DM Lite: $\approx$ 170 DM
Maple V.4	Win3.x, 95, NT, Linux 18-30 MB disk, 8 MB RAM	E: $\approx$ 1400 DM S: 128 DM
Mathematica 3.0	Win95, NT, Linux 8 MB RAM	E: $\approx$ 3000 DM S: 250 DM
2.2	Win 3.11.	E: $\approx$ 3000 DM
MuPAD 1.3.	Win95, NT, Linux	bisher kostenlos im akademischen Bereich
Reduce 3.6.	Win3.x, 95, NT, Linux 4 MB RAM, 10 MB disk	professional: 835 DM personal: 180 DM

Die großen Computeralgebrasysteme allgemeiner Ausrichtung im Überblick
(Stand November 1997)

bolisch orientierten Kontext zu realisieren. Computeralgebrasysteme bieten hierfür denkbar gute Voraussetzungen, denn ein solches Typsystem ist ja seinerseits symbolischer Natur mit einem stark algebraisierten Hintergrund, sollte also prinzipiell mit den vorhandenen sprachlichen Mitteln eines CAS modelliert werden können. Konsequente Versuche, ein solches Typ-System mit den Mitteln eines CAS der zweiten Generation zu modellieren, wurden auch mit dem Maple-Paket Gauss ([8]) sowie dem Domain-Konzept in MuPAD (das allerdings stark dem Ansatz von Maple folgt) vorgelegt. Die *Integration* eines solchen Typsystems in die Programmiersprache bedeutet jedoch, diese symbolischen Manipulationen unterhalb der Interpreterebene zu vollziehen, also gegenüber Systemen der zweiten Generation eine weitere Ebene zwischen Interpreter und Systemkern einzufügen, die diese Typinformationen in der Lage ist auszuwerten. Dieses Konzept wird von AXIOM auch in einer Standalone-Version, dem Programmiersystem ALDOR, mit beeindruckenden Ergebnissen verfolgt. Es zeigt sich, daß die algebraische Natur des Targets dieser Bemühungen mit der algebraischen Natur der theoretischen Fundierung von programmiertechnischen Entwicklungen gut harmoniert und etwa mit parametrisierten Datentypen und virtuellen Objekten (bzw. abstrakten Datentypen) bereits zu einer Zeit experimentiert wurde, in der an die bis heute rudimentären Versuche mit "Templates" und Ähnlichem in den "großen Sprachfamilien" C/C++ bzw. PASCAL/MODULA/OBERON noch nicht zu denken war. Besonders deutlich wird dies im Design von MAGMA, einem Nachfolger von CAYLEY, das für komplexe Fragestellungen aus den Bereichen Algebra, Zahlentheorie, Geometrie und Kombinatorik entworfen wurde und Konzepte der universellen Algebra und Kategorientheorie in einem objektorientierten Ansatz direkt umsetzt.

In den letzten fünf bis zehn Jahren wurden alle diese Systeme, die bis dahin nur in einer mehr oder weniger experimentellen Fassung vorlagen, mit größerem Aufwand unter marktorientierten Gesichtspunkten weiterentwickelt. Schwerpunkt waren dabei vor allem die Einbindung von bequemeren, fensterbasierten Ein- und Ausgabetechniken auf Notebook-Basis, hypertextbasierte Hilfesysteme und leistungsfähige Graphikmoduln. Sie besitzen damit heute in der Regel eine ausgereifte Benutzeroberfläche, sind gut dokumentiert und auf den verschiedensten Plattformen (bis hin zu

leistungsfähigeren Personalcomputern unter den gängigen Betriebssystemen Win 3.11, Win'95 oder OS/2 sowie LINUX) verfügbar[3]. Zu den Systemen gibt es einführende Bücher und Bücher zum vertieften Gebrauch, für spezielle Anwendungen und zum Einsatz in der Lehre. Zudem erscheinen regelmäßig Nutzerinformationen und Informationen über frei zugängliche Anwenderpakete. Weiterhin haben sich Benutzergruppen gebildet, und es werden Anwendertagungen durchgeführt.

Zugleich entstanden und entstehen auch neue Systeme allgemeiner Ausrichtung. Da CAS jedoch einen hohen Entwicklungsaufwand erfordern (mehr als 100 Mannjahre Programmierarbeiten), sind diese neuen Systeme in der Regel nicht sehr leistungsfähig oder haben nur einen sehr eingeschränkten Anwendungsbereich. Eine Ausnahme bildet hier das System MuPAD, dessen Entwicklung im Jahre 1989 von einer Arbeitsgruppe an der Uni-GH Paderborn unter der Leitung von Prof. B. Fuchssteiner begonnen und in den letzten Jahren unter aktiver Beteiligung einer großen Zahl von Studenten, Diplomanden und Doktoranden intensiv vorangetrieben worden ist. Auch hier stellte es sich heraus, daß ein solches System, wenn es eine gewisse Dimension erreicht hat, nicht allein aus dem akademischen Bereich heraus gewartet und gepflegt werden kann. Seit dem Frühjahr 1997 hat deshalb die eigens dafür gegründete Firma *SciFace* einen Teil dieser Aufgaben insbesondere aus dem software-technischen Bereich übernommen.

Schließlich gibt es mehrere Entwicklungen, deren Anstrengungen nicht auf ein komplett anwendungsfähiges System, sondern eine Art Toolbox für symbolisches Rechnen ausgerichtet sind, einer Umgebung also, die es gestattet, neue Algorithmen und programmiertechnische Ansätze auf der Basis einer soliden Programmbibliothek zu untersuchen. Der Prototyp dieser Herangehensweise war das System ALDES/SAC-2 von G.Collins und R.Loos, das inzwischen mehrere Migrationen auf neue Plattformen und Computerarchitekturen erfahren hat, etwa MAS (Modula), SACLIB (C), PARSAC und PACLIB (parallele Architekturen). In diesem Zusammenhang ist auch AXIOMs Standalone-Compiler ALDOR zu nennen, der einen zunehmenden Teil der mathematischen Fähigkeiten von AXIOM in ebendieser Weise zur Verfügung stellt und damit deren Integration auf Bibliotheksniveau in andere Programme ermöglicht.

Im Zusammenhang mit der Entwicklung paralleler Plattformen und verteilter Systeme auf Client-Server-Basis wird auch zunehmend über die Bereitstellung von symbolischer Rechenleistung in einem solchen Kontext nachgedacht. Hierfür sind in die gängigen Computeralgebrasysteme neue Kommunikationskonzepte zu integrieren, die an die besonderen Anforderungen an Struktur und Umfang des Datenaustauschs bei symbolischen Rechnungen angepaßt sind.

Daneben gibt es eine große Anzahl kleinerer experimenteller Systeme mit eingeschränkter Ausrichtung. Sie wurden für symbolische Rechnungen auf sehr begrenzten Gebieten der Mathematik oder Physik entwickelt und erreichen durch speziell angepaßte Datenstrukturen und Algorithmen oftmals eine erstaunliche Leistungsfähigkeit. Sie sind meist das Produkt kleiner Gruppen von Spezialisten, die das Topwissen ihres Fachgebiets mit guten Programmierkenntnissen verbinden konnten. Beispiele für solche Systeme sind in der Physik SCHOONSHIP, FORM (Hochenergiephysik), CAMAL (Himmelsmechanik), SHEEP und STENSOR (allgemeine Relativitätstheorie) sowie in der Mathematik MAGMA und GAP (Gruppentheorie), PARI, KANT, SIMATH (Zahlentheorie), Macaulay, AlPi, CoCoA, Singular, Felix, Bergman (kommutative und nichtkommutative Algebra) und LIE (Lietheorie).

Im Zuge der Herausbildung einer Computermathematik, wie in [6] prognostiziert, entsteht zunehmend die Frage nach der Integration von Softwareentwicklungen, die bisher in anderen Kreisen vorangetrieben und gepflegt wurden. Dies betrifft insbesondere die Interaktion zwischen symbolischen Verfahren der Computeralgebra im engeren Sinne, in sehr umfangreichen Programmpaketen vorhandene ausgefeilte numerische Applikationen des "wissenschaftlichen Rechnens" sowie Entwicklungen der Computergraphik. Die heute existierenden Computeralgebrasysteme haben sowohl

[3] Einige der Systeme haben sich aber nicht mehr der Mühe unterzogen, ihre Implementierung in die 16-Bit-Welt von Win 3.11 zu transportieren.

numerische als auch graphische Fähigkeiten, die durchaus zu beeindrucken vermögen, jedoch von der Leistungsfähigkeit der genannten "professionellen" Entwicklungen weit entfernt sind. Aktuelle Bemühungen sind deshalb darauf gerichtet, in Zukunft die Vorteile arbeitsteiligen wissenschaftlichen Vorgehens stärker zu nutzen, statt das Fahrrad mehrfach zu erfinden. Auf der Softwareseite finden sich solche Bemühungen in Designkonzepten wieder, die sich stärker an der eigenen *Kernkompetenz* orientieren und den "Rest" durch Anbindung an andere Kernkompetenzen abdecken. Dies erfolgte für die großen CAS zuerst im Graphikbereich, wo sich AXIOM (Open Inventor) und REDUCE (Gnuplot) bereits seit einiger Zeit an eigenständige Entwicklungen im Graphiksektor angekoppelt haben. Weiterhin spielen Verbindungen zu Numerikpaketen wie die AXIOM-Anbindung an die Fortranbibliothek f90 oder die IRENA-Anbindung von REDUCE eine zunehmend wichtige Rolle. Auch von der Numerikseite her gibt es derartige Aktivitäten, wie die Einbindung von symbolischen Fähigkeiten von MAPLE in das im ingenieurtechnischen Bereich stark verbreitete MATHCAD 6.0 belegt.

Inzwischen ist dies zu einem deutlich sichtbaren Trend geworden und eigene Protokolle wie etwa MathLink für die Kommunikation zwischen C und MATHEMATICA entwickelt worden. Eine solche C-Einbindungsmöglichkeit liegt auch der Integration von MAPLE in MATHCAD 6.0 zu Grunde. Darüber hinaus gibt es inzwischen im Rahmen des OpenMath-Projekts Bemühungen, ein eigenes Protokoll für symbolische Rechnungen zu entwickeln, das von allen CAS unterstützt werden soll, um den Datenaustausch zwischen verschiedenen Applikationen aus diesem Gebiet generell zu erleichtern.

Der dritte große Bereich des Zugriffs auf Fremdkompetenz liegt in der Gestaltung des Ein- und Ausgabedialogs der großen Systeme. Mathematische Formeln sind oft auch von drucktechnisch komplizierter Gestalt und verwenden eine Fülle extravaganter Symbole in den verschiedensten Größen. Hierfür hat sich im Bereich der mathematischen Fachliteratur das Satzsystem TEX von D.E.Knuth und dessen etwas komfortablere Umgebung LATEX als ein de facto Standard weitgehend durchgesetzt. Deshalb bieten die meisten großen CAS auch eine Ausgabe in dieser Form an, die es erlaubt, Formeln direkt in mathematische Texte zu übernehmen bzw. diese direkt mit entsprechenden Wiedergabewerkzeugen in optisch ansprechender Form auszugeben. Ein großer Schritt vorwärts in dieser Richtung wurde durch die Notebook-Oberflächen von Mathematica 3.0 und Maple V.4 erreicht, die als direktes Graphik-Frontend für eine solche Darstellungsart ausgelegt sind.

In Verbindung mit Client-Server-Techniken eröffnen sich für diese Entwicklungen nochmals vollkommen neue Perspektiven, die in entsprechenden Forschungsaktivitäten (z.B. PoSSo und OpenMath im EG-Raum) derzeit untersucht werden: Ein "Kontrollzentrum" mit im wesentlichen nur Notebook- und Kommunikationsfähigkeiten als Mensch-Maschinensytem-Schnittstelle hat Zugang zu einer Vielzahl von Spezialprogrammen unterschiedlicher Kompetenz, die die gestellten Aufgaben arbeitsteilig lösen. So kann z.B. ein Programm den symbolischen Teil der Aufgabe bearbeiten, das Ergebnis (über das Kontrollzentrum) zur numerischen Auswertung einem zweiten Programm zur Verfügung stellen, dieses daraus Daten für eine graphische Auswertung erzeugen, die einem dritten Programm zur Graphikausgabe weitergereicht werden.

2.4 Zusammenfassung

Wir hatten in den vorangehenden Abschnitten das symbolische Rechnen, verstanden als

inhaltliche Verarbeitung spezieller mathematischer Zeichenketten,

als eine zentrale menschliche Kulturtechnik kennengelernt, um Zusammenhänge in der uns umgebenden Natur (die Natur der menschlichen Gemeinschaft eingeschlossen) quantitativ zu erfassen und einer bewußten Ausnutzung zugänglich zu machen. Als solches ist das symbolische Rechnen nicht an den Computer gebunden, hat sich aber gleichwohl mit dem Einsatz des Computers als Werkzeug vollkommen neue Horizonte eröffnet.

Andererseits hatten wir gesehen, daß das symbolische Rechnen nicht eine weitere Computeranwendung schlechthin unter vielen anderen ist, sondern *in natürlicher Weise* Entwicklungen, die die Informatik als Ganzes hervorgebracht haben, weiterführt: Der Computereinsatz für symbolische Rechnungen eröffnet einen neuen Abschnitt auf dem Weg des Computers vom primitiven Bitknipser zu einem Universalwerkzeug für geistige Arbeit. Es beginnt damit eine neue Etappe auf dem Weg der praktischen Realisierung des theoretischen Anspruchs, den die CHURCHsche These impliziert.

Gleichwohl sucht man diesen Gedanken in verschiedenen Quellen, die sich eine umfassende Darstellung der Entwicklungen in der Informatik vorgenommen haben, vergebens. So enthält etwa der Duden Informatik [4] nicht einmal ein Stichwort *symbolisches Rechnen* oder *Computeralgebra*. Gleichwohl kann man den Einfluß dieser neuen Arbeitsmittel auf den Umbruch unserer technisierten Arbeitswelt insbesondere im ingenieurtechnischen Bereich kaum überschätzen. Dort, wo heute noch dicke Formelsammlungen und Tafelwerke das Berufsbild prägen, die trotz ihrer Dicke genau wie unser Berg numerischer Daten immer nur eine sehr beschränkte Sicht auf *Fakten* und keine *Einsichten* vermitteln können, werden computergestützte Formen der Wissensrepräsentation zu vollkommen neuen Arbeitsweisen führen.

Bereits mit dem Taschenrechner haben große Tafelwerke ausgedient. Die vorab generierte und in ihnen gespeicherte Information über wichtige, immer wieder auftretende Funktionen wurde durch ein Instrument ersetzt, das diese Information *unmittelbar vor Ort selbst generieren* kann. Neben der größeren Präzision der Antwort (sie wird ja genau nach den konkreten Erfordernissen, also *maßgeschneidert*, erzeugt und kommt nicht "von der Stange") erweist sich ein solcher Zugang auch als genauer, schneller, kompakter und flexibler.

Derselbe Prozeß,

der Übergang von einer fakten- zu einer algorithmenorientierten Wissenspräsentation,

vollzieht sich *auf der Seite des Fachwissens* mit dem Einzug von Computermethoden des symbolischen Rechnens. Statt dicke Integraltafeln einzusetzen, in denen ellenlang vorab erzeugte Information gespeichert ist, zu deren Einsatz für konkrete Berechnungen es aber trotzdem noch einer gehörigen Portion Kunstfertigkeit bedarf, wird es auch hier auf einmal möglich, die für eine konkrete Aufgabenstellung notwendige Information *vor Ort* aus allgemeineren symbolischen Zusammenhängen *algorithmisch* zu generieren. Die damit verbundene neue Qualität wird besonders augefällig beim Blick auf Anwenungsgebiete wie Elektrotechnik, Statik, Architektur etc., in denen die Migration von Fachwissen in diese neue, kompaktere Form zu denselben Effekten führt wie oben für die Taschenrechner beschrieben. Auch hier haben wir es wiederum mit einem Übergang

von der quantitativen Anhäufung einer Vielzahl aufbereiteter Fakten	zur qualitativen Implementierung konstruktiver Verfahren

zu tun. Wir werden damit in die Lage versetzt, komplexere Systeme zu beherrschen und Beschränkungen, die aus der früher notwendigen stärkeren Simplifizierung der fachlichen Grundlagen herrühren (z.B. Verwendung statisch einfach handhabbarer Trägerprofile, standardisierte architektonische Teillösungen etc.), zunehmend aufgeben zu können.

Der Vollständigkeit halber sei ein weiterer Umbruch benannt, der sich mit

zunehmender Vernetzung

auf der einen Seite und Mobilcomputing auf der anderen abzeichnet: In naher Zukunft wird es mit diesen Instrumenten möglich sein, zunehmend *selbst* in diesem heterogenen Informationsraum zu operieren, statt mit vorgefertigten Instrumenten vor Ort eine *vorab generierte Kompetenz* zu konsultieren. Man wird statt dessen

1. weltweit auf von Spezialisten an verschiedenen Orten gepflegte Kompetenz unmittelbar zugreifen können (OpenMath, Client-Server, Intranet-Konzept)

2. dezentral Meßdaten mit Monitoring-Verfahren erfassen können (Brückenprüfung, Patientenüberwachung,...)

3. die damit generierten Erfahrungen selbst unmittelbar in diesem Informationsraum einbringen und damit anderen zugänglich machen können (World Wide Web)

Es ergibt sich insgesamt ein Bild des symbolischen Rechnens mit dem Computer, das J.Grabmeier in [6] in die folgenden Worte faßt:

> Viele Probleme aus der Ingenieurwelt, den Naturwissenschaften und den Wirtschaftswissenschaften sind heute ohne massiven Einsatz von Computern nicht lösbar. Die dahinterliegenden Probleme werden mit den Methoden des Wissenschaftlichen Rechnens angegangen. Dabei werden mehr und mehr die traditionellen numerischen Rechnungen durch symbolisches Rechnen mit dem Computer ersetzt bzw. ergänzt...
>
> Der Siegeszug der Computeralgebra in den letzten Jahren ist eng gekoppelt mit der stürmischen Entwicklung von immer neuen Rechnergenerationen, die es erst möglich gemacht hat, die besonders rechen- und speicherintensiven Programme und Systeme zum symbolischen Rechnen zu realisieren.
>
> Aber der Aufwand lohnt sich: Wenn man statt einer Zahl eine parameterabhängige Formel als Ergebnis erzielt, hat man nicht nur ein Problem gelöst, sondern eine Klasse von möglicherweise *unendlich vielen* Problemen erledigt. Dadurch wird ein Qualitätssprung möglich, denn die Formel erlaubt es nun z.B., die Parameter zu optimieren oder schnell auf Veränderungen zu reagieren. ...
>
> Wie heute ein Taschenrechner zum Alltag gehört, wird künftig jeder Ingenieur und jeder, zu dessen Aufgaben das Lösen, Erlernen oder Lehren mathematischer Probleme gehört, Zugriff auf ein Computeralgebra-System haben. Die verschiedenen schon heute verfügbaren Komponenten für numerisches und symbolisches Rechnen, für Statistik und andere mathematische Gebiete, für Graphik und Animation, Textverarbeitung und Dokumentation mit Hypertext-Systemen und vieles mehr sehe ich in nicht allzu ferner Zukunft über entsprechende Schnittstellen zu individuell kombinierbaren Computermathematik-Systemen für das Wissenschaftliche Rechnen zusammenwachsen. Die Computeralgebra leistet damit einen wesenlichen Beitrag für eine der Schlüsseltechnologien unserer technikbestimmten Gesellschaft.

Kapitel 3

Aufbau und Arbeitsweise eines Computeralgebrasystems der zweiten Generation

In diesem Kapitel wollen wir uns näher mit dem Aufbau und der Arbeitsweise eines Computeralgebrasystems der zweiten Generation vertraut machen und dabei insbesondere Unterschiede und Gemeinsamkeiten mit ähnlichen Arbeitsmitteln aus dem Bereich der Programmiersysteme herausarbeiten.

3.1 Interpreter versus Compiler

Bei derartigen Programmiersystemen unterscheiden wir zunächst einmal Interpreter und Compiler. In den Demonstrationen in vergangenen Vorlesungen trat uns ein CAS stets mit einer Interpreter-Oberfläche gegenüber. Wir wollen uns deshalb zunächst die Unterschiede zwischen beiden Systemen ins Gedächtnis rufen.

Ein **Interpreter** ist ein Programmiersystem, in dem einzelne Anweisungen nach der notwendigen syntaktischen Prüfung *sofort* ausführt werden.

Ein **Compiler** ist ein Programmiersystem, das in einer ersten Phase, zur *Übersetzungszeit*, ein *vollständiges Programm* analysiert, in eine maschinennahe Sprache übersetzt und abspeichert.
In einer zweiten Phase, zur *Laufzeit*, wird das übersetzte Programm von einem *Lader* zur Abarbeitung in den Hauptspeicher geladen.

Interpreter und Compiler durchlaufen dabei beide die Phasen *lexikalische Analyse, syntaktische Analyse* und *semantische Analyse*. Ein Compiler durchläuft weiterhin die Phasen *Codegenerierung* und evtl. *Codeoptimierung*.

Besonderheiten eines Compilers:

Das Programm wird *einmal* während des Compilierens analysiert und liegt danach in unmittelbar durch den Computer abarbeitbarer Form vor. Das zentrale Paradigma des Compilers ist also das des *Programmflusses*, längs dessen Daten geschleust werden.

Das Übersetzen ist durch die ausführlichere Analysephase aufwendiger als die Interpretation desselben Programms, die Optimierung führt aber zu Laufzeitvorteilen.

Compiler werden deshalb vorwiegend dann eingesetzt, wenn ein und dasselbe Programm mit einer Vielzahl unterschiedlicher Datensätze abgearbeitet werden soll und die (einmaligen) Übersetzungszeitnachteile durch die (mehrmaligen) Laufzeitvorteile aufgewogen werden. Dies erfolgt besonders bei einer Sicht auf den Computer als "virtuelle Maschine", d.h. als *programmierbarer Rechenautomat*. Compiler sind typische Arbeitsmittel einer *stapelorientierten Betriebsweise*.

Besonderheiten eines Interpreters:

Diese bestehen insbesondere in seiner größeren Flexibilität, da auch noch während des Ablaufs in das Geschehen eingegriffen werden kann. Werte von (globalen) Variablen sind während der Programmausführung abfrag- und änderbar und einzelne Anweisungen oder Deklarationen im Quellprogramm können geändert werden. Das ganze Programm mit diesen Änderungen ist sofort ausführbar. Das zentrale Paradigma des Compilers ist also das der *Datenlandschaft*, die durch Anwendung verschiedener Konstruktionselemente erstellt wird.

Als entscheidender Nachteil schlagen längere Rechenzeiten zu Buche, da z.B. die Adressen der verwendeten Variablen mit Hilfe der Bezeichner ständig neu gesucht werden müssen. Ebenso ist Code-Optimierung nur beschränkt möglich.

Interpreter werden deshalb vor allem in Anwendungen eingesetzt, wo diese Nachteile zugunsten einer größeren Flexibilität des Programms, der Möglichkeit einer interaktiven Ablaufsteuerung und der Inspektion von Zwischenergebnissen in Kauf genommen werden. Interpreter sind typische Arbeitsmittel einer *dialogorientierten Betriebsweise*.

All diese Anforderungen ergeben sich auf natürliche Weise beim Einsatz eines CAS, wenn es als *Hilfsmittel für geistige Arbeit* verstanden wird. Deshalb sind alle großen CAS wenigstens in ihrer obersten Ebene Interpreter. Dies trifft selbst auf die erwähnten *Toolbox-Systeme* zu, die meist einen mit klassischen Analysewerkzeugen (`lex` und `yacc`) erstellten *Parser* als Minimaloberfläche anbieten.

Von dieser Dialogfläche aus werden einzelne Datenkonstruktoren (Funktionen, Unterprogramme) aufgerufen, für die jedoch eine andere Spezifik gilt: Bei diesen handelt es sich um standardisierte, auf einer höheren Abstraktionsebene optimierte algorithmische Lösungen, die während des Dialogbetriebs mit einer Vielzahl unterschiedlicher Datensätze aufgerufen werden (können). Sie gehören also in das klassische Einsatzfeld von Compilern. Teile des CAS, z.B. einzelnen Prozeduren, ganze Pakete oder Moduln liegen deshalb bereits in übersetzter Form vor. Auch während des Dialogbetriebs erzeugte neue Daten können entweder automatisch oder auf spezielle Anforderung des Nutzers hin übersetzt werden. Dieser Ansatz des *inkrementellen Übersetzers* wird besonders dann eingesetzt, wenn eine neukonstruierte Funktion mit vielen Argumenten ausgewertet werden soll wie etwa bei der Generierung von Graphikausgaben. Diese Compilation ist nur in der entsprechenden Sitzung wirksam.

Die Systeme sind also, wie alle größeren Interpretersysteme,

Top-level interpretiert, low-level compiliert.

Dabei sind drei Ebenen, auf denen Übersetzung möglich ist, zu unterscheiden:

- Während der Systementwicklung erstellte Übersetzungen, die in Form von Bibliotheken mit dem System mitgeliefert (oder nachgeliefert) werden.
- Eigenentwicklungen, die mit geeigneten Instrumenten übersetzt und archiviert und damit in nachfolgenden Sitzungen in bereits compilierter Form verwendet werden können.
- Im laufenden Betrieb erzeugte Übersetzungen, die für nachfolgende Sitzungen nicht mehr zur Verfügung stehen.

Die Möglichkeiten des inkrementellen Übersetzens führen dazu, daß die Systeme selbst inkrementellen Charakter haben, d.h. ihre Fähigkeiten laufend erweitert werden. Dabei sind die Rollen zwischen Entwicklern und Anwendern von CAS nicht so klar verteilt wie bei vielen anderen Programmpaketen, sind doch gerade die *mathematischen* Fähigkeiten der großen CAS aus einer Vielzahl von Aktivitäten zur programmtechnischen Fixierung mathematischen Know-Hows entstanden, an denen sich Arbeitsgruppen beteiligen, die rund um die Welt verteilt sind und nur sehr lose Kontakte zueinander und zum engeren Kreis der Systementwickler halten. Letztere tragen hauptsächlich die Verantwortung für die Weiterentwicklung der entsprechenden programmiertechnischen Instrumentarien. Die Kommunikation zwischen diesen Gruppen erfolgt über Mailing-Listen, News-Gruppen, Anwender- und Entwicklerkonferenzen, Artikel in Zeitschriften, Bücher etc., insgesamt also mit den im üblichen Wissenschaftsbetrieb anzutreffenden Mitteln.

Computeralgebrasysteme sind deshalb **offene Systeme**.

Der Umfang des bereits implementierten mathematischen Wissens macht es aus Effizienzgründen zugleich erforderlich, über eine flexible Konfigurierbarkeit der Systeme entsprechend der jeweiligen Aufgabenstellung nachzudenken. In diesem Zusammenhang spielen insbesondere Konzepte des *dynamischen Ladens* eine wichtige Rolle. Dabei wird eine Startkonfiguration des Systems in den Hauptspeicher geladen, zu der je nach den Erfordernissen der auszuführenden Rechnungen zur Laufzeit weitere Codestücke hinzugeladen bzw. entfernt werden, womit der Hauptspeicherbedarf begrenzt werden kann.

Die (konsistente) Entwicklung und Betreuung solcher Systeme stellt sehr hohe Anforderungen an die verwendeten Konzepte des Software-Managements. Alle modernen Entwicklungen, begonnen von Modularisierungs- und Datenkapselungskonzepten, über dynamisches Laden und Client-Server-Modelle bis hin zu Modellen verteilten Rechnens finden deshalb in diesem Bereich sowohl ein sehr komplexes Target, in dem die Leistungsfähigkeit neuer Konzepte geprüft werden kann, als auch eine Fülle von Anregungen und Fragestellungen, wie sie in dieser Komplexität von kaum einem zweiten Gebiet in der Informatik aufgeworfen werden.

3.2 Der prinzipielle Aufbau eines Computeralgebrasystems

Nach dem Start des entsprechenden Systems meldet sich die Interpreterschleife und erwartet eine Eingabe, typischerweise einen in *linearisierter Form* einzugebenden symbolischen Ausdruck, der vom System ausgewertet wird. Das Ergebnis wird in einer stärker an mathematischer Notation orientierten *zweidimensionalen Ausgabe* angezeigt.

Neben derartigen Eingaben sowie einem `quit`-Befehl zum Verlassen der Interpreterschleife gibt es noch eine Reihe von Eingaben, die offensichtlich andere Reaktionen hervorrufen. Dies sind in Maple

- Eingaben der Form `?topic` zur Aktivierung des Hilfesystems und
- Eingaben der Form `plot(...)`, worauf im Rahmen der X-Windows-Umgebung ein Graphik-Ausgabefenster geöffnet wird.

Offensichtlich werden hierbei andere als die unmittelbaren symbolischen Fähigkeiten des CAS herangezogen.

Die *Interpreterschleife* des CAS bringt also verschiedene Teile des Systems zusammen, wovon nur eines der unmittelbar symbolische Rechnungen ausführende Kern ist, den wir als den *Systemkern* bezeichnen wollen. Neben den beiden Komponenten *Hilfesystem* und *Graphik-Interface* sind dies noch die der Ein- und Ausgabe dienende *Nutzerschnittstelle* (front end) sowie ein *numerisches Interface*, das die numerische Auswertung symbolischer Ausdrücke übernimmt. Letzteres ist oftmals enger in den Systemkern intergriert. Wir wollen es trotzdem an dieser Stelle einordnen, da die entsprechende Funktionalität nicht direkt mit symbolischen Rechnungen zu tun hat.

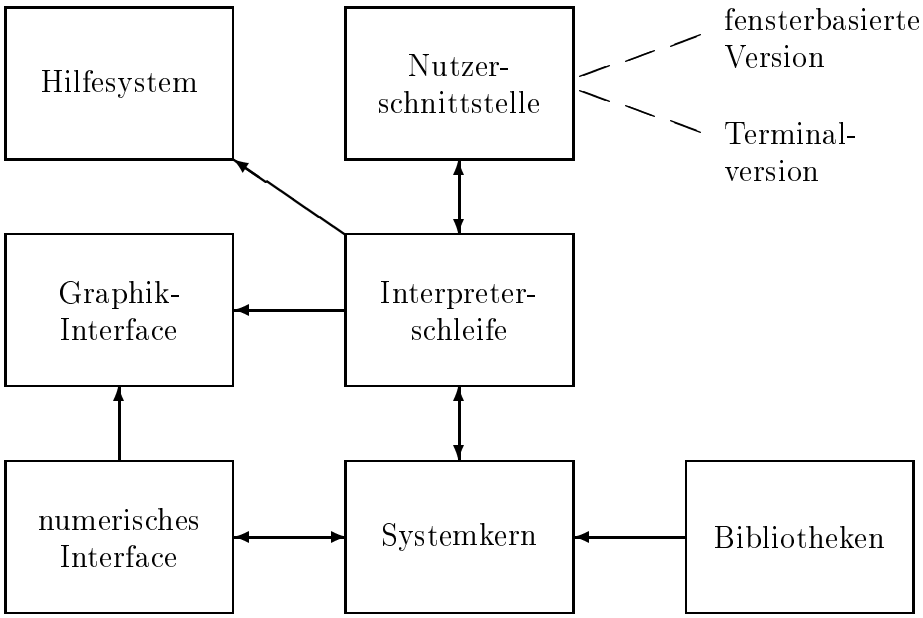

Bild 1: Prinzipieller Aufbau eines Computeralgebra-Systems

Das äußere Erscheinungsbild des Systems wird in erster Linie durch die Nutzerschnittstelle bestimmt. Genereller Bestandteil derselben ist ein *Formatierungssystem* zur Herstellung zweidimensionaler Ausgaben, das sich stärker an der mathematischen Notation orientiert als die Systemeingaben. Man unterscheidet *Terminalversionen*, in denen die Ausgabe im Textmodus erfolgt und die neben Ein- und Ausgabe meist einen rudimentären Zeileneditor besitzen, und *fensterbasierte Versionen*, in denen die Ausgabe im Graphikmodus erfolgt. Bei der Ausgabeformatierung mathematischer Formeln spielt die von D. KNUTH entwickelte Seitenbeschreibungssprache TEX bzw. deren komfortablere Erweiterung LATEX eine zentrale Rolle, die sich inzwischen als defacto-Standard für die elektronische Form mathematischer Publikationen etabliert hat. Alle CAS können deshalb ihre Ergebnisse in diesem Format ausgeben.

Graphikbasierte Ausgabesysteme sind heute in der Lage, *integrierte Dokumente* zu produzieren, die Texte, Ergebnisse von Rechnungen und Graphiken verbinden und in das Format gängiger Office-Systeme exportieren können. Eine besondere Vorreiterrolle spielen auf diesem Gebiet die neuen Versionen Mathematica 3.0 und Maple V.4 mit dem Konzept des *Arbeitsblatts*, womit sich zugleich vollkommen neue Horizonte in Richtung der (elektronischen) Publikation interaktiver mathematischer Texte eröffnen.

Hilfesysteme sind bei den einzelnen CAS sehr unterschiedlich entwickelt. Generell ist jedoch auch hier ein Trend hin zur Verwendung gängiger Techniken zum Entwurf von Hilfesystemen in Form hypertextbasierter Dokumente und entsprechender Analysewerkzeuge zu verspüren. Dabei wird zunehmend, ebenso wie beim Graphik-Interface, nicht nur auf entsprechende Ansätze, sondern auch auf bereits fertige Software-Komponenten zurückgegriffen. Dasselbe gilt, mit größeren Abstrichen, für das numerische Interface, da kommerzielle Pakete die speziellen Möglichkeiten adaptiver Präzision einer bigfloat-Arithmeitk, wie sie in CAS auf natürliche Weise zur Verfügung stehen, gewöhnlich nicht ausnutzen können. Insbesondere diese Fähigkeit, die etwa beim Bestimmen nahe beieinanderliegender Nullstellen von Polynomen oder beim Berechnen numerischer numeri-

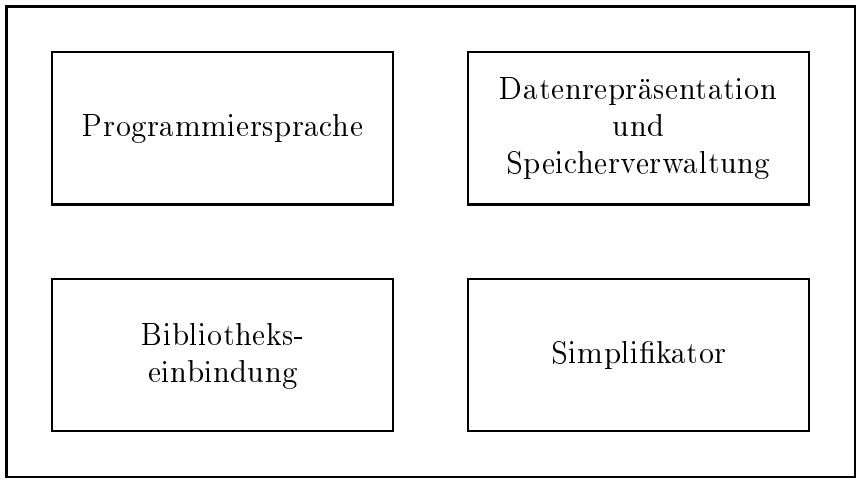

Bild 2: Komponenten des Systemkern-Designs

scher Näherungswerte hoher Präzision eine Rolle spielen, sind deshalb stärker in den Systemkern integriert.

Für das Design des Systemkerns sind die folgenden Komplexe zu berücksichtigen:

- Es wird eine *Programmiersprache* benötigt, mit der man den Ablauf des Interpreters steuern kann. Diese sollte die klassischen Steuerstrukturen einer *imperativen* Programmiersprache zur Verfügung stellen, vermehrt um Instrumente, die sich aus der Spezifik des symbolischen Rechnens ergeben.

- Es ist ein Konzept für die *Datenrepräsentation* zu entwickeln sowie damit verbunden eine entsprechende *Speicherverwaltung* bereitzustellen.

  Dabei ist zu berücksichtigen, daß symbolische Ausdrücke sehr unterschiedlicher und im Voraus nicht bekannter Größe sind, also als *dynamische Datentypen* mit einer ebensolchen *dynamischen* Speicherverwaltung anzulegen sind, natürlich ohne auf Nutzerebene explizit Pointer und Speicher-Allokations-Operationen zu verwenden.

  Im Lichte statischer Typkonzepte wie im klassischen Pascal, das für diese Zwecke nur Pointer kennt, klingt eine solche Anforderung auf den ersten Blick sehr anspruchsvoll. Wer jedoch über einige Erfahrung im objektorientierten Programmieren dynamischer Strukturen verfügt, weiß, daß dynamische Datentypen nichts ungewöhnliches sind.

- Es ist ein Konzept für das Zusammenwirken der verschiedenen *Bibliotheken* mit dem Systemkern zu entwickeln, um das in ihnen gespeicherte mathematisch-algorithmische Wissen zu aktivieren.

  Diese in Form von black-box-Wissen vorhandene Kompetenz ist die Kernkompetenz des Systems. Die Implementierung dieser Algorithmen baut wesentlich auf andere Teile des Designkonzepts auf, die damit darüber entscheiden, wie effektiv Implementierungen überhaupt sein können.

  Die Anzahl und die Komplexität der eingebauten Funktionen ist mit klassischen Programmiersprachen nicht vergleichbar.

- Schließlich ist ein Konzept für den *Simplifikator* zu entwickeln, mit dessen Hilfe Ausdrücke gezielt in zueinander äquivalente Formen nach unterschiedlichen Gesichtspunkten umgeformt werden können.

Als Minimalanforderung muß dieser Simplifikator wenigstens in der Lage sein, in gewissem Umfang die semantische Gleichwertigkeit syntaktisch unterschiedlicher Ausdrücke festzustellen.

Dabei handelt es sich meist um ein zweistufiges System, das aus einer *Polynomarithmetik* besteht, die in der Lage ist, rationale Ausdrücke umzuformen, und einem *Simplifikationssystem*, das die Navigation in der transitiven Hülle der dem System bekannten elementaren Umformungsregeln gestattet.

3.3 Variablen und Ausdrücke im symbolischen Rechnen

Wir wollen uns in diesem Abschnitt die wichtigsten Begriffe und Prinzipien einer imperativen Programmiersprache ins Gedächtnis rufen und auf dieser Basis die Besonderheiten des symbolischen Rechnens herausarbeiten.

Unter einem Programm versteht man im allgemeinen ([4])

> *die schrittweise Transformation einer Menge von* **Eingabedaten** *in eine Menge von* **Ausgabedaten** *nach einem vorgegebenen* **Algorithmus**.

Diese Transformation erfolgt durch die schrittweise Veränderung des Programmstatus, der den Zustand der Gesamtheit der durch das Programm manipulierten Daten erfaßt. Diese Daten unterteilt man in

- Werte von Variablen (transiente Daten) und
- Dateiinhalte (persistente Daten),

je nachdem, ob die Änderung der Daten das Programmende überlebt oder nicht.

Eine iterative Programmiersprache wie Pascal oder C betrachtet ein Programm als eine Folge von Anweisungen und Deklarationen, wobei Anweisungen den Programmstatus ändern, Deklarationen dagegen nicht, sondern lediglich Bedeutungen von Bezeichnern festlegen.

Man unterscheidet gewöhnlich Deklarationen von *Datentypen*, *Variablen*, *Funktionen* und *Prozeduren*.

Anweisungen setzen sich aus elementaren Anweisungen zusammen, die durch entsprechende *Steuerstrukturen* miteinander verbunden sind. Elementare Anweisungen können *Zuweisungen* oder *Prozeduraufrufe* sein. Durch erstere wird direkt der Wert einer Variablen geändert, durch zweitere erfolgt die Änderung des Programmstatus als Seiteneffekt.

In Anweisungen spielen Variablen und Ausdrücke eine Rolle. Ausdrücke sind dabei durch ihren *Wert* und *Typ* charakterisiert, Variablen weiterhin durch ihren *Bezeichner* und ihre *Speicherplatzadresse*.

Ausdrücke

Im Zusammenhang mit Anweisungen und Prozedur- bzw. Funktionsaufrufen spielt der Begriff des Ausdrucks eine zentrale Rolle. Ein solcher (zulässiger) Ausdruck tritt sowohl als rechte Seite einer Zuweisung als auch als Aufrufparameter in Erscheinung.

Der Begriff des zulässigen Ausdrucks wird dabei rekursiv definiert. In Sprachen wie Pascal oder C versteht man darunter eine nach bestimmten Regeln zusammengesetzte Zeichenkette, in der neben Konstanten, Variablenbezeichnern und Funktionsaufrufen auch noch verschiedene *Operationszeichen* auftreten. So sind etwa $a+b$ oder $b-c$ ebenso zulässige Ausdrücke wie $a+gcd(b,c)$, wenn a,b,c als `integer`-Variablen und `gcd` als zweistellige Funktion `gcd:(int,int)` $\mapsto$ `int` vereinbart wurden.

Solche zweistelligen Operatoren unterscheiden sich allerdings nur durch ihre spezielle Notation von zweistelligen Funktionen. Man bezeichnet sie als *Infix-Operatoren* im Gegensatz zu der gewöhnlichen Funktionsnotation als *Präfix-Operator*. Intern werden die Darstellungen in einer Präcompilierphase sofort in die entsprechende Präfix-Notation umgewandelt, so daß die obigen Ausdrücke in der Form `+(a,b)`, `-(b,c)`, `+(a,gcd(b,c))` erscheinen. Dabei sind eine Reihe weiterer Besonderheiten wie Ausdrücke der Form $a + b + c$ aufzulösen. Bei dieser Kurzschreibweise wird implizit die Tatsache verwendet, daß die Addition assoziativ ist. Um eine solche Summe aufzulösen, sind entsprechende Klammern zu setzen, um festzulegen, in welcher Reihenfolge die Addition auszuführen ist. Vereinbart man zum Beispiel, daß der Operator `+` linksassoziativ ist, so wird $a + b + c$ als $(a + b) + c$ berechnet. Ähnliche Fragen treten für Ausdrücke der Gestalt $a + b * c$ auf, bei deren Auswertung zu berücksichtigen ist, daß Punkt- vor Strichrechnung geht, d.h. die Multiplikation eine größere Bindungskraft hat als die Addition. Diese Tatsache wird gewöhnlich durch die Festlegung von *Vorrangzahlen* für die einzelnen Operatoren berücksichtigt, wobei Operatoren mit einer höheren Vorrangzahl zuerst angewendet werden.

Wir sehen also, daß Ausdrücke für einen klassischen Compiler (und Interpreter) einer imperativen Programmiersprache nach der Auflösung dieser Diversitäten ausschließlich aus einer rekursiven Schachtelung von Funktionsaufrufen bestehen, in die auf der untersten Ebene Konstanten und Variablen eingehen. Von Seiteneffekten einmal abgesehen (reference by name), ist der Unterschied zwischen beiden unerheblich, da jeweils ausschließlich der *Wert* in die Berechnung des entsprechenden Funktionswerts eingeht.

Mehr noch kann man auch eine Zuweisung als einen Funktionsaufruf mit Seiteneffekt verstehen. Die Mehrfachzuweisung `a=b=c` in der Sprache C ist genau nach diesem Prinzip aufgebaut. Der Zuweisungsoperator ist hier ein rechtsassoziativer Operator. Die Anweisung wird als `(a=(b=c))` verstanden und zu `=(a,=(b,c))` umgesetzt, wobei die innere Zuweisung als Nebeneffekt b den Wert von c zuweist und diesen Wert zugleich zurückgibt, der dann in der äußeren Zuweisung a als Wert zugewiesen wird.

In dieser Phase der Syntaxanalyse wird durch den Compiler einer klassischen Programmiersprache, wie wir bereits festgestellt hatten, symbolische Information analysiert. Die dabei entwickelten Konzepte sollten sich also auch auf die Ausdrucksanalyse in symbolischen Programmsystemen übertragen lassen.

Datentypen und Polymorphie

Um eine Zuweisung auszuführen oder einen Funktionsaufruf abzuarbeiten, sind die eingehenden Ausdrücke *auszuwerten*. Eine Besonderheit, die sich bereits in klassischen Programmiersprachen findet, ist dabei die *Polymorphie*, d.h. die unterschiedliche Bedeutung von Funktions- und insbesondere Operatorsymbolen in Abhängigkeit von der Art der Argumente. So sind zur Berechnung von $a + b$ für ganze und reelle Zahlen unterschiedliche Verfahren aufzurufen. Mehr noch sind Compiler in der Lage, etwa bei der Addition einer ganzen und einer reellen Zahl selbständig eine passende Typkonversion auszuführen.

Eine solche Polymorphie ist in der Mathematik weitverbreitet und hat inzwischen auch mit dem Paradigma der Objektorientierung in die Informatik Einzug gehalten. Grundlage der Polymporphie ist hier die Möglichkeit, Referenzen auf denselben Funktionsnamen an Hand der *Typinformationen*, die die Parameter des jeweiligen Funktionsaufrufes tragen, aufzulösen.

Untersuchen wir also zuerst einmal, welche Ansprüche an ein Typsystem im symbolischen Rechnen zu stellen sind. Datentypen sind (nach [4]) ein

> *grundlegender Begriff der Informatik, unter dem man die Zusammenfassung von Wertebereichen und Operationen zu einer Einheit*

versteht. Betrachen wir etwa den Ausdruck $f := 2 * x + 3$, so müßte ein passender Datentyp `Polynomial` die für Polynome übliche Signatur zur Verfügung stellen. Da Polynomoperationen sich

aus Operationen auf Koeffizienten und auf Termen zusammensetzen, müßte ein qualifiziertes Typsystem wenigstens die unterschiedlichen Koeffizientenbereiche berücksichtigen. In einer Sprache wie Pascal oder bzw. C++ müßten wir also unterscheiden zwischen

```
class IntegerPolynomial {
    int coeff;
    Term exp; ...
}
```

und

```
class FloatPolynomial {
    float coeff;
    Term exp; ...
}
```

da diese unterschiedliche Additionen und Multiplikationen der Koeffizienten verwenden. Da jedoch viele andere Koeffizientenbereiche wie etwa rationale Zahlen, komplexe Zahlen und selbst Matrizen gelegentlich benötigt werden, ist es sinnvoller, Polynome als einen *parametrisierten Datentyp* anzulegen. `Polynomial(R)` konstruiert dann als Datentyp Polynome mit Koeffizienten aus dem Bereich R, für den selbst die entsprechenden Ringoperationen definiert sein müssen. Im Gegensatz zu Templates in C++,

```
<template R>
class Polynomial {
    R coeff;
    Term exp; ...
}
```

die für verschiedene Parameterwerte R vollständige Kopien der Implementierung des Datentyps anlegen, ist es allerdings sinnvoller, die Auflösung der Referenzen auf die Koeffizientenoperationen über den Aufrufparameter R vorzunehmen, der also ein *abstrakter Datentyp* sein müßte. Neben der Reduktion des entsprechenden Codes erlaubt ein solches Vorgehen auch, Polynomringe *dynamisch* zu erzeugen, d.h. als Koeffizienten Bereiche zu verwenden, an die zur Compilezeit des Polynomcodes nicht gedacht worden ist bzw. die vielleicht noch nicht einmal zur Verfügung standen.

> Bereits diese einfache Überlegung zu Datentypkonzepten im symbolischen Rechnen führt uns also unvermittelt an die vorderste Front der Entwicklung programmiersprachlicher Instrumente.

Auch die Inferenz von Datentypen birgt erhebliche Probleme in sich. Um etwa den Ausdruck $f := 2 * x + 3$ als `Polynomial(Integer)` zu interpretieren, ist in einem ersten Schritt $2 * x$ zu berechnen, d.h. das Produkt aus `2:Integer` und `x:String` zu bilden. Dazu muß der gemeinsame Oberbereich erst gefunden werden, in den beide Datentypen transformiert werden können, damit die Multiplikation ausgeführt werden kann. Noch komplizierter wird es bei der Berechnung von $f+g$ mit $g = 2/3$. Für g ergibt sich bei entsprechender Analyse der Datentyp `Fraction(Integer)`, für f dagegen `Polynomial(Integer)`. Das Ergebnis könnte entweder die Einbettung `Integer`$\subset$ `Polynomial(Integer)` verwenden und damit den Typ `Fraction(Polynomial(Integer))` einer rationalen Funktion oder aber die Einbettung `Integer`$\subset$ `Fraction(Integer)` verwenden und damit den Typ `Polynomial(Fraction(Integer))` eines Polynoms mit rationalen Koeffizienten haben. Beiden Fällen ist gemein, daß der entsprechende Obertyp, in den eine Typkonversion erfolgen muß, aus einer Obertyprelation des Parameters zu inferieren ist.

> Diese kleine Liste von Fragestellungen mögen als Begründung dienen, weshalb CAS der zweiten Generation auf ein konsistentes Typsystem im Sinne der Theorie der Programmiersprachen weitestgehend verzichten und Typinformationen nur insoweit verwenden, wie sie sich aus der *syntaktischen Struktur* der entsprechenden Ausdrücke extrahieren lassen.

Die Liste als zentrale Datenstruktur des symbolischen Rechnens

Wir wollen nun untersuchen, wie Ausdrücke in Maple und Reduce intern dargestellt werden. Die folgenden Prozeduren gestatten es, diese innere Struktur sichtbar zu machen.

Maple: `structure` extrahiert die oberste Ebene, `totalstructure` stellt die gesamte rekursive Struktur der Ausdrücke dar.

```
structure:=proc(u) op(0..nops(u),u) end;

totalstructure := proc(u) local i;
    if nops(u) = 1 and not type(u, function) then u
    else [seq(totalstructure(op(i, u)), i = 0 .. nops(u))]
    fi
end;
```

Reduce: `struct` stellt die gesamte rekursive Struktur des entsprechenden Ausdrucks dar.

```
procedure struct(u); lisp prettyprint u;
```

Beispiele Maple:

```
f:=(x+y)^5;
f:=expand(f);
f:=1/2;
f:=1/x;
f:=[a,b,c]; # Liste
f:={a,b,c}; # Menge
f:=x;
f:=4;
f:=sin(x);

structure(f);
totalstructure(f);

f:=matrix(2,2,[[1,2],[3,4]]); # Verwende structure(evalm(f));
# totalstructure liefert einen Fehler.

f:=matrix(2,2,[[x-1,x+1],[x+1,x-1]]);

f:=sin(x)^3+cos(x)^3;
f:=sin(x^2+2*cos(y+z));
```

Beispiele Reduce:

```
f:=(x+y)^5;
f:=sin(x);
f:=1/2;
f:=1/x;
f:={a,b,c};

m1:=mat((1,2),(3,4));
m2:=mat((x-1,x+1),(x+1,x-1));

f:=sin(x)^3+cos(x)^3;
f:=sin(x^2+2*cos(y+z));
```

Wir sehen, daß in beiden Systemen die interne Darstellung symbolischer Daten durch geschachtelte Listen erfolgt, die sich eng an der für Compiler zur Syntaxanalyse üblichen Darstellung durch *Ableitungsbäume* orientiert. Allerdings steht das entsprechende Funktionssymbol nicht in der Wurzel den entsprechenden Teilbaums, sondern als erstes Element der Liste. Eine solche

Darstellung von Funktionsaufrufen durch (geschachtelte) Listen, in denen das erste Listenelement den Funktionsnamen und die restlichen Elemente die Parameterliste darstellen,

ist typisch für die Sprache LISP, die an der Wurzel aller modernen CAS steht. CAS verwenden das erste Listenelement darüber hinaus auch zur Unterscheidung von einfachen Datenstrukturen (Listen, Mengen, Matrizen) sowie zur Speicherung von Typangaben atomarer Daten.

Ein solches Datendesign erlaubt eine hochgradig homogene Datenrepräsentation, denn jedes Listenelement besteht aus einem Pointerpaar ("dotted pair"), wobei der erste Pointer auf das entsprechende Listenelement, der zweite auf die Restliste zeigt. Nur auf der Ebene der Blätter tritt eine überschaubare Zahl anderer (atomarer) Datenstrukturen wie ganze Zahlen, Floatzahlen oder Strings auf. Eine solche homogene Datenstruktur erlaubt trotz der sehr unterschiedlichen Größe der verschiedenen symbolischen Daten die effektive dynamische Allokation und Reallokation von Speicher, da einzelne Zellen jeweils dieselbe Größe haben.

Listen als abstrakte Datentypen werden dabei allerdings anders verstanden als in den meisten Grundkursen "Algorithmen und Datenstrukturen" (etwa [12] oder [19]), in denen eine durch `Einfügen` und `Entfernen` *destruktiv veränderbare Listenstruktur*, da Ausdrücke gemeinsame Teilstrukturen besitzen können und deshalb ein destruktives Listenmanagement zu unüberschaubaren Seiteneffekten führen würde. Stattdessen werden Listen als *rekursiv konstruierbare Objekte* betrachtet mit Zugriffsoperatoren `first` (erstes Listenelement) und `rest` (Restliste) sowie dem Konstruktor `cons`, der ein neues Pointerpaar erstellt, mit dem (ein Pointer auf) ein Listenelement und (ein Pointer auf) eine Restliste zu einer neuen Liste zusammengefügt werden. Dieses für die Sprache LISP typische Vorgehen erlaubt es, auf aufwendiges Duplizieren von Teilausdrücken zugunsten des Duplizierens von Pointern zu verzichten, indem beim Kopieren einer Liste die Referenzen übernommen, die Links aber kopiert werden. Diese Sprachelemente prädestinieren einen funktionalen und rekursiven Umgang mit Listen, der deshalb in LISP und damit in CAS weit verbreitet ist. Für einen aus einer imperativen Welt kommenden Nutzer ist das gewöhnungsbedürftig.

Variablen

Wir hatten bereits gesehen, daß die Analyse symbolischer Ausdrücke *zur Laufzeit*, die ja im Mittelpunkt des Interpreters eines CAS steht, weniger Ähnlichkeiten zu einer klassischen Programmiersprache hat als vielmehr zu den Techniken, die ein Compiler oder Interpreter derselben verwendet.

Das trifft insbesondere auf das Variablenkonzept zu, in dem statt der klassischen Bestandteile *Bezeichner, Wert, Speicherbereich* und *Datentyp*, von denen in symbolischen Umgebungen sowieso nur die ersten beiden eine Bedeutung besitzen, neben dem Bezeichner *verschiedenartige* symbolische Informationen eine Rolle spielen, die wir als **Eigenschaften** dieses Bezeichners zusammenfassen wollen.

Die entscheidende Besonderheit in der Verwendung von Variablen in einem symbolischen Kontext besteht aber darin, daß sich **Namensraum und Wertebereich überlappen.** In der Tat ist in einem Ausdruck wie $x^2 + y$ nicht klar, ob etwa x hier als *Symbol* für sich selbst steht oder als *Variable* Container eines anderen Werts ist. Mehr noch kann sich die Bedeutung je nach Kontext unterscheiden. So wird etwa in einem Aufruf

```
> u:=int(1/(sin(x)*cos(x)-1),x);
```

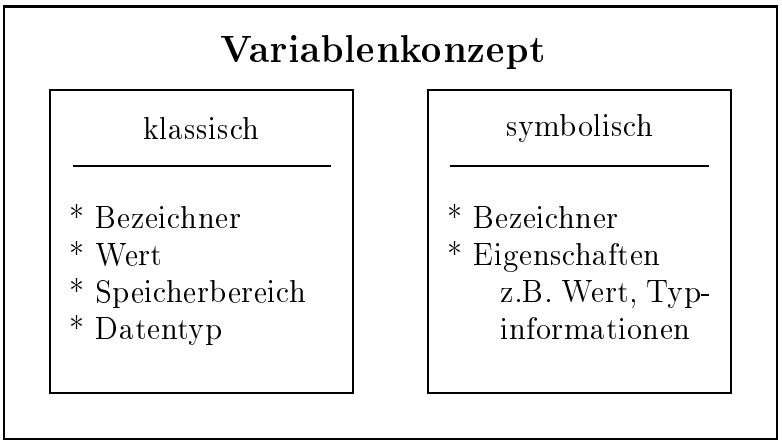

Bild 3: Das Variablenkonzept im Vergleich

x symbolisch gemeint sein, selbst wenn diese Variable bereits einen Wert besitzt.

Sehen wir uns diese Besonderheit des Variablenkonzepts zunächst in einigen Maple-Beispielen an.

```
p:=9*x^3-37*x^2+47*x-19;
                                   3       2
                          p := 9 x  - 37 x  + 47 x - 19

x:=2;
p;
                                        -1


x:=unknown;
p;
                             3             2
                  9 unknown  - 37 unknown  + 47 unknown - 19

x:=x;
                                  x := unknown

# Es wurde der Wert der Variablen x verwendet.

x:='x';
                                     x := x

# Es wurde das Symbol x verwendet.

p;
                               3       2
                            9 x  - 37 x  + 47 x - 19
```

Wir sehen, daß aus einem Symbol automatisch durch eine Wertzuweisung eine (globale) Variable wird. Die *Deklaration* einer solchen globalen Variablen ist nicht notwendig, da ja kein Typ vereinbart

werden muß.

Variablen werden in klassischen Compilern und Interpretern in einer *Symboltabelle* erfaßt, um sie an allen Stellen, an denen sie im Quellcode auftreten, in einheitlicher Weise zu verarbeiten. Für jede Variable existiert **genau** ein Eintrag in der Tabelle, unter dem auch weitere Informationen über *Eigenschaften* der Variablen, wie ihr Datentyp, ihre Speicheradresse, ihr Auftreten in lokalen Kontexten usw. vermerkt sind.

Da in einem CAS jedes in irgendeinem Ausdruck auftretende Symbol später als Variable verwendet werden kann, muß dieser *Mechanismus der eindeutigen Referenz* in einem solchen Kontext auf **alle** auftretenden Symbole ausgedehnt werden. Dementsprechend wird bei der Analyse der einzelnen Eingaben jeder String, der ein Symbol darstellen könnte, daraufhin geprüft, ob er bereits an anderer Stelle aufgetreten ist und in diesem Fall eine Referenz an die entsprechende Stelle der Symboltabelle eingetragen. Andernfalls ist die Symboltabelle um einen neuen Eintrag zu erweitern.

CAS unterscheiden damit nicht zwischen Symbolen und Variablen.

Bei manchem der Systeme kann man sich diese Symboltabelle ganz oder teilweise anzeigen lassen. Maple verfügt sogar über zwei solche Funktionen. `unames()` listet alle bereits verwendeten Symbole auf, die keinen Wert besitzen (unassigned names), `anames()` die Symbole mit "Wert", die also Variablen im klassischen Sinne entsprechen[1].

Betrachten wir, wie sich diese beiden Mengen im Zuge von Wertzuweisungen ändern. Mit den folgenden Anweisungen kann man Maple zurücksetzen und sich damit den ursprünglichen Zustand beider Sequenzen ansehen:

```
restart: [unames()]; [anames()];
```

Eine ganze Reihe von Informationen, die beim Start von Maple geladen werden, liegen als Strings vor und sind deshalb nach dem beschriebenen Mechanismus in diese Tabelle aufgenommen worden. Die zweite Menge ist dagegen noch leer.

Betrachten wir einen Teil aus diesem Raum, die Namen der Länge 1, und beobachten, wie sich dieser Namensraum durch die Verwendung von Variablen verändert:

```
[anames()]; select(s->length(s)=1,[unames()]);
```

```
                                  []

                        [!, ., <, =, I, O, ^, s, x, y]
```

```
u; [anames()]; select(s->length(s)=1,[unames()]);
```

```
                                   u

                                  []

                       [!, ., <, =, I, O, ^, s, u, x, y]
```

```
u:=1; [anames()]; select(s->length(s)=1,[unames()]);
```

```
                                u := 1

                                [u]
```

[1] Die in der Dokumentation gegebene Spezifikation ist allerdings nicht ganz korrekt, da nicht alle Symbole, die einen Wert haben, angezeigt werden, wie man sich leicht überzeugt, wenn man sich etwa mit `anames(integer)` alle Symbole mit Integer-Werten anzeigen läßt.

```
[!, ., <, =, I, O, ^, s, x, y]

u:='u';[anames()]; select(s->length(s)=1,[unames()]);

                    u := u

                      []

[!, ., <, =, I, O, ^, s, u, x, y]
```

In der letzten Zeile ist `u` aus der Liste der Variablen wieder verschwunden, d.h. die angegebene Zuweisung "löscht" den Wert von `u`. Symbole können also einen Wert besitzen, müssen es aber nicht. Hilfsweise kann man sich vorstellen, daß Variablen, die keinen Wert haben, zu sich selbst auswerten, was aber nicht vollkommen korrekt ist. So ist etwa eine Zuweisung

```
> x:=x;
```

in den meisten Systemen nicht zulässig.

In CAS besitzt nicht jedes Symbol einen Wert. Variablen müssen deshalb nicht initialisiert werden.

Die Eigenschaft eines Symbols, einen Wert zu besitzen, kann im Laufe seiner Lebenszeit mehrfach wechseln. Der für das symbolische Rechnen typische Ausgangszustand kann in den Systemen auf unterschiedlichem Weg hergestellt werden:

AXIOM	)clear value x
MACSYMA	kill(x)
MAPLE	x:='x'
MATHEMATICA	Clear[x]
MuPAD	unassign(x)
REDUCE	clear x

Tabelle 4: Symbole "wertlos" setzen.

Der Wert eines Symbols wird ähnlich wie andere Eigenschaften, dem Symbol direkt zugeordnet. Dies kann man z.B. in Reduce verfolgen, indem man sich mit der Funktion `prop` die Eigenschaften des entsprechenden Symbols (in interner Notation) auflisten läßt:

```
1: prop 'x;

2: prop 'y;
```

Anfangs verfügen beide Symbole über keine Eigenschaften.

```
3: x:=2;

    x := 2

4: prop 'x;

    ((avalue scalar 2))
```

Der zugewiesene Wert wurde unter dem Symbol x als eine Eigenschaft `avalue` gespeichert.

```
6: x:=y^2+z;

          2
    x := y  + z

7: prop 'x;

    ((avalue scalar (!*sq ((((y . 2) . 1) ((z . 1) . 1)) . 1) t)))
```

Der geänderte Wert wurde auf dieselbe Weise, diesmal in einer bereits übersetzten Form, gespeichert.

```
8: prop 'y;

    (used!*)
```

Für y wurde zusätzlich die Information gespeichert, daß es als Symbol in einem anderen Ausdruck verwendet wurde.

```
9: clear x;

10: prop 'x;

11: prop 'y;

    (used!*)
```

Löscht man x, so ist die Wertzuweisung verschwunden, die Zusatzinformation für y bleibt dagegen erhalten.

Dieses einheitliche Management der Symbole führt dazu, daß auch zwischen Variablenbezeichnern und Funktionsbezeichnern nicht unterschieden wird. Wir hatten beim Auflisten der dem System bekannten Symbole bereits an verschiedenen Stellen Funktionsbezeichner in trauter Eintracht neben Variablenbezeichnern stehen sehen.

CAS unterscheiden nicht zwischen Variablen- und Funktionsbezeichnern.

Da andererseits Funktionsdefinitionen ebenfalls symbolischer Natur sind und "nur" einer entsprechenden Interpretation bedürfen, verwenden einige Systeme wie etwa Maple sogar das Zuweisungssymbol, um Funktionsdefinitionen mit einem Symbol zu verbinden.

Eine solche einheitliche Behandlung interner und externer Symbole erlaubt es auch, Systemvariablen und selbst -funktionen zur Laufzeit zu überschreiben oder neue Funktionen zu erzeugen und (selbst in den compilierten Teil) einzubinden. Da dies zu unvorhersagbarem Systemverhalten führen kann, sind die meisten internen Funktionen allerdings vor Überschreiben geschützt. Hier einige Beispiele in Maple:

```
> Pi:=3.14;

Error, attempting to assign to `Pi` which is protected

> unprotect(Pi);
```

```
> Pi:=1;
                                    Pi := 1

> eval(arctan(1));
                                      1/4

> gcd:=2;

Error, attempting to assign to `gcd` which is protected

>  unprotect(gcd);
> gcd:=5;
                                    gcd := 5

> gcd(2,3);
                                       5

> gcd(12,13);
                                       5
```

Wir sehen, daß es Maple nach Entfernen des Überschreibschutzes selbst erlaubt, Funktionsnamen als Variablennamen zu verwenden. Die meisten CAS unterscheiden jedoch zwischen Wert- und Funktionsdefinitionen und ordnen sie als *unterschiedliche* Eigenschaften dem jeweiligen Symbol zu.

Auswerten von Ausdrücken

Die Tatsache, daß der Wert von Variablen symbolischer Natur sein kann, in den seinerseits Symbole eingehen können, die selbst einen Wert besitzen, führt zu einer weiteren Besonderheit von CAS. Betrachten wir die folgenden Zuweisungen in Maple

```
a:=b;b:=c;c:=3;
                                    a := b

                                    b := c

                                    c := 3
```

und sehen uns an, welche Wertzuweisungen für a vorgenommen worden ist:

```
a;
                                       3
```

Es ist also die gesamte Evaluationskette durchlaufen worden: Der Wert von a ist das Symbol b, das seinerseits als Wert das Symbol c hat, welches den Wert 3 besitzt. Denkbar wäre auch eine eingeschränkte Evaluationstiefe, etwa nur Tiefe 1. In Maple kann man diese Tiefe mit speziellen Kommandos variieren:

```
seq(eval(a,i),i=1..6);
                                b, c, 3, 3, 3, 3

b:=d;
```

```
                                    b := d

a;
                                      d
```

Indem b ein anderer Wert zugewiesen wurde, ändert sich auch die Evaluationskette für a.

Die Variablen, die in den bisherigen Zuweisungen als rechte Seiten auftraten, hatten selbst noch keinen Wert. Bei Variablen mit Wert müssen wir unterscheiden, ob wir deren Wert oder aber das Symbol selbst meinen.

```
u:=3;a:=u;b:='u';
                                    u := 3

                                    a := 3

                                    b := u

a;b;
                                      3

                                      3
```

An dieser Stelle ist der Unterschied noch nicht zu erkennen. Betrachten wir jedoch

```
u:=5;a;b;
                                    u := 5

                                      3

                                      5
```

so erkennen wir, daß sich Änderungen des Werts von u auf b auswirken, auf a dagegen nicht.

```
seq(eval(a,i), i=1..5);
                                 3, 3, 3, 3, 3

seq(eval(b,i), i=1..5);
                                 u, 3, 3, 3, 3
```

Geht in einen Ausdruck eine Variable x durch ihren *Wert* ein, so spricht man von *früher Auswertung*, wird sie dagegen nur als *Symbol* verwendet, so spricht man von *später Auswertung*, da sich ein evtl. vorhandener Wert erst bei späteren Auswertungen auswirkt. CAS benötigen also Mechanismen, um Variablen in beiden Bedeutungen, Wert und Symbol, einzusetzen. Dafür gibt es zwei gängige Herangehensweisen: Variablen, die häufiger in ihrer symbolischen Gestalt benötigt werden, weist man durch einen **Substitutionsoperator** *lokal* einen Wert zu, der nur in dem entsprechenden Ausdruck wirksam ist. Für Variablen, denen global ein Wert zugewiesen worden ist, gibt es einen Schutzmechanismus, der sie vor der Auswertung bewahrt. Dies geschieht entweder wie in Maple durch eine spezielle *Hold*-Funktion oder wie in Axiom und Mathematica durch zwei Zuweisungsoperatoren, wovon einer die in die rechte Seite eingehenden Variablen auswertet (*frühe Auswertung*), der andere dagegen nicht (*verzögerte Auswertung*). Mathematica verfügt sogar über beide Mechanismen.

System	Substitutions-operator	frühe Zuweisung	späte Zuweisung	Hold-Operator
AXIOM	subst(f(x),x=a)	x:=a	x==a	–
MACSYMA	subst(x=a,f(x))	x:a	–	'x
MAPLE	subs(x=a,f(x))	x:=a	–	'x'
MATHEMATICA	f(x) $/.x- > a$	x=a	x:=a	Hold[x]
MuPAD	subs(f(x),x=a)	x:=a	–	hold(x)
REDUCE	subst(x=a,f(x))	x:=a	let x=a	–

Tabelle 5: Substitutions- und Zuweisungsoperatoren der verschiedenen Systeme

Diese Besonderheiten sind in einer klassischen Programmiersprache, in der Namensraum und Wertebereich getrennt sind, unbekannt.

Bei dem bisherigen Evaluierungsverfahren maximaler Tiefe kann es eintreten, daß ein zu evaluierendes Symbol nach endlich vielen Evaluationsschritten selbst wieder in dem entstehenden Ausdruck auftritt und damit der Evaluationsprozeß stets von Neuem durchlaufen wird wie in folgendem Beispiel (Maple, ähnlich MuPAD):

```
x:=y+1;
                                  x := y + 1

y:=x;
Warning, recursive definition of name
                                  y := y + 1

y;
Error, too many levels of recursion
```

Solche rekursiven Verkettungen können leicht eintreten, wenn man die Konsequenzen der getroffenen Wertzuweisungen nicht genau überblickt. Auch aus diesem Grund spielen lokale Wertzuweisungen, die nur innerhalb eines einzigen Aufrufs Gültigkeit haben, eine wichtige Rolle.

Reduce erlaubt eine solche rekursive Zuweisung nicht, sondern bricht mit einer Fehlermeldung ab:

```
x:=x+1;

***** x improperly defined in terms of itself
```

Mathematica besitzt eine interne Variable `$RecursionLimit`, die die Anzahl der Evaluationsschritte, die wirklich ausgeführt werden, begrenzt.

Die beiden Systeme Axiom und Macsyma verwenden standardmäßig nur Evaluationen der Tiefe 1, d.h. ersetzen standardmäßig nach dem ersten Evaluationsschritt auftretende Symbole nicht weiter durch ihre Werte, wodurch dieses Problem der rekursiven Wertzuweisung ebenfalls vermieden wird. Allerdings unterscheidet sich damit das Auswertungsverhalten dieser beiden Systeme von dem der anderen CAS.

Zuweisung	Axiom Macsyma	Maple Mathematica MuPAD Reduce
a:=b+1	b+1	b+1
b:=c+1	c+1	c+1
c:=d+1	d+1	d+1
a	b+1	d+3
a:=b+1	c+2	d+3

Tabelle 6: Unterschiedliches Auswertungsverhalten der verschiedenen Systeme

Funktionen, Funktionsausdrücke und Funktionssymbole

Eine weitere Besonderheit symbolischer Systeme besteht darin, daß neben

klassischen Funktionsaufrufen

und Prozeduren auch Funktionsausdrücke und sogar Funktionssymbole auftreten. So wird etwa in den folgenden Maple-Konstrukten das *Funktionssymbol* `sin` zur Konstruktion symbolischer Ausdrücke verwendet, die für Sinus-Funktionswerte stehen, die nicht weiter ausgewertet werden können. Solche Ausdrücke wollen wir *Funktionsausdrücke* nennen.

```
sin(x); sin(2);
                                  sin(x)

                                  sin(2)

# aber

sin(Pi/4);
                                       1/2
                                  1/2 2

sin(2.55);
                                 .5576837174

u:=f(x)+2*x+1;
                           u := f(x) + 2 x + 1

subs(x=2,u);
                                  f(2) + 5
```

Im zweiten Beispiel wurde sogar ein neues Funktionssymbol eingeführt und aus diesem zwei Funktionsausdrücke `f(x)` und `f(2)` konstruiert.

Solche Funktionssymbole sind dem System bereits bekannt oder können vom Nutzer vereinbart werden. Letztere werden (zunächst) wie unbekannte Funktionen behandelt. Dem System bekannte Funktionssymbole werden dagegen partiell wie klassische Funktionen behandelt.

Eine besondere Art von Funktionen sind solche wie die Maple-Funktionen `expand, collect, normal`, die Funktionsausdrücke transformieren, d.h. wie klassische Funktionen aussehen, jedoch im Ergebnis selbst Funktionssymbole enthalten. Oftmals kann man aber diese

Trennung in Funktionsaufrufe, Funktionssymbole und Funktionstransformationen

nicht sauber vornehmen, wie etwa das folgende Beispiel für die `exp`-Funktion in Maple zeigt:

```
exp(x);
```

```
exp(x)
```

```
exp(2.0);
```

```
7.389056099
```

```
exp(ln(x+y));
```

```
x + y
```

Im ersten Fall erhalten wir ein Ergebnis mit dem Funktions*symbol* `exp`, im zweiten Fall eine Float-Zahl, die mit einem entsprechenden Näherungsverfahren in einem Funktions*aufruf* berechnet wurde, im letzten Fall dagegen ist das Ergebnis eine Funktions*transformation*.

Wir hatten bereits gesehen, daß es aus der Sicht eines CAS keinen wesentlichen Unterschied zwischen Operatoren und Funktionen gibt, so daß wir ebenfalls zwischen *Operatorsymbolen* und *Operatoraufrufen* zu unterscheiden haben. So tritt etwa das Gleichheitszeichen bei der Formulierung eines Gleichungssystems sowie seiner Lösung als Operator*symbol* auf (Maple)

```
gls:={2*x+3*y=2, 3*x+2*y=1};
```

```
{3 x + 2 y = 1, 2 x + 3 y = 2}
```

```
solve(gls,{x,y});
```

```
{{x = -1/5, y = 4/5}}
```

Es ist zu beachten, daß in einem `while`- oder `if`-Konstrukt der den Ablauf steuernde logische Ausdruck nicht zu einem symbolischen Ausdruck auswerten darf, der verschieden von den Konstanten `true` oder `false` ist.

Die meisten Systeme formen einen symbolischen Ausdruck der Form $a = b$ nicht weiter um, sondern lassen ihn als *Gleichung* stehen, selbst wenn man ihm einen Wert `true` oder `false` zuordnen kann. Diese weitergehende Transformation wird aber in Kontexten, die einen booleschen Wert erfordern, ausgeführt.

```
MuPAD (aehnlich Reduce u.a.)
1=2;
```

```
1 = 2
```

```
# aber #
if 1=2 then yes else no end_if;
```

```
no
```

Funktionstransformationen

Funktionstransformationen, die man oft auch als Simplifikation wahrnimmt, sind eines der zentralen Designelemente eines CAS, da auf diesem Wege syntaktisch verschiedene, aber semantische gleichwertige Ausdrücke produziert werden können.

Funktionstransformationen sind Regeln, nach denen gewisse Kombinationen von Funktionssymbolen und evtl. speziellen Funktionsargumenten durch andere, semantisch gleichwertige Kombinationen ersetzt werden.

Signifikant für die Anwendbarkeit solcher Transformationsregeln ist das Aufeinandertreffen gewisser Funktionssymbole im Ableitungsbaum des entsprechenden Ausdrucks. Dieses kann darin bestehen, daß wie in obigem Beispiel das Nacheinanderausführen einer Funktion und ihrer Inversen durch die identische Funktion ersetzt wird oder daß wie in der Funktion `expand` Produkte distributiv durch Summen, trigonometrische Funktionen mit Mehrfachwinkeln durch solche mit Einfachwinkeln, Potenzen mit Exponentsummen nach den Potenzgesetzen durch die entsprechenden Basisprodukte usw. ersetzt werden.

```
expand((x+1)*(x+2));
                                   2
                                  x  + 3 x + 2

expand(sin(x+y));
                        sin(x) cos(y) + cos(x) sin(y)

expand(exp(a+sin(b)));
                               exp(a) exp(sin(b))
```

Funktionen können als Argumente beliebige symbolische Ausdrücke haben, also neben Variablen, rationalen Ausdrücken und Funktionsaufrufen auch Funktionsausdrücke wie in den folgenden Beispielen (Maple):

```
diff(f(x),x);
                                    d
                                    -- f(x)
                                    dx

# aber

diff(sin(x),x);
                                    cos(x)



diff(f(y(x)),x);
                                      /d      \
                             D(f)(y(x)) |-- y(x)|
                                      \dx     /
```

Im ersten Beispiel kann das Funktionssymbol `diff`, das als ein Argument den Funktionsausdruck $f(x)$ hat, nicht weiter vereinfacht werden. Die etwas andere Ausgabegestaltung hat nur mit der speziellen zweidimensionalen Ausgabeformatierung zu tun. Im zweiten Beispiel dagegen treffen `diff` und `sin` aufeinander, was zu `cos` vereinfacht wird. In Wirklichkeit wird dabei die Kettenregel angewendet, wie man am dritten Beispiel sieht.

Bei dieser Berechnung der Ableitung einer allgemeinen zusammengesetzten Funktion tritt im letzten Ergebnis mit dem Symbol D sogar eine Funktion auf, die keinen Funktions*ausdruck*, sondern ein reines Funktions*symbol* als Argument hat, weil an der Stelle die Ableitung der Funktion f an der Stelle $y(x)$ einzusetzen ist.

```
eval(subs(f=sin,"));
                                        /d      \
                                cos(y(x)) |-- y(x)|
                                        \dx     /

D(sin);
                                          cos

D(exp+ln);
                                  exp + (a -> 1/a)
```

Um die Antwort im letzten Beispiel zu formulieren, wurde eine weitere Funktion benötigt, von der nur die Zuordnungsvorschrift bekannt ist, die aber keinen Namen besitzt. Eine solche Funktion wird auch als *reine Funktion* (pure function) bezeichnet. Sie ist das Gegenteil eines Funktionssymbols, von dem umgekehrt der Name, aber keine Anwendungsvorschrift bekannt war.

> Schließlich muß das System der möglichen Funktionstransformationen zur Laufzeit erweiterbar sein.

Das ist notwendig, wenn es dem Nutzer gestattet werden soll, selbstdefinierte Funktionssymbole auf dieselbe Weise einzusetzen wie systemdefinierte. Dazu muß er etwa in die Lage versetzt werden, auch selbstdefinierte Paare Funktion/Inverse oder deren Ableitungen in den Transformationsmechanismus des Systems auf konsistente Weise einzubauen.

Funktionale und regelbasierte Transformationskonzepte

Ein erster Blick auf verschiedene Transformationen zeigt uns, daß hier ein Grundsatz der klassischen Funktionsauswertung verletzt ist: Es wird nicht nur der *Wert* der Aufrufargumente benötigt, sondern auch Information über deren *Struktur*.

So muß etwa beim Aufruf `expand(`sum_1`*`sum_2`)` die Funktion erkennen, daß ihr Argument ein Produkt ist, dessen Faktoren extrahieren und eine Funktion `expandproduct(`sum_1`,`sum_2`)` aufrufen, die alle paarweisen Produkte zwischen den einzelnen Summanden der beiden Summen bildet und diese danach aufsummiert.

> Charakteristisch für Transformationen ist also der Umstand, daß nicht nur ein Knoten des Ableitungsbaums zusammen mit den von ihm ausgehenden Kanten an der Bestimmung des Rückgabewerts beteiligt ist, sondern komplexere Teilstrukturen desselben.

Funktionstransformationen sind ihrer Natur nach *rekursiv,* da nach dem Anwenden einer Transformation ein Ausdruck entstehen kann, auf den weitere Transformationen angewendet werden können. So führt Maple bei der Berechnung von

```
cos(exp(ln(arcsin(x))));
                                      2 1/2
                                    (1 - x )
```

etwa erst die Transformation $exp(ln(u)) = u$ und dann $cos(arcsin(x)) = \sqrt{1-x^2}$ aus.

CAS verfolgen zur Realisierung solcher Funktionstransformationen zwei verschiedene Konzepte, ein *funktionales* (Maple, MuPAD) bzw. ein *regelbasiertes* (Axiom, Reduce). Macsyma und Mathematica stellen beide Mechanismen zur Verfügung.

Beim funktionalen Konzept versucht man, alle Funktionsinvokationen als Funktionsaufrufe zu gestalten und die Besonderheiten von Funktionstransformationen durch eine genaue Unterscheidung

zwischen (rekursiven) Funktionsaufrufen und Funktionssymbolen in den Griff zu bekommen. Da in die Abarbeitung der entsprechenden Funktion die *Struktur* der Argumente mit eingeht, wird dazu eine Funktion benötigt, die diese Struktur wenigstens teilweise analysiert. Maple verfügt für diesen Zweck über die Funktion `type(A,T)`, die prüft, ob ein Ausdruck A den "Typ" T hat. Wie bereits an anderer Stelle erwähnt handelt es sich dabei allerdings nicht um ein strenges Typkonzept, sondern um eine Typanalyse, die nur die entsprechende syntaktische Analyse, noch dazu nur der obersten Ebene, der Struktur des entsprechenden Ausdrucks verwendet. Dies erkennen wir etwa an folgendem Beispiel:

```
exp(ln(x));
                                        x

exp(ln(x)+ln(y));
                                exp(ln(x) + ln(y))

simplify(");
                                        x y
```

Die Struktur des Arguments verbirgt im zweiten Beispiel die Anwendbarkeit der Transformationsregel. Erst nach eingehenderer Analyse, die mit `simplify` angestoßen wird, gelingt die Umformung[2].

Betrachten wir als Beispiel, wie Maple die Exponentialfunktion definiert:

```
interface(verboseproc=2):print(`exp`);

 proc(x)
local i,t,q;
options `Copyright 1993 by Waterloo Maple Software`;
    if nargs <> 1 then ERROR(`expecting 1 argument, got `.nargs)
    elif type(x, 'complex(float)') then evalf('exp'(x))
    elif type(x, 'rational') then exp(x) := 'exp'(x)
    elif type(x, 'function') and op(0, x) = ln then exp(x) := op(1, x)
    elif type(x, 'function') and type(op(0, x), 'function') and
    op([0, 0], x) = `@` and op([0, 1], x) = ln then
        exp(x) := op([0, 2], x)(op(x))
    elif type(x, 'function') and type(op(0, x), 'function') and
    op([0, 0], x) = `@@` and op([0, 1], x) = ln then
        exp(x) := (ln@@(op([0, 2], x) - 1))(op(x))
    elif type(x, `*`) and type(- I*x/Pi, 'rational') then
        i := - I*x/Pi;
        if 1 < i or i <= -1 then
            t := trunc(i);
            t := t + irem(t, 2);
            exp(x) := exp(I*(i - t)*Pi)
        elif type(6*i, 'integer') or type(4*i, 'integer') then
            exp(x) := cos(- I*x) + I*sin(- I*x)
        else exp(x) := 'exp'(x)
        fi
    elif type(x, `*`) and nops(x) = 2 and type(op(1, x), 'rational') and
```

[2] Dies gelingt auch mit `expand`, was in diesem Fall Potenzgesetze anwendet, um Exponenten*summen* zu ersetzen. MuPAD 1.3. war nur auf diese Weise in der Lage, die genannte Vereinfachung vorzunehmen.

```
    type(op(2, x), 'function') and op([2, 0], x) = ln then
        if type(op(1, x), fraction) and irem(op([1, 2], x), 2, 'q') = 0
        then exp(x) := sqrt(op([2, 1], x))^(op([1, 1], x)/q)
        else exp(x) := op([2, 1], x)^op(1, x)
        fi
    else exp(x) := 'exp'(x)
    fi
end
```

Die meisten Zeilen beginnen mit `type(x, ...)`, analysieren also zuerst die Struktur des aufrufenden Arguments. Sehen wir uns die einzelnen Zeilen nacheinander an.

```
    elif type(x, 'complex(float)') then evalf('exp'(x))
```

Handelt es sich um eine Float-Zahl, wird `evalf`, das numerische Interface, mit dem Argument `'exp'(x)` aufgerufen. Die Quotes stehen für die fehlende Auswertung, d.h. an dieser Stelle wird das Funktions*symbol* samt Argument, also der *Funktionsausdruck* $\exp(x)$ an `evalf` übergeben. Dasselbe gilt in der nächsten Zeile

```
    elif type(x, 'rational') then exp(x) := 'exp'(x)
```

wenn das Argument x nämlich zu einer (ganzen oder) rationalen *Zahl* auswertet. In diesem Fall kann man den Ausdruck nicht weiter vereinfachen und es wird der entsprechende *Funktionsausdruck* zurückgeben:

```
exp(1/2);
                                    exp(1/2)

exp(27/3);
                                     exp(9)
```

Am letzten Beispiel sehen wir noch einmal, daß das Argument vor dem Aufruf von `exp` wirklich ausgewertet wurde. Im dritten else-Zweig

```
    elif type(x, 'function') and op(0, x) = ln then exp(x) := op(1, x)
```

wird schließlich die von uns betrachtete Transformation vorgenommen: Ist x ein Funktionsausdruck, der mit `ln` beginnt, so wird das (erste und einzige) Argument dieses Funktionsausdrucks zurückgegeben. Die folgenden Zeilen dienen der Analyse von Ausdrücken, die den Funktionsiterationsoperator `@` verwenden. Wir wollen sie übergehen. Interessant ist noch die Zeilen

```
    elif type(x, `*`) and type(- I*x/Pi, 'rational') then
        i := - I*x/Pi;
        if 1 < i or i <= -1 then
            t := trunc(i);
            t := t + irem(t, 2);
            exp(x) := exp(I*(i - t)*Pi)
        elif type(6*i, 'integer') or type(4*i, 'integer') then
            exp(x) := cos(- I*x) + I*sin(- I*x)
        else exp(x) := 'exp'(x)
        fi
```

in der geprüft wird, ob es sich vielleicht um einen Ausdruck $e^{I\pi i}$ handelt, den man bekanntlich in die Form $cos(\pi i) + I\,sin(\pi i)$ umformen kann. Wenn nicht $-1 < i \leq 1$ gilt, wird `exp` mit einem entsprechend adjustierten Argument rekursiv aufgerufen, ansonsten geprüft, ob i ein Vielfaches von

1/6 oder 1/4 ist. Für die entsprechenden Winkelgrößen sind die Funktionswerte $\sin(x)$ und $\cos(x)$ bekannt und die entsprechende Transformation wird durchgeführt, ansonsten wird der Funktionsausdruck in unveränderter Form zurückgegeben.

Funktionstransformationen sind in einem solchen Konzept manchmal nur über Umwege zu realisieren, wenn Funktions*aufrufe*, also Code algorithmischer Natur, abzuarbeiten sind, da hierbei die Aufrufargumente ausgewertet werden, wobei deren ursprüngliche Struktur verändert werden kann. Möchte man etwa ein Polynom $f \in \mathbf{Z}[x]$ nicht über den ganzen Zahlen, sondern modulo einer Primzahl p, also im Ring $\mathbf{Z}/p\mathbf{Z}[x]$, faktorisieren, so führen in Maple weder

```
> factor(f) mod p;
```

noch

```
> factor(f mod p);
```

zum Ziel, denn `factor` ist ein Funktions*aufruf*, der als Ergebnis ein Produkt, die Faktorzerlegung des Arguments über den ganzen Zahlen, zurückliefert. Im ersten Fall wird vor dem Aufruf von `mod` das Polynom (über den ganzen Zahlen) faktorisiert, im zweiten Fall dagegen das Polynom f erst modulo p reduziert und dann (wiederum ganzzahlig) faktorisiert. Nur wenn man beide Funktionen `mod` und `factor` im Zusammenhang betrachtet, kann man überhaupt die Frage korrekt formulieren. Dieser Zusammenhang wird aber aufgehoben, wenn durch einen Funktions*aufruf* bei Argumentauswertung einer der beiden Funktionsnamen bereits "verarbeitet" ist. In einem klassischen Ansatz würde man in diesem Fall auf Polymorphie oder, in unserem typlosen Ansatz, auf verschiedene Funktionsnamen, etwa `factor` und `factormod`, zurückgreifen. Um dies wenigstens in dem Teil, der dem Nutzer sichtbar ist, zu vermeiden, führt Maple ein Funktions*symbol* `Factor` ein, das sein Argument unverändert zurückgibt, so daß `mod` im Aufruf

```
> Factor(f) mod p;
```

der intern sofort als `mod(Factor(f),p)` umgesetzt wird, an der speziellen Form seines ersten Arguments erkennen kann, daß modular zu faktorisieren ist und die spezielle modulare Faktorisierungsroutine `` `mod/factor`(f,p) `` aufrufen kann[3].

Ähnlich geht die Maple-Funktion `expand` vor. `expand(A)` analysiert die Gestalt des Ausdrucks $A = f(x_1, \ldots, x_n)$ und ruft eine entsprechende Funktion

$$\texttt{`expand/f`}(x_1, \ldots, x_n)$$

auf, die die eigentliche Transformation vornimmt. Hier wird bereits die Fähigkeit eines CAS zur Stringmanipulation verwendet, indem zur Laufzeit (!), während des Aufrufs von `expand`, aus einem in den Argumenten vorkommenden Funktionssymbol `f` ein neuer Funktionsname `expand/f` und, sofern eine entsprechende Funktion definiert ist, auch der zugehörige Funktionsaufruf generiert wird. Dies ist zugleich der Mechanismus, der es einem Nutzer erlaubt, die Fähigkeiten des Systems zu "Expandieren" für selbstdefinierte Funktionen zu erweitern.

Soll etwa l eine additive Funktion darstellen, für die `expand` die Transformation $l(x + y) = l(x) + l(y)$ ausführt, so muß man dazu eine Funktion

```
`expand/l`:= proc(y) local x;
   x:=expand(y);
   if type(x,`+`) then convert(map(l,[op(x)]),`+`)
   else l(x) fi
end;
```

[3] Die Interna sind im Systemkern verborgen.

definieren.

Einen anderen Zugang verfolgt MuPAD: Hier wird jedem Funktionssymbol, auf dem `expand` nichttrivial agieren soll, als Eigenschaft ein Funktionsattribut mit dem Namen "expand" zugeordnet, das eine entsprechende Funktion trägt. Obigen Effekt für l kann man folgendermaßen erreichen:

```
l:= funcattr(l, "expand",
  proc(x) begin x:= expand(x);
       if type(x) = "_plus" then _plus(map(op(x),l))
       else l(x)
       end_if
  end_proc);
```

Die Nachteile dieses Konzepts fallen sofort ins Auge:

1. Man hat mit jedem Funktionsaufruf eine Menge verschiedener Typinformationen zu ermitteln, womit jeder Funktionsaufruf relativ aufwendige Operationen anstößt. Deshalb ist es oft sinnvoll, bereits berechnete Funktionswerte zu speichern, wenn man weiß, daß sie sich nicht verändern.

2. Die Typanalyse ist bei vielen Funktionen von ähnlicher Bauart, so daß unnötig Code dupliziert wird.

3. Die so definierten Transformationsregeln sind relativ "starr". Möchte man z.B. das Verhalten der `exp`-Funktion dahingehend ändern, daß sie bei rein imaginärem Argument *immer* die trigonometrische Darstellung verwendet, so müßte man den gesamten oben gegebenen Code kopieren, an den entsprechenden Stellen ändern und dann die (natürlich außerdem vor Überschreiben geschützte) `exp`-Funktion entsprechend neu definieren.

Deshalb hat eine Funktionsdefinition eine komplexere Struktur als in einer klassischen Programmiersprache. In Maple etwa kann eine Funktion eine *Funktionswert-Tabelle* anlegen, in die alle bereits berechneten Funktionswerte eingetragen werden. Diese wird inspiziert, bevor die Auswertung entsprechend der Funktionsdefinition initiiert wird. In MuPAD kann eine Funktionsdefinition außerdem noch über eine Tabelle von Funktionsattributen verfügen.

Beim regelbasierten Zugang werden die Funktionstransformationen über einen allgemeinen Simplifikationsmechanismus als direkte Transformationen des Ableitungsbaums ausgeführt. Dazu wird ein Satz von *Transformationsregeln* der Gestalt `lhs => rhs` auf den aktuellen Ausdruck A anwendet, indem grob gesprochen im Ableitungsbaum von A Teilbäume der Gestalt *lhs* gesucht und durch Teilbäume der Gestalt *rhs* ersetzt werden. Das wird so lange wiederholt, bis keine Ersetzungen mehr möglich sind. Im Gegensatz zum funktionalen Zugang sind hier also der Simplifikator und die jeweils anzuwendenden Regelsätze voneinander getrennt, was es auf einfache Weise ermöglicht, Regelsätze zu ergänzen und für spezielle Zwecke zu modifizieren und anzupassen.

Wir werden uns in einem späteren Kapitel genauer mit der Funktionsweise eines solchen Regelsystems vertraut machen. An dieser Stelle wollen wir uns nur einmal anschauen, welche Regeln Reduce zum Simplifizieren verschiedener Funktionen kennt. Die jeweiligen Regeln sind unter dem Funktionssymbol als Liste gespeichert und können mit der Funktion `showrules` ausgegeben werden:

```
showrules exp;

              x
{exp(~x) => e }

showrules log;

{log(1) => 0,
```

```
 log(e) => 1,

      ~x
 log(e  ) => x,

                      1
 df(log(~x),~x) => ---}
                      x
```

Zum Verständnis der ersten Regel muß hinzugefügt werden, daß Reduce die Exponentialfunktion damit als spezielle Potenzfunktion auffaßt und Potenzfunktionen wie die anderen Arithmetikoperatoren aus Effizienzgründen außerhalb des Regelsystems "fest verdrahtet" sind. Wie bei Funktionen ist auch bei Regeln zwischen Deklaration und Anwendung zu unterscheiden. Dabei ist insbesondere auch die Verwendung von formalen Parametern denkbar, die als Platzhalter für beliebige Teilausdrücke, d.h. für entsprechende Teilbäume in den Regeln auftreten. So ist etwa $log(exp(A)) = A$, egal wie der Teilausdruck A beschaffen ist. Solche formalen Parameter werden in Reduce durch eine vorangestellte Tilde gekennzeichnet und sind bei einer Anwendung an *allen* Stellen durch denselben Ausdruck zu ersetzen. Regeln dieser Art nennt man allgemeine Regeln im Gegensatz zu Regeln, wo der zu ersetzende Teilbaum *genau* mit *lhs* übereinzustimmen hat, welche man spezielle Regeln nennt. So sind die beiden ersten Regeln für `log` spezielle, die beiden anderen allgemeine. Letztere beiden Regeln geben die Transformationsvorschriften an, wenn die Funktionssymbole `log` und `exp` bzw. `df` (für die Ableitung) und `log` irgendwo im Ableitungsbaum aufeinanderfolgen.

Ein solches Regelsystem kann durchaus einen größeren Umfang erreichen:

```
showrules sin;

{sin(pi) => 0,
 sin(pi/2) => 1,
 sin(pi/3) => sqrt(3)/2,
 sin(pi/4) => sqrt(2)/2,
 sin(pi/6) => 1/2,
 sin((5*pi)/12) => sqrt(2)/4*(sqrt(3) + 1),
 sin(pi/12) => sqrt(2)/4*(sqrt(3) - 1),
 sin((~(~ x)*i)/~(~ y)) => i*sinh(x/y) when impart(y)=0,
 sin(atan(~u)) => u/sqrt(1 + u**2),
 sin(2*atan(~u)) => 2*u/(1 + u**2),
 sin(~n*atan(~u)) => sin((n - 2)*atan(u))*(1 - u**2)/(1 + u**2)
       + cos((n - 2)* atan(u))*2*u/(1 + u**2)
                when fixp(n) and n>2,
 sin(acos(~u)) => sqrt(1 - u**2),
 sin(2*acos(~u)) => 2*u*sqrt(1 - u**2),
 sin(2*asin(~u)) => 2*u*sqrt(1 - u**2),
 sin(~n*acos(~u)) => sin((n - 2)*acos(u))*(2*u**2 - 1)
       + cos((n - 2)*acos(u))*2* u*sqrt(1 - u**2)
                when fixp(n) and n>2,
 sin(~n*asin(~u)) => sin((n - 2)*asin(u))*(1 - 2*u**2)
       + cos((n - 2)*asin(u))*2* u*sqrt(1 - u**2)
                when fixp(n) and n>2,
 sin((~x + ~(~ k)*pi)/~d) => sign(k/d)*cos(x/d)
                when x freeof pi and abs(k/d)=1/2,
```

```
 sin((~(~ w) + ~(~ k)*pi)/~(~ d)) =>
        (if evenp(fix(k/d)) then 1 else  - 1)
        *sin(( w + remainder(k,d)*pi)/d)
                when w freeof pi and ratnump(k/d) and abs(k/d)>=1,
 sin((~(~ k)*pi)/~(~ d)) => sin((1 - k/d)*pi)
                when ratnump(k/d) and k/d>1/2,
 sin(asin(~x)) => x,
 df(sin(~x),~x) => cos(x)}$
```

Manche der angegebenen Regeln sind noch konditional untersetzt, d.h. nur dann anwendbar, wenn die in der Definition verwendeten formalen Parameter mit aktuellen Parametern belegt sind, die noch Zusatzvoraussetzungen erfüllen. Diese Effekte sind von regelorientierten Programmiersprachen wie etwa Prolog aber gut bekannt.

Diese Regeln werden automatisch angewendet, wenn Reduce das Funktionssymbol `sin` in einem Ausdruck antrifft. So werden etwa (Regel $9-11$) Ausdrücke der Form $sin(n * atan(x))$ aufgelöst:

```
sin(5*atan(x));
              4       2
          x*(x  - 10*x  + 5)
    ------------------------------
           2        4      2
     sqrt(x  + 1)*(x  + 2*x  + 1)
```

Dasselbe Ergebnis kann man mit Maple über die Funktion `expand` erreichen, die Mehrfachwinkel-Ausdrücke auflöst, sowie die in der Definition von `sin` eingebrannte Kenntnis, daß

$$\sin(\arctan(x)) = \frac{x}{\sqrt{1+x^2}}$$

gilt.

```
print(sin);
proc(x)
    local n, t;
    ...
        elif type(x, 'function') and nops(x) = 1 then
            n := op(0, x);
            t := op(1, x);
    ...
            elif n = 'arctan' then t/sqrt(1 + t^2)
    ...
    end

sin(5*arctan(x));
                            sin(5 arctan(x))

expand(");
                        x                x             x
                  16 ----------- - 12 ----------- + -----------
                          2 5/2            2 3/2         2 1/2
                     (1 + x )         (1 + x )      (1 + x )
```

```
normal(");
                                  2    4
                          x (5 - 10 x  + x )
                          ------------------
                                  2 5/2
                             (1 + x )
```

In beiden Konzepten sind aus Effizienzgründen oft Teile der Transformationen im Systemkern fest eingebrannt. Dies gilt insbesondere, auch aus anderen Gründen, für die Polynomarithmetik.

Steuerstrukturen im symbolischen Rechnen

Die größte Ähnlichkeit mit klassischen Programmiersprachen weist ein CAS im Bereich der Steuerstrukturen auf. Es werden in der einen oder anderen Form alle für eine imperative Programmiersprache üblichen Steuerstrukturen (Anweisungsfolgen, Verzweigungen, Schleifen) zur Verfügung gestellt, die sich selbst in der Syntax an gängigen Vorbildern wie PASCAL oder C orientieren. Auch Unterprogrammtechniken stehen zur Verfügung, wie wir im Abschnitt "Funktionen" bereits gesehen hatten.

Eine Besonderheit ist die zentrale Stellung von (geschachtelten) Listen, für die ein CAS deshalb ein ausreichendes Repertoire an Funktionalität zur Verfügung stellen muß. Dabei ist zu beachten, daß Maple und MuPAD die etwa in einem Funktionsaufruf auftretende, komma-separierte *Argumentliste* (expression sequence) als grundlegenden Datentyp verwenden, aus dem durch entsprechende Funktionen Listen, Mengen und andere Datenstrukturen erstellt werden können, die anderen CAS dagegen direkt mit auch dem Nutzer zugänglichen Listen als grundlegendem Datentyp operieren.

Benötigt wird erst einmal ein Zugriffsoperator auf einzelne Listenelemente, evtl. auch über mehrere Ebenen hinweg, der gewöhnlich `part` (als Teil einer Liste) oder `op` (für Operand in einer Argumentliste) heißt. Die meisten Systeme stellen auch eine iterierte Version zur verfügung, mit der man tiefergelegene Teilausdrücke in einer geschachtelten Liste selektieren kann.

Mit diesen Operatoren wäre eine Listentraversion nun mit der klassischen `for`-Anweisungen möglich, etwa als

```
> for i from 1 to nops(l) do ... something with op(l,i) ...
```

wobei `nops(l)` die Länge der Liste l zurückgibt und mit `op(l,i)` auf die einzelnen Listenelemente zugegriffen wird. Aus naheliegenden Effizienzgründen stellen die meisten CAS jedoch eine spezielle Listentraversion der Form

```
> for x in l do ... something with x ...
```

zur Verfügung.

Daneben spielt die Listenmanipulation eine wichtige Rolle, insbesondere deren Generierung, selektive Generierung und uniforme Manipulation. Zur Erzeugung von Listen gibt es in allen CAS einen *Sequenzierungsoperator*, der eine Liste aus einzelnen Elementen nach einer Bildungsvorschrift generiert. In Maple und MuPAD wird dafür ein eigener Operator verwendet, in Reduce dagegen die Syntax von `for` dahingehend erweitert, daß sie einen Wert zurückgibt.

```
Maple:
u:=[seq(i^2,i=1..5)];
                          u := [1, 4, 9, 16, 25]
```

```
# oder

u:=[i^2 $ i=1..5];
                              u := [1, 4, 9, 16, 25]

Reduce:

u:=for i:=1:5 collect i^2;

     u := {1,4,9,16,25}
```

Bei der selektiven Listengenerierung möchte man aus einer bereits vorhandenen Liste alle Elemente mit einer gewissen Eigenschaft auswählen. Maple, MuPAD und Mathematica haben dafür einen eigenen Select-Operator, der als Argumente eine boolesche Funktion und eine Liste nimmt und die gewünschte Teilliste zurückgibt. Reduce nutzt schließlich die erweiterte `for`-Syntax, mit der man auch iteriert erzeugte Listen zusammenhängen kann, sowie die Tatsache, daß eine `if`-Anweisung einen Wert zurückgeben kann, dazu Listen der Länge 1 oder 0 zu erzeugen und diese zusammenzufügen.

Betrachten wir etwa, wie man die Primzahlen bis 50 in einer Liste aufsammeln kann:

```
Maple (aehnlich MuPAD):

select(isprime,[$1..50]);

           [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

Mathematica:

Select[Range[1,50],PrimeQ]

           {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47}

Macsyma:

sublist(1..50,primep);

           [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

Reduce:

for i:=1:50 join if primep(i) then {i} else {};

     {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47}
```

Uniforme Listenmanipulationen treten immer dann auf, wenn man auf alle Elemente einer Liste ein und dieselbe Funktion anwenden möchte. Viele der Systeme gehen bei Funktionen, die auf Listen angewendet werden, jedoch nur für deren einzelne Elemente einen Sinn ergeben, davon aus, daß die Funktion auf die einzelnen Elemente anzuwenden ist. So liefert etwa Maple

```
u:=[sin(i) $ i=1..3];
                          u := [sin(1), sin(2), sin(3)]
```

```
evalf(u);
                    [.8414709848, .9092974268, .1411200081]
```

wobei offensichtlich die Funktionstransformation `evalf` $\circ$ `list` $\longrightarrow$ `list` $\circ$ `evalf` vorgenommen worden ist. Dort, wo dies nicht automatisch geschieht, kann die Funktion `map` eingesetzt werden, die als Argumente eine Funktion f und eine Liste l nimmt und die Funktion auf alle Listenelemente anwendet.

```
u:=[1,2,3];
                                u := [1, 2, 3]

sin(u);
                                sin([1, 2, 3])
#aber

map(sin,u);
                          [sin(1), sin(2), sin(3)]
```

In der folgenden Tabelle sind die wichtigsten Operationsmöglichkeiten auf Listen in den einzelnen CAS zusammengestellt.

System	Zugriff	Erzeugung	Selektion	Mapping
Axiom	`l.i`	`[f(i) for i in a..b]`	`[i for i from 1 to 50 | prime?(i)]`	`map(f,l)`
Macsyma	`part(l,i)`	`makelist(f(i),i,a,b)` oder `map(f,a..b)`	`sublist(1..50,primep)`	`map(f,l)`
Maple	`op(i,l)`	`[seq(f(i),i=a..b)]`	`select(isprime,[$1..50])`	`map(f,l)`
Mathematica	`Part[l,i]`	`Table[f(i),i,{a,b}]` oder `Map[f,Range[a,b]]`	`Select[Range[1,50],PrimeQ]`	`Map[f,l]`
MuPAD	`op(l,i)`	`[f(i)$ i=a..b]`	`select([$1..50],isprime)`	`map(l,f)`
Reduce	`part(l,i)`	`for i:=a:b collect f(i)`	`for i:=1:50 join if primep(i) then {i} else {}`	`map(f,l)`

Zugriff: Zugriff auf das Element i der Liste l
Erzeugung: Erzeugung einer Liste $\{f(i),\ i = a, \ldots, b\}$
Selektion: Liste der Primzahlen bis 50
Mapping: Wende die Funktion f auf alle Elemente der Liste l an

Tabelle 7: Operationen auf Listen

Wir hatten in den Beispielen bereits gesehen, daß in Reduce

Steuerstrukturen einen Wert zurückgeben.

Dieses Verhalten zeigen die meisten der Systeme (ausgenommen Maple, MuPAD und teilweise Mathematica). Es zeigt sich, daß die spezielle Art einer Steuerstruktur es sogar erlaubt, diese selbst als Funktionsaufruf zu implementieren, etwa wie in Mathematica:

```
> While[test,body]
```

führt die Anweisungsfolge `body` so lange aus, bis `test` den Wert *true* zurückgibt. Dies eröffnet weitere Möglichkeiten der Unifizierung von Funktions- und Steuerstrukturen.

Kapitel 4

Das Simplifizieren von Ausdrücken

Eine wichtige Eigenschaft von CAS ist die Möglichkeit, *zielgerichtet* Ausdrücke in eine semantisch gleichwertige, aber syntaktisch verschiedene Form zu transformieren. Derartige Transformationen wollen wir im folgenden als Simplifikation bezeichnen.

Neben einigen allgemeingültigen Vereinfachungen wie etwa

> das Zusammenfassen gleichartiger Summanden in einem polynomialen Ausdruck
>
> oder Vereinfachungen der Art

$$\begin{array}{rcl} \sin(\arcsin(x)) & \rightarrow & x \\ \sin(\arctan(x)) & \rightarrow & \frac{x}{\sqrt{x^2+1}} \\ abs(abs(x)) & \rightarrow & abs(x) \end{array}$$

können dabei an verschiedenen Stellen einer Rechnung Transformationen mit unterschiedlichen Intentionen und sogar einander widersprechenden Zielvorgaben erforderlich sein. Zur Berechnung des Integrals

$$\int \log\left(\frac{x+1}{x-1}\right)\, d\,x = (x+1)\log(x+1) - (x-1)\log(x-1)$$

und zur Lösung der Gleichung

$$\log(x+1) - \log(x-1) = 1 \quad \Leftrightarrow x = \frac{e+1}{e-1}$$

sind etwa die Logarithmengesetze in jeweils unterschiedlicher Richtung anzuwenden. Ähnlich kann man polynomiale Ausdrücke expandieren oder aber in faktorisierter Form darstellen, Basen in Potenzfunktionen zusammenfassen oder aber trennen, Additionstheoreme anwenden, um trigonometrische Ausdrücke eher als Summen oder eher als Produkte darzustellen, die Gleichung $\sin(x)^2 + cos(x)^2 = 1$ verwenden, um eher `sin` durch `cos` oder eher `cos` durch `sin` zu ersetzen usw.

Das Simplifikationssystem eines CAS muß deshalb genügend flexibel ausgelegt sein, um auch einen derartigen Wechsel zwischen verschiedenen Simplifikationsstrategien zu ermöglichen.

4.1 Simplifikationssysteme

Um einen solchen Anspruch zu realisieren, werden im Design der CAS zwei verschiedene Strategien verfolgt. Unter der ersten Strategie, Simplifikation nach Aufforderung, nimmt das CAS eine Reihe allgemeingültiger Simplifikationen, die allen Simplifikationsstrategien gemeinsam sind, automatisch vor. Weitergehende Simplifikationen bedürfen expliziter Aktivität des Nutzers, dem dafür

verschiedene Funktionen wie `expand`, `collect`, `factor` oder `normal` zur Verfügung stehen, die verschiedene, häufig erforderliche, aber fest vorgegebene Simplifikationsstrategien (Ausmultiplizieren, Zusammenfassen von Termen nach gewissen Prinzipien, Anwendung von Additionstheoremen für Winkelfunktionen, Anwendung von Potenz- und Logarithmengesetzen usw.) realisieren. Daneben existiert meist eine (oder mehrere) komplexere Funktion `simplify`, die das Ergebnis verschiedener Transformationsstrategien miteinander vergleicht und an Hand des Ergebnisses entscheidet, welches denn nun das "einfachste" ist.

Diese Strategie wird von den Systemen Derive, Maple, Mathematica und MuPAD verwendet. Ihr Vorteil ist die leichte Änderbarkeit der Simplifikationsrichtung im Laufe des interaktiven Dialogs. Der Nachteil dieses Herangehens besteht in der relativen Starrheit des Simplifikationssystems, womit ein Abweichen von den fest vorgegebenen Simplifikationsstrategien nur unter erheblichem Aufwand möglich ist.

Reduce sowie in gewissem Sinne auch Axiom verfolgen die zweite Strategie, die der

vollständigen automatischen Simplifikation.

Die Zielrichtung der Simplifikation wird dabei durch einen *gültigen Kontext* bestimmt. Ein eingegebener Ausdruck wird automatisch nach allen im entsprechenden Kontext gerade aktiven Simplifikationsregeln umgeformt. Simplifikation erfolgt also über einen *allgemeinen Simplifikationsmechanismus*, der mit unterschiedlichen Regelsystemen als den jeweiligen "Daten" zu unterschiedlichen Ergebnissen führt. Die Änderung der Simplifikationsrichtung erfolgt durch Veränderung des gültigen Kontexts, der durch den Wert globaler Variablen, sogenannter *Schalter*, bestimmt ist, die man frei untereinander kombinieren kann und die das Zu- bzw. Abschalten bestimmter Simplifikationseffekte bewirken. Auf diese Weise sind Simplifikationsstrategien wesentlich flexibler formulierbar. Allerdings besteht der Nachteil dieses Zugangs darin, daß man in der Vielzahl der verschiedenen Schalter schnell den Überblick über den jeweils aktuell gültigen Kontext verlieren kann.

Ein größeres Problem bei diesem Zugang ist der Umgang mit einander widersprechenden Simplifikationsstrategien, da man diese nicht gleichzeitig wirken lassen darf, wenn man Endlosschleifen vermeiden möchte. Was im ersten Zugang durch das Design der verschiedenen systemspezifischen Simplifikationsfunktionen gewährleistet wurde, kann hier bei einer *beliebigen* Kombinierbarkeit der verschiedenen Schalter *prinzipiell* nicht gewährleistet werden. So kann man die Beziehung $log(x * y) = log(x) + log(y)$ als Transformationsregel

$$log(x * y) \mapsto log(x) + log(y)$$

aber auch als

$$log(x) + log(y) \mapsto log(x * y)$$

anwenden. Verwendet man einen Schalter, um zwischen beiden Strategien *umzuschalten*, so wird das System einen Ausdruck der Form $\log(u \cdot v) + \log(x) + \log(y)$ immer umformen. Da auch solche Ausdrücke manchmal eine Daseinsberechtigung haben, stellt Reduce zwei Schalter `combinelogs` und `expandlogs` zur Verfügung, mit denen man das entsprechende Simplifikationsverhalten ein- oder ausschalten kann. Würde man beide gleichzeitig aktivieren, so würde eine unendliche Simplifikationsschleife eintreten. Dies kann man vermeiden, indem das Einschalten des einen Schalters den anderen automatisch ausschaltet[1].

Wir sehen an diesem Beispiel zugleich, daß es in einem solchen Ansatz schwierig ist, gezielte *teilweise* Simplifikationen vorzunehmen.

Macsyma verfolgt eine Mischung aus beiden Strategien, indem sowohl Schalter zur Beeinflußung des Simplifikationsverhaltens als auch verschiedene Simplifikationsfunktionen zur Verfügung stehen.

[1] In der gegenwärtigen Version 3.6 von Reduce können allerdings beide Schalter nebeneinander existieren. Sind beide aktiviert, so wird zwischen beiden Darstellungen bei jedem Simplifikationszyklus gewechselt.

Die Vorteile konsistenter Simplifikationssysteme und deren flexible Einsetzbarkeit werden, ähnlich den lokalen Wertzuweisungen an Variablen, durch *lokale Simplifikationssysteme* miteinander kombiniert. Über einfache Mechanismen zur Erstellung und Anwendung lokaler Simplifikationssysteme verfügen derzeit nur die Systeme Axiom, Mathematica und Reduce. Reduce nutzt dabei auf elegante Weise denselben zentralen Simplifikationsmechanismus wie auch für die interne Simplifikation, indem dieser mit *lokalen Regelsystemen* gefüttert wird, die dieselbe Struktur haben wie die dem System bekannten Regeln für Funktionssymbole, wie wir sie mit `showrules` extrahiert hatten.

4.2 Simplifikation und mathematische Exaktheit

Wir hatten bereits gesehen, daß es in beiden Zugängen zur Simplifikationsproblematik einen

Kern von allgemeingültigen Simplifikationen

gibt, die allen Simplifikationsstrategien gemeinsam sind und deshalb stets automatisch ausgeführt werden[2].

Dazu gehört zunächst einmal die Strategie, spezielle Werte von Funktionsausdrücken, sofern diese durch "einfachere" Symbole exakt ausgedrückt werden können, durch diese zu ersetzen.

Beispiele (Maple):

```
sqrt(36);
                                    6

sin(Pi/4);
                                     1/2
                                  1/2 2

tan(Pi/6);
                                     1/2
                                  1/3 3

arcsin(1);
                                   1/2 Pi
```

Dies trifft auch für kompliziertere Funktionsausdrücke zu, die auf "elementarere" Funktionen zurückgeführt werden, in denen mehr oder weniger gut studierte spezielle mathematische Funktionen auftreten.

```
GAMMA(1/2);
                                    1/2
                                  Pi

int(exp(-x^2),x=0..infinity);
                                      1/2
                                  1/2 Pi

int(exp(-x^2),x=0..y);
                                          1/2
                              1/2 erf(y) Pi
```

[2] Derive nimmt initial allerdings überhaupt keine Simplifikationen vor.

```
sum(1/i^2,i=1..infinity);
                                    1/2 Pi


sum(1/i^n,i=1..infinity);
                                    Zeta(n)
```

In den Beispielen wurden die Gamma-Funktion $\Gamma(x)$, die Gaußsche Fehlerfunktion $erf(x)$ sowie die Riemannsche Zeta-Funktion $\zeta(n)$ verwendet.

Weiterhin wird auch eine Reihe komplizierterer Umformungen von einigen der Systeme automatisch ausgeführt wie z.B. (Maple):

```
sqrt(24);
                                       1/2
                                    2 6


sqrt(2*sqrt(3)+4);
                                     1/2
                                    3    + 1


sqrt(11+6*sqrt(2))+sqrt(11-6*sqrt(2));

                                       6
```

Auch werden eindeutige Simplifikationen von Funktionsausdrücken ausgeführt wie etwa

```
abs(abs(x));
                                     | x |


tan(arctan(x));
                                       x


tan(arcsin(x));
                                       x
                                  -----------
                                        2 1/2
                                  (1 - x )


abs(-Pi*x);
                                   Pi | x |


cos(-x);
                                    cos(x)


exp(3*ln(x));
                                       3
                                      x
```

Auf den ersten Blick mag es deshalb verwunderlich erscheinen, daß folgende Ausdrücke nicht vereinfacht werden:

```
sqrt(x^2);
                                    2 1/2
                                   (x )
ln(exp(x));
                                  ln(exp(x))

arctan(tan(x));
                                 arctan(tan(x))
```

In jedem der drei Fälle würde der durchschnittliche Nutzer als Ergebnis wohl x erwarten. Für die letzte Beziehung ist das allerdings vollkommen falsch, denn *arctan* ist nur im Intervall $[-\pi/2, \pi/2]$ die Umkehrfunktion von *tan*. Die korrekte Antwort lautet für $x \in \mathbf{R}$ also

$$\arctan(\tan(x)) = x - \left[\frac{x}{\pi} + \frac{1}{2}\right] \cdot \pi,$$

wobei $[a]$ für den ganzen Teil der Zahl $a \in \mathbf{R}$ steht.

Daß auch $\sqrt{x^2} = x$ mathematisch nicht exakt ist, dürfte bei einigem Nachdenken ebenfalls einsichtig sein und als Ergebnis der Simplifikation $|x|$ erwartet werden. Diese Antwort wird auch von Reduce gegeben. Maple allerdings gibt nach expliziter Aufforderung

```
simplify(sqrt(x^2));
                                   csgn(x) x
```

zurück, obwohl wir in den Beispielen gesehen hatten, daß es auch die Betragsfunktion kennt. Der Grund liegt darin, daß das naheliegende Ergebnis $|x|$ nur für *reelle* Argumente korrekt ist, nicht dagegen für komplexe. Für komplexe Argumente ist die Wurzelfunktion mehrwertig, so daß $\sqrt{x^2} = \pm x$ eine korrekte Antwort wäre. Da man in diesem Fall oft vereinbart, daß der Wert der Wurzel der Hauptwert ist, also derjenige, der in der oberen komplexen Halbebene liegt, wird hier die *komplexe Vorzeichenfunktion* `csgn` verwendet. In diesem Kontext ist auch die Vereinfachung des Ergebnisses zu $|x|$, dem Betrag der komplexen Zahl x, fehlerhaft.

Für noch allgemeinere mathematische Strukturen, in denen Multiplikationen und deren Umkehrung definiert werden können, wie etwa Gruppen (quadratische Matrizen oder ähnliches), ist allerdings selbst diese Simplifikation nicht korrekt. In einem vollkommen rigorosen Verständnis dürfte also $\sqrt{x^2}$ ohne weitere Annahmen über den Wertebereich, aus dem das Argument genommen wird, überhaupt nicht simplifiziert werden.

Die dritte Beziehung $ln(exp(x)) = x$ ist wegen der Monotonie der beteiligten Funktionen dagegen für alle reellen Werte von x richtig. Für komplexe Argumente kommt dagegen, ähnlich wie für die Wurzelfunktion, die Mehrdeutigkeit der Logarithmusfunktion ins Spiel.

Weitere interessante Simplifikationsfälle sind die Ausdrücke

```
sqrt(1/x)-1/sqrt(x);

sqrt(x*y)-sqrt(x)*sqrt(y);
```

die für solche Argumente, für die sie "sinnvoll" definiert sind (also hier etwa für positive reelle x), zu Null vereinfacht werden können. Für negative reelle Argumente haben wir in diesem Fall allerdings nicht nur die Mehrdeutigkeit der Wurzelfunktion im Bereich der komplexen Zahlen zu berücksichtigen, sondern kollidieren mit anderen, wesentlich zentraleren Annahmen, wie etwa der automatischen Ersetzung von $\sqrt{-1}$ durch die imaginäre Einheit I. Setzen wir in obigen Ausdrücken $x = y = -1$ und führen diese Ersetzung aus, so erhalten wir im ersten Fall $I - 1/I = 2\,I$ und im zweiten Fall $\sqrt{1} - I^2 = 2$. Solche Inkonsistenzen tief in komplexen Berechnungen versteckt können zu vollkommen falschen Resultaten führen, ohne daß der Grund dafür offensichtlich wird.

Natürlich sind Simplifikationssysteme mit zu rigiden Annahmen für die meisten Anwendungszwecke untauglich, wenn sie derart simple Umformungen "aus haarspalterischen Gründen" nicht oder nur nach gutem Zureden ausführen. Jedes der CAS muß deshalb für sich entscheiden, auf welchen Grundannahmen seine Simplifikationen sinnvollerweise basieren, um ein ausgewogenes Verhältnis zwischen mathematischer Exaktheit einerseits und Praktikabilität andererseits herzustellen.

In der folgenden Tabelle sind die Simplifikationsergebnisse der verschiedenen CAS (in der Grundeinstellung) auf einer Reihe von Beispielen zusammengestellt (u bedeutet unsimplifiziert):

Ausdruck	Axiom	Derive	Macsyma	Maple	MMa	MuPAD	Reduce
$\lvert -\pi \cdot x \rvert$	$\lvert\pi \cdot x\rvert$	$\pi \cdot \lvert x\rvert$	$\pi \cdot \lvert x\rvert$	$\pi \cdot \lvert x\rvert$	$\pi \cdot \lvert x\rvert$	$\pi \cdot \lvert x\rvert$	$\pi \cdot \lvert x\rvert$
$\arctan(\tan(x))$	x	u	u	u	u	u	u
$\arctan(\tan(\frac{25}{4}\pi))$	$\pi/4$	$\pi/4$	$\pi/4$	$\pi/4$	$\pi/4$	$\pi/4$	$\pi/4$
$\sqrt{x^2}$	u	$\lvert x\rvert$	$\lvert x\rvert$	u	u	u	$\lvert x\rvert$
$\sqrt{x\,y} - \sqrt{x}\,\sqrt{y}$	u	u	u	u	u	u	u
$\sqrt{\frac{1}{z} - \frac{1}{\sqrt{z}}}$	0	u	u	u	u	u	u
$\ln(\exp(x))$	x	u	x	u	u	u	x
$\ln(\exp(10 * I))$	u	-2i(2π-5)	i(10-4π)	u	10i-4πi	-4πi+10i	10i

Tabelle 1: Simplifikationsverhalten der verschiedenen Systeme an ausgewählten Beispielen

Macsyma, Maple und nun auch teilweise MuPAD besitzen mit dem `assume`-Mechanismus die Möglichkeit, einzelnen Symbolen *Eigenschaften* zuzuordnen, die beim Simplifizieren in die Entscheidungsfindung mit einbezogen werden. Solche Eigenschaften können Annahmen über die Natur einer Variablen (`assume(x,integer), assume(x,real)`), die Zugehörigkeit zu einem reellen Intervall (`assume(x>0)`), die spezielle Natur einer Matrix (`assume(m,quadratic)`) oder andere Eigenschaften sein. Symbole, für die Eigenschaften definiert sind, werden in Maple mit einer Tilde versehen. Einzelne Funktionen sind in der Lage, aus Eigenschaften ihrer Argumente Eigenschaften der Funktionswerte abzuleiten und weiterzugeben. Ähnlich können in Derive Annahmen (positiv, nichtnegativ, reell, komplex, aus einem reellen Intervall) über die Natur einzelner Variablen getroffen werden.

```
> assume(y,real);
> about(y);
Originally y, renamed y~:
  is assumed to be: real

> assume(x>0);
> about(x);
Originally x, renamed x~:
  is assumed to be: RealRange(Open(0),infinity)

> abs(-Pi*x);
                                    Pi x~

> sqrt(x^2);
                                     x~

> sqrt(x*y)-sqrt(x)*sqrt(y);
                                 1/2     1/2   1/2
                          (x~ y~)    - x~    y~

> simplify(");
                                     0
> sqrt(1/x)-1/sqrt(x);
```

```
                                    0

> ln(exp(y));
                                    y~
```

In Macsyma und Reduce läßt sich außerdem die Strenge der mathematischen Umformungen durch verschiedene Schalter verändern. So kann man etwa in Reduce über den (allerdings nicht dokumentierten) Schalter `reduced` die klassischen Vereinfachungen von Wurzelsymbolen, die für komplexe Argumente zu fehlerhaften Ergebnissen führen können, zulassen. In Macsyma können die entsprechenden Schaltern wie `logexpand, radexpand` oder `triginverses` sogar drei Werte annehmen: `false` (keine Simplifikation), `true` (bedachtsame Simplifikation) oder `all` (ständige Simplifikation).

Die mathematische Strenge der Rechnungen, die mit einem CAS allgemeiner Ausrichtung möglich sind, ist in den letzten Jahren stärker ins Blickfeld der Entwickler geraten. Die Konzepte der verschiedenen Systeme sind unter diesem Blickwinkel mehrfach geändert worden, was man an den differierenden Ergebnissen verschiedener Vergleiche, die zu unterschiedlichen Zeiten angefertigt wurden ([16], [15], [18]), erkennen kann. Allerdings werden selbst innerhalb eines Systems an unterschiedlichen Stellen manchmal unterschiedliche Maßstäbe der mathematischen Strenge angelegt, insbesondere bei der Implementierung komplexerer Verfahren. Ein abschließendes Urteil ist deshalb nicht möglich.

Für eine umfassendere Betrachtung dieser Problematik, die auch tiefergehende mathematische Fähigkeiten der verschiedenen Systeme in einen Vergleich einbezieht, sei dazu auf [15] verwiesen.

4.3 Das allgemeine Simplifikationsproblem

Zur genaueren Definition des Simplifikationsbegriffs sei $\mathcal{E}$ eine Menge linguistischer Objekte, etwa regulär zusammengesetzter Ausdrücke, und $\sim$ eine Äquivalenzrelation auf $\mathcal{E}$, die die semantische Äquivalenz repräsentiert. Als *Simplifikator* wollen wir eine effektiv berechenbare Funktion $S : \mathcal{E} \longrightarrow \mathcal{E}$ bezeichnen, die die beiden Bedingungen

$$(I) \quad S(S(t)) = S(t) \quad \text{(Idempotenz)} \qquad \text{und} \qquad (E) \quad S(t) \sim t \quad \text{(Äquivalenz)}$$

für alle $t \in \mathcal{E}$ erfüllt.

Die Tatsache, daß $S(t)$ *einfacher* als t ist, läßt sich am deutlichsten durch eine (teilweise) Ordnung $\leq$ auf $\mathcal{E}$ beschreiben, bezüglich welcher die Abbildung S die zusätzliche Bedingung

$$(S) \quad S(t) \leq t \qquad \text{für alle } t \in \mathcal{E}$$

erfüllen muß.

In einem CAS besteht $\mathcal{E}$ aus einer Menge von zulässigen Zeichenketten, den *Ausdrücken*, die intern als *Ableitungsbäume* (d.h. knotenmarkierte Wurzelbäume) dargestellt sind. Sei $\mathcal{A}$ die Menge dieser Ableitungsbäume und $p : \mathcal{E} \longrightarrow \mathcal{A}$ die (umkehrbar eindeutige) Parse-Funktion des Interpreters. Die Menge der Ableitungsbäume (in einem typfreien CAS) besitzt die folgenden drei Eigenschaften:

(1) Alle Variablen sind Ableitungsbäume (der Tiefe 0).

(2) Jeder Teilbaum eines Ableitungsbaums ist selbst wieder ein Ableitungsbaum.

(3) Ersetzt man in einem Ableitungsbaum einen Teilbaum durch einen anderen Ableitungsbaum (insbesondere durch eine Variable), so erhält man wieder einen Ableitungsbaum.

In diesem Kontext läßt sich ein Simplifikator durch ein (endliches) Regelsystem $\mathcal{R}$ beschreiben, dessen Regeln die Gestalt

$$L_i(u) \longrightarrow R_i(u)$$

haben mit $L_i, R_i \in \mathcal{A}$, wobei u eine Liste von formalen Parametern ist, die als Blätter in L_i bzw. R_i auftreten, und für alle Variablenbelegungen $u \vdash U$

$$p^{-1}\, L_i(U) \sim p^{-1}\, R_i(U)$$

in $\mathcal{E}$ gilt. Betrachtet man die Anwendung der einzelnen Regeln als elementaren Simplifikationsschritt, so ist $S : \mathcal{E} \longrightarrow \mathcal{E}$ der mit p bzw. p^{-1} gekoppelte transitive Abschluß der Ersetzungsrelationen aus $\mathcal{R}$.

Die Termination eines solchen Simplifikationsprozesses ist gewährleistet, wenn für die teilweise Ordnung $\leq$ auf $\mathcal{E}$ zusätzlich gilt:

(1) $\leq$ ist wohlfundiert.

(2) Für jede Regel $(L(u) \to R(u)) \in \mathcal{R}$ und jede Belegung $u \vdash U$ gilt $p^{-1}\, L(U) > p^{-1}\, R(U)$.

Die zweite Eigenschaft sichert, daß in einem elementaren Simplifikationsschritt die Vereinfachung zu einem "kleineren" Ausdruck führt, die erste, daß eine solche Simplifikationskette nur endlich viele Schritte haben kann. Jede Ordnung, die sich als

$$e_1 > e_2 \;:\Leftrightarrow\; l(e_1) > l(e_2) \quad \text{für } e_1, e_2 \in \mathcal{E}$$

mit einem natürlichzahligen Kompliziertheitsmaß $l : \mathcal{E} \longrightarrow \mathbf{N}$ (wie z.B. die Anzahl der verwendeten Zeichen, die Zahl der Klammern, der Additionen usw.) darstellen läßt, besitzt diese Eigenschaft.

Oftmals ist jedoch die Angabe einer solchen Ordnung auf $\mathcal{E}$ nicht so einfach zu bewerkstelligen. Wie will man etwa im Zusammenhang mit der Betrachtung der `expand`-Funktion den Grad der Expandiertheit eines Ausdruckes messen ? Deswegen werden im Zusammenhang mit Simplifikationen zwei weitere Begriffe verwendet, die von einem Ordnungsbegriff auf der Menge der Ausdrücke unabhängig sind und sich an anderen wünschenswerten Eigenschaften eines Simplifikators orientieren.

Als *kanonische Form* bezeichnet man einen Simplifikator, der zusätzlich der Bedingung

$$(C) \quad s \sim t \Rightarrow S(s) = S(t) \qquad \text{für alle } s, t \in E$$

genügt. Eine Klasse semantisch äquivalenter Ausdrücke kann also an Hand des Aussehens der entsprechenden kanonischen Form eindeutig identifiziert werden. Ein solcher kanonischer Simplifikator kann deshalb insbesondere dazu verwendet werden, die semantische Äquivalenz von zwei gegebenen Ausdrücken zu prüfen, d.h. das *Identifikationsproblem* zu lösen: Zwei Ausdrücke sind genau dann äquivalent, wenn ihre kanonischen Formen (Zeichen für Zeichen) übereinstimmen.

Solche kanonischen Simplifikatoren existieren nur für spezielle Klassen von Ausdrücken $\mathcal{E}$. Deshalb ist auch die folgende Abschwächung dieses Begriffs von Interesse. Nehmen wir an, daß $\mathcal{E}/\sim$, wie in den meisten Anwendungen in der Computeralgebra, die Struktur einer additiven Gruppe trägt, d.h. ein spezielles Symbol $0 \in \mathcal{E}$ für das Nullelement sowie eine Funktion $M : \mathcal{E} \times \mathcal{E} \longrightarrow \mathcal{E}$ besitzt, die die Differenz zweier Ausdrücke aufzuschreiben vermag. Letzeres bedeutet insbesondere, daß $s \sim t \Leftrightarrow M(s,t) \sim 0$ gilt. Als *Normalform* bezeichnen wir dann einen Simplifikator S, für den zusätzlich

$$(N) \quad t \sim 0 \Rightarrow S(t) = 0 \qquad \text{für alle } t \in \mathcal{E}$$

gilt, d.h. jeder Nullausdruck $t \in \mathcal{E}$ wird durch S auch als solcher erkannt.

Diese Forderung ist schwächer als die der Existenz einer kanonischen Form: Es kann sein, daß für zwei Ausdrücke $s, t \in \mathcal{E}$, die keine Nullausdrücke sind, zwar $s \sim t$ gilt, diese jedoch zu verschiedenen Normalformen simplifizieren. Gleichwohl können wir mit einem solchen Normalform-Operator S ebenfalls das Identifikationsproblem lösen, denn es gilt offensichtlich

$$s \sim t \Leftrightarrow S(M(s,t)) = 0.$$

4.4 Simplifikation polynomialer und rationaler Ausdrücke

Wir hatten bei der Betrachtung der Fähigkeiten eines CAS bereits gesehen, daß sie besonders gut in der Lage sind, polynomiale und rationale Ausdrücke zu vereinfachen. Der Grund dafür ist die Tatsache, daß in der Menge dieser Ausdrücke kanonische bzw. normale Formen existieren.

Betrachten wir dazu den Polynomring $S := R[x_1, \ldots, x_n]$ in den Variablen $X = (x_1, \ldots, x_n)$ über dem Grundring R. Wir hatten gesehen, daß solche Polynome $f \in S$ in expandierter Form als Summe von Monomen $f = \sum_a c_a X^a$ dargestellt werden, wobei $a = (a_1, \ldots, a_n) \in \mathbf{N}^n$ ein Multiindex ist und X^a kurz für $x_1^{a_1} \cdots x_n^{a_n}$ steht. In Maple, MuPAD und Mathematica kann man eine solche Darstellung durch die Funktionen `expand` oder `normal` erzeugen, in Reduce und Macsyma wird sie standardmäßig für die Darstellung von Polynomen verwendet.

Diese Darstellung ist eindeutig, d.h. eine kanonische Form für Polynome $f \in S$, wenn für die Koeffizienten, also die Elemente aus R eine solche kanonische Form existiert und die Reihenfolge der Summanden festgelegt ist. Zur Festlegung der Reihenfolge definiert man gewöhnlich eine Ordnung auf $T(X) = \{X^a : a \in \mathbf{N}^n\}$, dem *Monoid der Terme* in $(x_1, \ldots, x_n)$.

Als *distributive Darstellung* eines Polynoms $f \in S$ bzgl. einer solchen Ordnung bezeichnet man eine Darstellung $f = \sum_a c_a X^a$, in der die Summanden paarweise verschiedene Terme enthalten, diese in fallender Reihenfolge angeordnet sind und die einzelnen Koeffizienten in ihre kanonische Form gebracht wurden. In dieser Darstellung ist die Addition von Polynomen besonders effizient ausführbar. Ist die gewählte Ordnung darüberhinaus *monoton*, d.h. gilt

$$s < t \Rightarrow s \cdot u < t \cdot u \quad \text{für alle } s, t, u \in T(X),$$

so kann man auch die Multiplikation recht effektiv ausführen, da dann beim gliedweisen Multiplizieren einer geordneten Summe mit einem Monom die Summanden geordnet bleiben. Wohlfundierte Ordnungen mit dieser Zusatzeigenschaft bezeichnet man als *Termordnungen*.

Zusammenfassend können wir folgenden Satz formulieren:

Satz 3 *Existiert auf R eine kanonische Form, dann ist die distributive Darstellung von Polynomen $f \in S$ mit Koeffizienten in kanonischer Form eine kanonische Form auf S.*

Der Beweis ist offensichtlich. Damit kann in Polynomringen über gängigen Grundbereichen wie den ganzen oder rationalen Zahlen, für die kanonische Formen existieren, effektiv gerechnet werden.

Verwendet man die rekursive Natur des bewiesenen Satzes, so erhält man unmittelbar, daß auch eine zweite Darstellung, die *rekursive Darstellung*, eine kanonische Form auf S ist. Diese beruht darauf, Polynome in S als Polynome in x_n mit Koeffizienten aus $R[x_1, \ldots, x_{n-1}]$ zu betrachten.

Eine rekursive Darstellung des in distributiver kanonischer Form gegebenen Polynoms

$$f := 2\,x^3\,y^2 + 3\,x^2\,y^2 + 5\,x^2\,y - x\,y^2 - 3\,x\,y + x - y - 2$$

als Element von $R[y][x]$ ist dann

$$(2y^2)\,x^3 + (3y^2 + 5y)\,x^2 + (-y^2 - 3y + 1)\,x + (-y - 2)$$

Betrachten wir das Polynom

$$u := \left(x - xy + x^2 + (x - y + 1)\left(x^2 + 3\right)\right)\left(-9y - 3y^2 + y^3 + x^4\left(2y + y^2 + 1\right) - 5\right)$$

```
> normal(u);
```

$$\begin{aligned} &-20x - 12y - 31xy - 10x^2 - 5x^3 + 18y^2 + 3x^4 + 12y^3 + 4x^5 - 3y^4 + 2x^6 + x^7 - 3xy^2 - 13x^2y \\ &\quad +7xy^3 - 9x^3y - xy^4 + 3x^4y + 7x^5y + 3x^6y + 2x^7y + 3x^2y^2 + 5x^2y^3 \\ &\quad -3x^3y^2 - x^2y^4 + x^3y^3 - 3x^4y^2 - 3x^4y^3 + 2x^5y^2 - x^5y^3 - x^6y^3 + x^7y^2 - 15 \end{aligned}$$

```
> factor(u);
```

$$\left(x+x^2+3\right)\left(x-y+1\right)\left(y+1\right)^2\left(y+x^4-5\right)$$

so erkennen wir, daß auch in nichttrivialen Beispielen eine auf einer Faktorzerlegung beruhende Simplifikation zu kompakten Ergebnissen führen kann.

Allerdings ist eine Darstellung von Polynomen in faktorisierter Form mit wesentlich mehr Rechenaufwand verbunden als eine Darstellung in distributiver oder rekursiver Form, da bei *jeder* Addition zweier Polynome eine (wenigstens teilweise) Faktorisierung notwendig wird, was eine sehr zeitaufwendige Prozedur sein kann. Hinzu kommt, daß sich Primfaktoren nur eindeutig bis auf eine Einheit des jeweiligen Integritätsbereichs bestimmen lassen, d.h. für eine kanonische Form selbst bei Gültigkeit des Satzes von der eindeutigen Primfaktorzerlegung zusätzliche Normierungen notwendig werden.

Die *faktorisierte Darstellung* von Polynomen erfüllt deshalb in den meisten CAS nur die Anforderungen an eine Normalform.

Rationale Funktionen:

In unserer Sammlung von Beispielen hatten wir gesehen, daß CAS auch hervorragend in der Lage sind, mit rationalen Funktionen umzugehen. Allerdings ist es in einigen Beispielen besser, die spezielle Simplifikationsstrategie `normal` zu verwenden als den allgemeinen Ansatz `simplify` (MuPAD):

```
u:= a^3/((a-b)*(a-c)) + b^3/((b-c)*(b-a)) + c^3/((c-a)*(c-b));
```

$$\frac{a^3}{\left(a-b\right)\left(a-c\right)}+\frac{b^3}{\left(b-a\right)\left(b-c\right)}+\frac{c^3}{\left(c-a\right)\left(c-b\right)}$$

```
simplify(u);
```

$$\frac{a^3}{-ab-ac+bc+a^2}-\frac{b^3}{ab-ac+bc-b^2}+\frac{c^3}{ab-ac-bc+c^2}$$

```
normal(u);
```

$$a+b+c$$

```
u:= (x-y)/sum(x^i*y^(5-i),i=0..5)-(x^2-x*y+y^2)/(x^6-y^6);
```

$$-\frac{-xy+x^2+y^2}{x^6-y^6}+\frac{x-y}{x^5+y^5+xy^4+x^4y+x^2y^3+x^3y^2}$$

```
simplify(u);
```

$$\frac{1}{-x^4+y^4+xy^3-x^3y}+\frac{x}{x^5+y^5+xy^4+x^4y+x^2y^3+x^3y^2}-\frac{y}{x^5+y^5+xy^4+x^4y+x^2y^3+x^3y^2}$$

```
normal(u);
```

$$-\frac{xy}{x^6-y^6}$$

Die Simplifikationsstrategie von `normal` ist offensichtlich: Finde einen gemeinsamen Hauptnenner und überführe das entstehende Zählerpolynom in die distributive Form. Das ist zwar keine kanonische Form wie im Fall der Polynome, da ja Zähler und Nenner nicht unbedingt teilerfremd sein müssen, allerdings eine *Normalform*, denn auf diese Weise werden Null-Ausdrücke sicher erkannt. Diese Darstellung nennt man deshalb auch die *rationale Normalform*.

Von dieser Normalform gelangt man zu einer *kanonischen Form*, indem man den gcd von Zähler und Nenner ausdividiert und dann die beiden Teile des Bruches noch geeignet normiert, vgl. etwa [3]. Da die Berechnung des gcd aber aufwendig im Vergleich zu den anderen arithmetischen Operationen sein kann, verzichtet z.B. Reduce auf diese Berechnung, wenn nicht der Schalter `gcd` gesetzt ist:

```
(x^5-1)/(x^3-1);
```

$$\frac{x^5-1}{x^3-1}$$

```
on gcd; ws;
```

$$\frac{x^4+x^3+x^2+x+1}{x^2+x+1}$$

Ähnlich verfährt die Mathematica-Funktion `Together`, während die Normalformberechnung `normal` in Maple und MuPAD das Austeilen des gcd einschließt. Allerdings wird auch hier auf eine Normierung verzichtet, so daß das Ergebnis eine Normalform, aber keine kanonische Form ist.

Auf Grund der guten Eigenschaften, die die Simplifikation polynomialer und rationaler Ausdrücke besitzt, wird sie auch auf Ausdrücke angewendet, die nur teilweise eine solche Struktur besitzen.

```
Reduce:
(sin(x)+cos(x))^5;
```

$$\begin{aligned}\cos(x)^5 + 5\cos(x)^4\sin(x) + 10\cos(x)^3\sin(x)^2 + 10\cos(x)^2\sin(x)^3\\ +5\cos(x)\sin(x)^4 + \sin(x)^5\end{aligned}$$

```
MuPAD:
expand((sin(x)+cos(x))^5);
```

$$\begin{aligned}\cos(x)^5 + \sin(x)^5 + 5\cos(x)\sin(x)^4 + 5\cos(x)^4\sin(x)\\ +10\cos(x)^2\sin(x)^3 + 10\cos(x)^3\sin(x)^2\end{aligned}$$

Dazu werden einzelne Funktionsausdrücke wie in diesem Fall `sin(x)` und `cos(x)` als algebraisch unabhängig angesehen und die Normalformexpansion mit diesen Kernen als verallgemeinerten Variablen betrieben. Die zwischen ihnen bestehende algebraische Relation $\sin(x)^2 + \cos(x)^2 = 1$ kann durch andere Simplifikationsmechanismen zur Wirkung gebracht werden.

Polynomiale und rationale Ausdrücke in solchen Kernen spielen damit eine wichtige Rolle im Simplifikationsdesign aller CAS. Sowohl die Herstellung solcher Normalformen und anschließende Anwendung weitergehender Vereinfachungen wie in obigen Beispiel als auch die gezielte Umformung anderer funktionaler Abhängigkeiten in rationale wie etwa beim Expandieren trigonometrischer Ausdrücke werden angewendet.

Macsyma und Reduce unterteilen ihre Simplifikationssysteme sogar in zwei Teile; einen fest eingebrannten Teil, der die als "standard quotients" (Reduce) bezeichneten rationalen Ausdrücke in

Kernen managt und einen regelbasierten, der die Beziehungen zwischen nicht-polynomialen Funktionssymbolen ausdrückt. In Macsyma liegt allerdings eine spezielle Schicht zwischen dieser als *Canonical Rational Expression* (CRE) bezeichneten Darstellung und der defaultmäßig verwendeten. Liegt ein Ausdruck als CRE vor, so erkennt man das an einer speziellen Markierung `/R/` in der Ausgabe. Für unseren Ausdruck erhalten wir dann

```
u:(sin(x)+cos(x))^5;
```

$$(\sin x + \cos x)^5$$

```
rat(u);
```

$$\sin^5 x + 5\,\cos x\,\sin^4 x + 10\,\cos^2 x\,\sin^3 x + 10\,\cos^3 x\,\sin^2 x + 5\,\cos^4 x\,\sin x + \cos^5 x$$

```
listratvars(u);
```

```
[cos(x), sin(x)]
```

und sehen, daß er in der Tat als Polynom in gewissen Kernen als verallgemeinerten Unbestimmten dargestellt wird. Noch deutlicher wird das an folgendem komplexerem Ausdruck

```
u:(exp(x)*cos(x)+cos(x)*sin(x)^4+2*cos(x)^3*sin(x)^2+cos(x)^5)/
(x^2-x^2*exp(-2*x))-(exp(-x)*cos(x)+cos(x)*sin(x)^2+cos(x)^3)/
(x^2*exp(x)-x^2*exp(-x));
```

$$\frac{\cos x\,\sin^4 x + 2\,\cos^3 x\,\sin^2 x + \cos^5 x + e^x\,\cos x}{x^2 - x^2\,e^{-2\,x}} - \frac{\cos x\,\sin^2 x + \cos^3 x + e^{-x}\,\cos x}{x^2\,e^x - x^2\,e^{-x}}$$

```
rat(u);
```

$$\frac{\begin{array}{c}(e^x)^2\,\cos x\,\sin^4 x + \left(2\,(e^x)^2\,\cos^3 x - e^x\,\cos x\right)\,\sin^2 x \\ + (e^x)^2\,\cos^5 x - e^x\,\cos^3 x + \left((e^x)^3 - 1\right)\,\cos x\end{array}}{x^2\,(e^x)^2 - x^2}$$

```
listratvars(u);
```

$$[x, e^x, \cos x, \sin x]$$

System	Rationale Normalform bilden
Macsyma	`rat(u)` wandelt in interne CRE um
Maple	`normal(u)`
Mathematica	`Together[u]`
MuPAD	`normal(u)`
Reduce	standardmäßig verwendet

Tabelle 2: Zur Berechnung von Normalformen in den verschiedenen Systemen

4.5 Simplifizieren trigonometrischer Ausdrücke und Regelsysteme

Die verschiedenen Additionstheoreme zeigen, daß zwischen den trigonometrischen Funktionen eine große Zahl von Abhängigkeiten bestehen. Die trigonometrischen Ausdrücke bilden damit eine Klasse, in der besonders viele verschiedene Darstellungen möglich und für die verschiedensten Zwecke auch nützlich sind.

Wir wollen diesen Effekt zunächst an einigen Beispielen studieren, in denen vom entsprechenden CAS selbst solche Umformungen eingesetzt werden. Dazu betrachten wir verschiedene Ausdrücke, die trigonometrische Funktionen enthalten, integrieren diese, bilden die Ableitung der so erhaltenen Stammfunktionen und untersuchen, ob die Systeme in der Lage sind zu erkennen, daß die so produzierten Ausdrücke mit den ursprünglichen Integranden zusammenfallen. Neben unserer eigentlichen Problematik erlaubt die Art der Antwort der einzelnen Systeme zugleich einen gewissen Einblick, wie diese die entsprechenden Integrationsaufgaben lösen.

Als Beipiele betrachten wir die Funktionen

$$
\begin{aligned}
f_1 &:= \sin(3x)\,\sin(5x),\\
f_2 &:= \cos\left(\tfrac{x}{2}\right)\,\cos\left(\tfrac{x}{3}\right),\\
f_3 &:= \sin\left(2x-\tfrac{\pi}{6}\right)\,\cos\left(3x+\tfrac{\pi}{4}\right),\\
f_4 &:= \frac{1}{\cos^4(x)},\\
f_5 &:= \frac{1}{\sin^2(x)\,(1-\cos(x))},\\
f_6 &:= \frac{1}{\sin(2x)\,(3\tan(x)+5)}
\end{aligned}
$$

Eine typische Antwort von Maple lautet etwa

$$
g_1 := \int f_1\,dx = \frac{\sin(2x)}{4} - \frac{\sin(8x)}{16}
$$

$$
g_1' := \frac{\cos(2x)}{2} - \frac{\cos(8x)}{2}
$$

woran wir das Vorgehen gut erkennen: Zur Berechnung des Integrals wurde das Produkt von Winkelfunktionen durch Anwenden der entsprechenden Additionstheoreme in eine Summe einzelner Winkelfunktionen umgewandelt, die man leicht integrieren kann. Wollen wir aber jetzt prüfen, daß $f_1 = g_1'$ gilt, so sind Winkelfunktionen mit verschiedenen Argumenten zu vergleichen. Hierzu müßten wir die Summe wieder in ein Produkt verwandeln, also die entsprechenden Regeln in der anderen Richtung anwenden.

Welcher Art sind die verwendeten Simplifikationsregeln in diesem Fall? Sie sollen Produkte in Summen verwandeln. Das erreicht man durch (mehrfaches) Anwenden der Regeln **Produkt-Summe**

$$
\begin{aligned}
\sin(x)\sin(y) &\Rightarrow \tfrac{1}{2}(\cos(x-y)-\cos(x+y))\\
\cos(x)\cos(y) &\Rightarrow \tfrac{1}{2}(\cos(x-y)+\cos(x+y))\\
\sin(x)\cos(y) &\Rightarrow \tfrac{1}{2}(\sin(x-y)+\sin(x+y))
\end{aligned}
$$

Diese Regeln sind invers zu den Regeln **Summe-Produkt**, die man beim Expandieren von Winkelfunktionen, deren Argumente Summen oder Differenzen sind, anwendet:

$$
\begin{aligned}
\sin(x+y) &\Rightarrow \sin(x)\cos(y)+\cos(x)\sin(y)\\
\cos(x+y) &\Rightarrow \cos(x)\cos(y)-\sin(x)\sin(y)\\
\sin(-x) &\Rightarrow -\sin(x)\\
\cos(-x) &\Rightarrow \cos(x)
\end{aligned}
$$

Bei der Formulierung des letzten Regelsatzes haben wir bereits berücksichtigt, daß Reduce (wie im übrigen auch die anderen CAS) eine Differenz $x - y$ als Summe $x + (-y)$ darstellt, wir also keine Simplifikationsregeln für eine *binäre* Operation MINUS, wohl aber für die *unäre* Operation MINUS angeben müssen.

Zur Anwendung der Produkt-Summe-Regeln als Funktionstransformationen ist der Ableitungsbaum des zu vereinfachenden Ausdrucks darauf zu untersuchen, ob Produkte von Funktionsausdrücken vorkommen, die die Funktionssymbole `sin` und `cos` enthalten, d.h. der Ableitungsbaum ist nach gewissen *Mustern* zu durchsuchen. Findet man eine solche Stelle, so sind die entsprechenden formalen Parameter x und y, die bei der Definition der Regeln als *Platzhalter* fungieren, durch die entsprechenden Teilausdrücke zu ersetzen und der Ableitungsbaum zu transformieren. Diese Transformationen sind so oft auszuführen, so lange sich noch die zu ersetzenden Muster im Ableitungsbaum finden lassen.

Wir haben also in einem solchen Regelsystem ebenfalls zwischen formalen Parametern während der Regeldeklaration und Belegung dieser Variablen während der Regelanwendung zu unterscheiden. Im Gegensatz zu Funktionsaufrufen ist bei der Zuordnung von Ausdrücken zu den einzelnen formalen Parametern einer Regel einige Arbeit zu leisten: Es muß das in der Regel angegebene Muster *erkannt* werden. Diesen Prozeß nennt man *Unifikation*. Weiterhin müssen wir wie generell im symbolischen Rechnen zwischen Variablen und Symbolen unterscheiden. Betrachten wir noch einmal das Regelsystem für `log`, das von Reduce verwendet wird, um diese Unterschiede zu verdeutlichen.

```
{log(1) => 0,
log(e) => 1,
log(e^~x) => x,
df(log(~x),~x) => 1/x}
```

Hier steht `e` für das Symbol e, die Basis des natürlichen Logarithmus, `~x` dagegen für einen formalen Parameter, wobei in der letzten Regel sogar an zwei Stellen des Musters derselbe Teilausdruck stehen muß.

Diese Feinheiten von Regelsystemen sind in der Form besonders gut in Reduce und Mathematica ausgeprägt. Wir wollen deshalb für diesen Abschnitt das System Reduce zugrunde legen.

Formulieren wir zunächst die obigen Regeln in Reduce-Notation. Die Tilde vor einer Variablen bedeutet, daß sie als formaler Parameter verwendet wird, d.h. durch einen beliebigen Ausdruck (= Teilbaum) beim Anwenden der Regel ersetzt werden kann.

```
trigsum:={ % Produkt-Summen-Regeln
        cos(~x)*cos(~y) => 1/2 * ( cos(x+y) + cos(x-y)),
        sin(~x)*sin(~y) => 1/2 * (-cos(x+y) + cos(x-y)),
        sin(~x)*cos(~y) => 1/2 * ( sin(x+y) + sin(x-y))};
trigexpand:={ % Summen-Produkt-Regeln
        sin(~x+~y) => sin x * cos y + cos x * sin y,
        cos(~x+~y) => cos x * cos y - sin x * sin y};
```

Die Regeln $\sin(-x) = -\sin(x)$ und $\cos(-x) = \cos(x)$ werden nicht benötigt, da dieser Teil der Simplifikation (gerade/ungerade Funktionen) als spezielle *Eigenschaft* der jeweiligen Funktion vermerkt ist und genau wie die Vereinfachung rationaler Funktionen gesondert behandelt wird.

Wendet man diese Regeln (mehrfach) auf eine Summe von Produkten von Winkelfunktionen `sin` und `cos` an, so sollte man am Ende eine Summe von Winkelfunktionen erhalten, die wegen der Linearität des Integraloperators für das Integrieren besonders geeignet ist. In der Tat wird beim einmaligen Anwenden der Regel auf ein Produkt dieses in zwei Summanden zerlegt, die selbst wieder Produkte sind, deren Faktorenzahl sich jedoch um 1 verringert hat. Leider klappt das nicht so, wie erwartet, wie etwa das Beispiel

```
sin(x)*sin(2x)*cos(3x) where trigsum;
```

```
                                    2
  - cos(6*x) + cos(4*x) - 2*sin(x)
-----------------------------------
                 4
```

zeigt. Hier ist der Ausdruck $\sin(x)^2$ nicht weiter vereinfacht worden, weil er nicht die Gestalt `mult(sin(A),sin(B))`, sondern `expt(sin(A),2)` hat. Wir müssen also noch Regeln hinzufügen, die es erlauben, auch $\sin(x)^n$ und analog $\cos(x)^n$ zu vereinfachen.

Solche Regeln können wir aus der Beziehung $\sin^2(x) = \frac{1-\cos(2x)}{2}$ (analog für $\cos(x)^2$) ableiten, indem wir einen solchen Faktor von $\sin(x)^n$ abspalten und auf die verbleibende Potenz $\sin(x)^{n-2}$ dieselbe Regel rekursiv anwenden. Ein derartiges rekursives Vorgehen ist typisch für Regelsysteme und erlaubt es, Simplifikationen zu definieren, deren Länge von einem der formalen Parameter (wie hier n) abhängig ist. Allerdings müssen wir dazu *konditionierte Regeln* formulieren, denn obige Ersetzung darf nur für ganzzahlige $n > 1$ angewendet werden, um den Abbruch der Rekursion zu sichern. Auch das ist in REDUCE möglich. Eine entsprechende Erweiterung der Regeln `trigsum` sähe dann so aus:

```
sin(~x)^(~n) => (1-cos(2x))/2*sin(x)^(n-2) when fixp n and (n>1),
cos(~x)^(~n) => (1+cos(2x))/2*cos(x)^(n-2) when fixp n and (n>1)
```

In Reduce können wir diese Regeln jedoch weiter vereinfachen, wenn wir den

Unterschied zwischen algebraischen und genauen Ersetzungsregeln

beachten:

```
Mathematica:
Expand[(a+1)^5] /. (a^2->x)
```

$$1 + 5\,a + 10\,a^3 + 5\,a^4 + a^5 + 10\,x$$

```
Reduce:
(a+1)^5 where (a^2 => x);

   2                      2
a*x  + 10*a*x + 5*a + 5*x  + 10*x + 1
```

Wir sehen, daß Mathematica nur solche Muster ersetzt hat, die *genau* auf den Ausdruck a^2 passen, während Reduce erkennt, daß auch in einem Ausdruck $a^k, k > 2$ der Faktor a^2 enthalten ist, also die eine Regel durch eine Serie von Regeln $(a^{k+2} \Rightarrow x \cdot a^k), k \geq 0$ ersetzt hat.

Unser gesamtes Regelsystem `trigsum` für Reduce lautet dann:

```
trigsum:={ % Produkt-Summen-Regeln
        sin(~x)*sin(~y) => 1/2*(cos(x-y)-cos(x+y)),
        sin(~x)*cos(~y) => 1/2*(sin(x-y)-sin(x+y)),
        cos(~x)*cos(~y) => 1/2*(cos(x-y)+cos(x+y)),
        cos(~x)^2 => (1+cos(2x))/2,
        sin(~x)^2 => (1-cos(2x))/2};
```

Die Anwendung dieses Regelsystems **Produkt-Summe** führt auf eine kanonische Darstellung polynomialer trigonometrischer Ausdrücke, ist also für Simplifikationszwecke hervorragend geeignet. Genauer gilt folgender

Satz 4 *Sei R ein Teilring der reellen Zahlen mit kanonischer Normalform. Dann kann jeder polynomiale Ausdruck $P(\sin(x), \cos(x))$ mit Koeffizienten aus R mit obigem Regelsystem* `trigsum` *in einen Ausdruck der Form*

$$\sum_{k>0} (a_k \sin(kx) + b_k \cos(kx)) + c$$

mit $a_k, b_k, c \in R$ verwandelt werden.

Diese Darstellung ist eindeutig, also auch eine kanonische Form für die Klasse der betrachteten Ausdrücke.

BEWEIS : Da das Regelsystem alle Produkte und Potenzen von Winkelsystemen ersetzt und sich in jedem elementaren Simplifikationsschritt die Anzahl der Multiplikationen verringert, ist nur die Eindeutigkeit der Darstellung zu zeigen. Die Koeffizienten a_k, b_k, c in einer Darstellung

$$f(x) = \sum_{k>0} (a_k \sin(kx) + b_k \cos(kx)) + c$$

kann man aber wie in der Theorie der Fourierreihen zurückgewinnen. Berechnen wir etwa für ein festes ganzzahliges $n > 0$ das Integral

$$\begin{aligned}
&2 \int_0^{2\pi} f(x) \sin(nx)\, dx \\
&\quad = \textstyle\sum_{k>0} a_k \int_0^{2\pi} 2 \sin(kx) \sin(nx)\, dx + \sum_{k>0} b_k \int_0^{2\pi} 2 \cos(kx) \sin(nx)\, dx + 2c \int_0^{2\pi} \sin(nx)\, dx \\
&\quad = \textstyle\sum_{k>0} a_k \int_0^{2\pi} (\cos((k-n)x) - \cos((k+n)x))\, dx \\
&\qquad\qquad + \sum_{k>0} b_k \int_0^{2\pi} (\sin((n-k)x) + \sin((n+k)x))\, dx \\
&\quad = 2\pi\, a_n,
\end{aligned}$$

so sehen wir, daß $f(x)$ jeden Koeffizienten a_n eindeutig bestimmt. Wir haben dabei von unseren Formeln Produkt-Summe und der Tatsache $\int_0^{2\pi} \sin(mx)\, dx = \int_0^{2\pi} \cos(mx)\, dx = 0$ für $m \neq 0$ Gebrauch gemacht. Analog ergeben sich die Koeffizienten b_n und c eindeutig aus der Funktion $f(x)$. □

Neben der Umwandlung von Potenzen der Winkelfunktionen in Summen mit Mehrfachwinkel-Argumenten ist an anderen Stellen, etwa beim Lösen goniometrischer Gleichungen, auch die umgekehrte Transformation von Interesse, da sie die Zahl der verschiedenen Funktionsausdrücke, die polynomial in einen Ausdruck eingehen, verringert und den Gesamtausdruck damit für die polynomiale Simplifikation, die ja die verschiedenen Funktionsausdrücke als Kerne betrachtet, vereinfacht. Die entsprechenden Formeln ergeben sich im Wesentlichen aus der *Moivreschen Formel* für komplexe Zahlen und den Potenzgesetzen: Aus

$$\cos(nx) + i \cdot \sin(nx) = e^{inx} = (e^{ix})^n = (\cos(x) + i \cdot \sin(x))^n$$

und den binomischen Formeln ergibt sich durch Vergleich der Real- und Imaginärteile unmittelbar

$$\cos(nx) = \sum_{2k \leq n} (-1)^k \binom{n}{2k} \sin(x)^{2k} \cos(x)^{n-2k}$$

und

$$\sin(nx) = \sum_{2k < n} (-1)^k \binom{n}{2k+1} \sin(x)^{2k+1} \cos(x)^{n-2k-1}$$

Zum Glück muß man diese komplizierten rechten Seiten, deren Länge wieder vom formalen Parameter n abhängt, und zu deren Formulierung man den bisher für die Formulierung von Regeln verwendeten Sprachumfang wesentlich erweitern müßte, ebenfalls nicht in dieser Form in das Regelwerk aufnehmen. Statt dessen folgen wir dem Ansatz

$$\sin(kx) = \sin((k-1)x + x) = \sin((k-1)x) \cos(x) + \cos((k-1)x) \sin(x),$$

den wir wieder rekursiv zur Anwendung bringen. Unser gesamtes Regelsystem hat dann folgende Gestalt:

```
trigexpand:={ % Produkt-Summen-Regeln
        sin(~x+~y) => sin x * cos y + cos x * sin y,
        cos(~x+~y) => cos x * cos y - sin x * sin y,
        sin(~k*~x) => sin((k-1)*x)*cos x+cos((k-1)*x)*sin x
                      when fixp k and k>1,
        cos(~k*~x) => cos((k-1)*x)*cos x-sin((k-1)*x)*sin x
                      when fixp k and k>1};
```

Diese Umformungsregeln erlauben es, komplizierte trigonometrische Ausdrücke, in deren Argumenten Summen und Mehrfachwinkel auftreten, durch trigonometrische Ausdrücke mit nur noch wenigen verschiedenen und einfachen Argumenten zu ersetzen. Hängen die Argumente nur ganzzahlig von einer Variablen x ab, so können wir das System wiederum zu einer kanonische Normalform erweitern. Genauer gesagt gilt der folgende Satz:

Satz 5 *Sei R ein Ring mit kanonischer Normalform. Dann kann man jeden polynomialen Ausdruck in $\sin(k\,x)$ und $\cos(k\,x)$, $k \in \mathbf{Z}$, mit Koeffizienten aus R durch obiges Regelsystem* `trigexpand` *und die zusätzliche Regel $\{\sin(x)^2 \Rightarrow 1 - \cos(x)^2\}$ in einen Ausdruck der Form*

$$P(\cos(x)) + \sin(x)\,Q(\cos(x))$$

verwandeln, wobei $P(z)$ und $Q(z)$ Polynome mit Koeffizienten aus R sind.

Diese Darstellung ist eindeutig, also eine kanonische Form für die Klasse der betrachteten Ausdrücke.

BEWEIS : Es ist wiederum nur die Eindeutigkeit zu zeigen.

$$P_1(\cos(x)) + \sin(x)\,Q_1(\cos(x)) = P_2(\cos(x)) + \sin(x)\,Q_2(\cos(x))$$

gilt aber genau dann, wenn $(P_1 - P_2)(\cos(x)) + \sin(x)\,(Q_1 - Q_2)(\cos(x))$ identisch verschwindet. Wir haben also nur zu zeigen, daß aus der Tatsache, daß $P(\cos(x)) + \sin(x)\,Q(\cos(x))$ identisch verschwindet, bereits folgt, daß $P(z)$ und $Q(z)$ beides Nullpolynome sind. Aus $P(\cos(x)) + \sin(x)\,Q(\cos(x)) = 0$ folgt aber nach der Substitution $x = -x$ auch $P(\cos(x)) - \sin(x)\,Q(\cos(x)) = 0$ und schließlich $P(\cos(x)) = Q(\cos(x)) = 0$. Da ein nichttriviales Polynom aber nur endlich viele Nullstellen besitzt, folgt die Behauptung. □

Mit diesen beiden Strategien lassen sich die ersten drei Aufgaben zufriedenstellend lösen. Einzige Schwierigkeit ist die Vereinfachung von $f_2 - g_2'$, da

$$g_2 := \int f_2\,dx = \frac{3}{5}\sin\left(\frac{5x}{6}\right) + 3\sin\left(\frac{x}{6}\right)$$

Winkel enthält, von denen (etwa für Reduce[3]) nicht auf den ersten Blick sichtbar ist, daß es sich um ganzzahlige Vielfache eines gemeinsamen Winkels handelt. Die explizite Substitution $x = 6y$, die das System darauf hinweist, hilft hier aber weiter.

Die klare Struktur der beiden Simplifikationssysteme verwendet implizit die Tatsache, daß Reduce standardmäßig polynomiale Ausdrücke in ihre distributive Normalform transformiert. Geht man etwa in Reduce zur faktorisierten Normalform über, so liefert `trigsum` nicht mehr unbedingt lineare Ausdrücke, da nach einem Simplifikationsschritt in dem entstehenden Ausdruck keine unmittelbaren Produkte von Winkelfunktionen mehr enthalten sein müssen.

[3] Hier wirkt sich aus, daß in Reduce rationale Funktionen eine zentrale Rolle spielen und deshalb Differenzen zwar verkappte Summen, aber Quotienten keine verkappten Produkte sind.

```
on factor;
sin(x)^4  where trigsum;

              2
 (cos(2*x) - 1)
-----------------
        4
```

Dies wirkt sich auf die Formulierung von Regelsystemen in solchen CAS aus, die generell auf Normalformen verzichten. Wir wollen diesen Aspekt an **Mathematica** demonstrieren. Vorab sei auf einige Besonderheiten der Syntax dieses Systems hingewiesen, die zum Verständnis der Formeln unerläßlich sind. Für eine detailliertere Einführung verweisen wir auf die einschlägige Systemliteratur.

Mathematica unterscheidet im Gegensatz etwa zu Reduce streng zwischen Groß- und Kleinschreibung. Die meisten Funktionsnamen beginnen jede "Silbe" mit einem Großbuchstaben. Ihre Argumentliste wird im Gegensatz zu anderen Systemen in eckige Klammern eingeschlossen. Runde Klammern dienen ausschließlich der Gruppierung von Teilausdrücken. Einstellige Funktionen wie etwa `Simplify, Together` oder `Expand` können auch in der Postfix-Notation `<Expr>//Expand` statt `Expand[<Expr>]` geschrieben werden, was der Auswertungsreihenfolge (erst die Argumente, dann der Funktionsaufruf) stärker entspricht als die übliche Präfixnotation. Für viele zweistellige Funktionen existieren Infix-Notationen in Operatorform, wie etwa für die Substitutionsoperatoren `/.` (einmalige Substitution) und `//.` (rekursive Substitution; beide unterscheiden sich genau wie Evaluationstiefe 1 und maximale Evaluationstiefe). Formale Parameter werden durch einen Unterstrich, etwa als `x_`, gekennzeichnet, dem im Gegensatz zu Reduce weitere Information über den Typ oder eine Default-Initialisierung folgen können. Deshalb dürfen, ebenfalls im Gegesatz zu Reduce, Unterstriche nicht in Variablennamen auftreten.

Obige Regelsysteme hätten damit in Mathematica folgende Gestalt:

```
trigexpand={ (* Expandiert Winkelfunktionen *)
        Cos[x_+y_] -> Cos[x]*Cos[y] - Sin[x]*Sin[y],
        Sin[x_+y_] -> Sin[x]*Cos[y] + Cos[x]*Sin[y],
        Cos[n_Integer*x_]/; (n>1)  ->
                Cos[(n-1)*x]*Cos[x]-Sin[(n-1)*x]*Sin[x],
        Sin[n_Integer*x_]/; (n>1)  ->
                Sin[(n-1)*x]*Cos[x]+Cos[(n-1)*x]*Sin[x]};

trigsum={ (* Verwandelt Produkte in Summen von Mehrfachwinkeln *)
        Cos[x_]*Cos[y_] -> 1/2*(Cos[x+y]+Cos[x-y]),
        Sin[x_]*Sin[y_] -> 1/2*(-Cos[x+y]+Cos[x-y]),
        Sin[x_]*Cos[y_] -> 1/2*(Sin[x+y]+Sin[x-y]),
        Sin[x_]^(n_Integer)/; (n>1) -> (1-Cos[2*x])/2*Sin[x]^(n-2) ,
        Cos[x_]^(n_Integer)/; (n>1) -> (1+Cos[2*x])/2*Cos[x]^(n-2) };
```

Wenden wir das zweite System auf $\sin(x)^7$ an, so erhalten wir nicht die aus unseren Rechnungen mit Reduce erwartete Normalform, sondern einen teilfaktorisierten Ausdruck:

```
Sin[x]^7//.trigsum
```

$$\frac{(1-\cos(2\,x))^3\,\sin(x)}{8}$$

Dieser Ausdruck muß erst expandiert werden, damit man die Regeln erneut anwenden kann

```
%//Expand;
%//.trigsum
```

$$\frac{\sin(x)}{8} + \frac{3\ (1+\cos(4\,x))\ \sin(x)}{16} - \frac{3\ (-\sin(x)+\sin(3\,x))}{16} - \frac{(1+\cos(4\,x))\ (-\sin(x)+\sin(3\,x))}{32}$$

usw., ehe nach vier solchen Schritten schlielich das Endergebnis

$$\frac{35\ \sin(x)}{64} - \frac{21\ \sin(3\,x)}{64} + \frac{7\ \sin(5\,x)}{64} - \frac{\sin(7\,x)}{64}$$

feststeht.

Wir benötigen also an dieser Stelle die Funktionalität von `Expand`, genauer gesagt ihre Distributivität auf Produkten, in Regel- und nicht in funktionaler Form. Das kann man durch eine einzige weitere Regel erreichen:

```
trigsum={ (* Verwandelt Produkte in Summen von Mehrfachwinkeln *)
        Cos[x_]*Cos[y_] -> 1/2*(Cos[x+y]+Cos[x-y]),
        Sin[x_]*Sin[y_] -> 1/2*(-Cos[x+y]+Cos[x-y]),
        Sin[x_]*Cos[y_] -> 1/2*(Sin[x+y]+Sin[x-y]),
        Sin[x_]^(n_Integer)/; (n>1) -> (1-Cos[2*x])/2*Sin[x]^(n-2) ,
        Cos[x_]^(n_Integer)/; (n>1) -> (1+Cos[2*x])/2*Cos[x]^(n-2) ,
        (a_+b_)*c_ -> a*c+b*c};
```

Nun zeigt die Simplifikation das erwünschte Verhalten:

```
Sin[x]^7//.trigsum
```

$$\frac{35\ \sin(x)}{64} - \frac{21\ \sin(3\,x)}{64} + \frac{7\ \sin(5\,x)}{64} - \frac{\sin(7\,x)}{64}$$

Da die beiden Regelsysteme `trigsum` und `trigexpand` zueinander entgegengesetzt sind, kann man nicht beide gleichzeitig in einem Simplifikationssystem zulassen, ohne Zirkelschlüsse zu verursachen. Dem Nutzer stehen in den verschiedenen Systemen dafür verschiedene fest eingebaute Simplifikationsstrategien zur Verfügung. Welche Freiheiten erlauben in dieser Richtung die einzelnen Systeme?

Derive: Über Knöpfe kann zwischen verschiedenen eingebauten Strategien gewechselt werden. Neben Expandieren und Faktorisieren polynomialer Ausdrücke (= verschiedene polynomiale Normalformen) kann zwischen Expandieren und Sammeln von Exponenten, von logarithmischen Ausdrücken und Winkelfunktionen gewählt werden. Letzteres entspricht den beiden hier vorgestellten Strategien, wobei noch gewählt werden kann, ob in Richtung $\sin(x)$ oder $\cos(x)$ simplifiziert werden soll.

Die vier "M-Systeme" **Macsyma, Maple, Mathematica** und **MuPAD** stellen eine ganze Reihe von Simplifikationsprozeduren mit verschiedenen Parametern zur Verfügung, die unterschiedliche Ziele ansteuern. In Maple sind dies insbesondere die Befehle `expand, combine, simplify` und `convert`.

Auf trigonometrische Ausdrücke angewendet bewirken `combine` (Produkte in Winkelsummen) und `expand` (Winkelsummen in Produkte) die obigen Umformungen, während man mit `convert` bzw. `rewrite` (u.a.) trigonometrische Funktionen und deren Umkehrfunktionen in Ausdrücke mit *exp* und *log* von komplexen Argumenten umformen kann. Dies entspricht dem REDUCE-Regelsatz

```
trigexp:={
        tan(~x) => sin(x)/cos(x),
        sin(~x) => (exp(i*x)-exp(-i*x))/(2*i),
        cos(~x) => (exp(i*x)+exp(-i*x))/2};
```

In der folgenden Tabelle sind die Namen der entsprechenden Funktionen in den einzelnen Systemen einander gegenübergestellt, die dieselben Transformationen wie oben beschrieben bewirken.

Wirkung	Macsyma	Maple	Mathematica	MuPAD
Argumentsummen auflösen	trigexpand	expand	TrigExpand	expand
Produkte zu Mehrfachwinkeln	trigreduce	combine	TrigReduce	combine
Trig $\mapsto$ Exp	exponentialize	convert(u,exp)	TrigToExp	rewrite(u,exp)
Exp $\mapsto$ Trig	demoivre	convert(u,trig)	ExpToTrig	?

Tabelle 3: Ausgewählte Transformationsfunktionen für Ausdrücke, die trigonometrische Funktionen enthalten.

Maple erlaubt es auch, innerhalb des Simplify-Befehls eigene Regelsysteme anzugeben, kennt dabei aber keine formalen Parameter. Statt dessen kann man für einzelne neue Funktionen den simplify-Befehl erweitern, was aber ohne interne Systemkenntnisse recht schwierig ist. Ähnlich kann man in MuPAD die Funktionsaufrufe `combine` und `expand` durch die Definition entsprechender Attribute auf andere Funktionssymbole ausdehnen.

Macsyma und Mathematica erlauben die Definition eigener Regelsysteme, mit denen man die intern vorhandenen Transformationsmöglichkeiten erweitern kann.

Dies gilt auch für **Axiom**, welches im Paket `TranscendentalManipulations` neben Umformungen von einer Winkelfunktion zu einer anderen die beiden Funktionen `expand` (Winkelsummen in Produkte) und `simplify` (alles in `sin` und `cos`) zur Verfügung stellt. Ein ähnlich umfangreicher Satz von Transformationsfunktionen wie in den anderen CAS fehlt allerdings.

Auch **REDUCE** kennt nur wenige eingebaute Regeln, was sich insbesondere für die Arbeit mit trigonometrischen Funktionen als nachteilig erweist. Wir haben aber in diesem Abschnitt gesehen, daß es nicht schwer ist, solche Regelsysteme selbst zu entwerfen. Jedoch muß der Nutzer dazu wenigstens grobe Vorstellungen über die interne Datenrepräsentation besitzen. Besonders günstig ist, daß man eigene Simplifikationsregeln nicht nur als global gültige, sondern seit der Version 3.5 auch als lokal gültige Regeln anwenden kann. Insbesondere letzteres erlaubt es, gezielt Umformungen mit überschaubaren Seiteneffekten auf Ausdrücken vorzunehmen.

Doch kehren wir zur Berechnung der 6 Beispielintegrale zurück. Hier sind die von DERIVE berechneten Ergebnisse im einzelnen aufgelistet:

$$g_1 := \frac{\sin(2x)}{4} - \frac{\sin(8x)}{16}$$

$$g_2 := \frac{3}{5}\sin\left(\frac{5x}{6}\right) + 3\sin\left(\frac{x}{6}\right)$$

$$g_3 := -\frac{\sqrt{6}+\sqrt{2}}{40}\cos(5x) + \frac{\sqrt{6}-\sqrt{2}}{40}\sin(5x) + \frac{\sqrt{6}-\sqrt{2}}{8}\cos(x) - \frac{\sqrt{6}+\sqrt{2}}{8}\sin(x)$$

$$g_4 := \frac{2}{3}\tan(x) + \frac{\sin(x)}{3\cos^3(x)}$$

$$g_5 := -\frac{2\cos^2(x) - 2\cos(x) - 1}{3\sin(x)\,(\cos(x)-1)}$$

$$g_6 := \frac{1}{10}\ln\left(\frac{\sin(x)}{5\cos(x)+3\sin(x)}\right)$$

REDUCE geht bei der Berechnung der ersten drei Integrale anders vor: Aus der Kenntnis der Form der Antwort wird diese mit unbestimmten Koeffizienten angesetzt und die konkreten Koeffizientenwerte werden ausgerechnet. Dies erkennen wir deutlich, wenn wir Integrale mit allgemeinen Koeffizienten eingeben

```
int(sin(a*x)*sin(b*x),x);
```

$$\frac{sin(a\,x)\,cos(b\,x)\,b\;-\;sin(b\,x)\,cos(a\,x)\,a}{a^2\;-\;b^2}$$

oder

```
int(sin(a*x)*sin(b*x)*sin(c*x),x);
```

$$\begin{aligned}&\big(\sin(ax)\sin(bx)\cos(cx)a^2c+\sin(ax)\sin(bx)\cos(cx)b^2c-\sin(ax)\sin(bx)\cos(cx)c^3\\&+\sin(ax)\sin(cx)\cos(bx)a^2b-\sin(ax)\sin(cx)\cos(bx)b^3+\sin(ax)\sin(cx)\cos(bx)bc^2\\&-\sin(bx)\sin(cx)\cos(ax)a^3+\sin(bx)\sin(cx)\cos(ax)ab^2+\sin(bx)\sin(cx)\cos(ax)ac^2\\&+2\cos(ax)\cos(bx)\cos(cx)abc\big)/(a^4-2a^2b^2-2a^2c^2+b^4-2b^2c^2+c^4)\end{aligned}$$

Damit vermeidet das System in den ersten drei Fällen, Winkelfunktionen mit neuen Argumenten einzuführen, was die nachfolgende Simplifikation natürlich wesentlich vereinfacht. Die Berechtigung zu solchem Vorgehen folgt aus dem Satz

Satz 6 *Sei $\{m_1, m_2, \ldots, m_k\}$ eine endliche (Multi-)Menge reeller Zahlen. Wir betrachten die Menge der Ausdrücke*

$$T := \{t_1(m_1 x)\;\cdots\;t_k(m_k x)\;\;:\;\;t_i \in \{\sin, \cos\}\}$$

und $L = L(T)$ die Menge der reellen Linearkombinationen von Produkten aus T.

Dann gibt es zu jedem $f(x) \in L$ einen Ausdruck $g(x) \in L$ und ein $c \in \mathbf{R}$ mit

$$\int f(x)\,dx = g(x) + c\,x,$$

d.h. das Integral von $f(x)$ läßt sich bis auf einen linearen Korrekturterm als polynomialer Ausdruck in im wesentlichen denselben Kernen darstellen wie $f(x)$.

Beweis : Wendet man auf $f(x) = t_1(m_1x)\cdots t_k(m_k x)$ unsere Regeln `trigsum` an, so erhält man eine Linearkombination von Ausdrücken $t(\sum_{i=1}^{k} \varepsilon_i m_i x)$ mit $t \in \{\sin, \cos\}$ und $\varepsilon_i = \pm 1$. Integration solcher Ausdrücke bleibt in der Klasse außer für $\sum \varepsilon_i m_i = 0$ und $t = \cos$. Expansion der Ausdrücke $t(\sum \varepsilon_i m_i x)$, aus denen das Integral zusammengesetzt ist, längs der "Sollbruchstellen" mit `trigexpand` führt wieder in die Klasse L zurück. □

REDUCE liefert damit die folgenden Antworten:

$$g_1 := \frac{3\sin(5x)\cos(3x) - 5sin(3x)cos(5x)}{16}$$

$$g_2 := \frac{6\big(-2\sin(\frac{x}{3})\cos(\frac{x}{2}) + 3\sin(\frac{x}{2})\cos(\frac{x}{3})\big)}{5}$$

$$g_3 := \frac{3\sin(\frac{12x-\pi}{6})\sin(\frac{12x+\pi}{4}) + 2\cos(\frac{12x-\pi}{6})\cos(\frac{12x+\pi}{4})}{5}$$

$$g_4 := \frac{\sin(x)\big(2\sin(x)^2 - 3\big)}{3\cos(x)\big(\sin(x)^2 - 1\big)}$$

$$g_5 := \frac{2\sin(x)^2 + 2\cos(x) - 1}{3\sin(x)\big(\cos(x) - 1\big)}$$

$$g_6 := \frac{-\log\big(3\tan(x) + 5\big) + \log\big(\tan(x)\big)}{10}$$

Die Berechnung der letzten drei Integrale erfolgte mit Hilfe eines allgemeinen Verfahrens, das darauf beruht, die Kerne $\sin(x)$, $\cos(x)$ durch $\tan(\frac{x}{2})$ auszudrücken. Dies ist mit folgendem Regelsatz möglich, nachdem alle anderen Winkelfunktionen allein durch `sin` und `cos` ausgedrückt sind:

```
trigtan:={
        sin(~x) => (2*tan(x/2))/(1+tan(x/2)^2),
        cos(~x) => (1-tan(x/2)^2)/(1+tan(x/2)^2)};
```

Die dazu inverse Regel

```
invtrigtan:={ tan(~x) => sin(2x)/(1+cos(2x) };
```

erlaubt es, Halbwinkel im Ergebnis wieder so weit als möglich zu eliminieren. Grundlage dieses Vorgehens ist die Fähigkeit der Systeme, rationale Funktionen in der Integrationsvariablen zu integrieren sowie der

Satz 7 *Sei* $R(z) = \frac{P(z)}{Q(z)}$ *eine rationale Funktion. Dann kann*

$$\int R(\tan(\frac{x}{2}))\,dx$$

durch die Substitution $y = \tan(\frac{x}{2})$ *auf die Berechnung des Integrals einer rationalen Funktion zurückgeführt werden.*

Beweis: Es gilt $\tan(x)' = 1 + tan(x)^2$ und folglich $dx = \frac{2\,dy}{1+y^2}$, womit wir

$$\int R(\tan(\frac{x}{2}))\,dx = \int \frac{R(y)}{1+y^2}\,dy$$

erhalten.

4.6 Das allgemeine Simplifikationsproblem

Betrachten wir zum Abschluß dieses Kapitels, wie sich die verschiedenen CAS bei der Simplifikation komplexerer Ausdrücke verhalten.

1. Beispiel:

```
u:=(exp(x)*cos(x)+cos(x)*sin(x)^4+2*cos(x)^3*sin(x)^2+cos(x)^5)/
(x^2-x^2*exp(-2*x))-(exp(-x)*cos(x)+cos(x)*sin(x)^2+cos(x)^3)/
(x^2*exp(x)-x^2*exp(-x));
```

Dieser nicht zu komplizierte Ausdruck enthält neben trigonometrischen auch Exponentialfunktionen. Hier sind offensichtlich die Regeln anzuwenden, die weitestgehend eine der Winkelfunktionen durch die andere ausdrücken. Als Ergebnis erhält man den Ausdruck

$$\frac{\cos(x)\,(e^x+1)}{x^2}$$

Eine genauere Analyse zeigt, daß hierfür einzig die Regel $\cos(x)^2 \Rightarrow 1 - \sin(x)^2$ anzuwenden ist.

Maple hatte noch in der Version V.3 bei diesem Ausdruck Schwierigkeiten, den kleinsten gemeinsamen Nenner zu erkennen:

```
simplify((exp(x)-exp(-x))/(1-exp(-2*x)));
```

lieferte

$$-\frac{e^x - e^{-x}}{-1 + e^{-2\,x}} \qquad \text{statt} \qquad e^x$$

Das ist in der neuen Version V.4 behoben.

Reduce benötigt, nach unseren bisherigen Ausführungen nicht unerwartet, als einzige Hilfestellung die oben angegebene Regel, während Derive, Mathematica und MuPAD ebenfalls bereits mit `Simplify` zu dem erwarteten Ergebnis kommen. Macsyma kann den Ausdruck mit einer speziellen Simplifikationsroutine für trigonometrische Funktionen ebenfalls zufriedenstellend vereinfachen.

2. Beispiel:

```
u:=16*cos(x)^3*cosh(x/2)*sinh(x)-6*cos(x)*sinh(x/2)-
6*cos(x)*sinh(3/2*x)-cos(3*x)*(e^(3/2*x)+e^(x/2))*(1-e^(-2*x));
```

Hier sind die Simplifikationsregeln für trigonometrische und hyperbolische Funktionen anzuwenden. Das kann man etwa durch Umformung in Exponentialausdrücke erreichen. Dieses Zusatzwissen erlaubt es, mit Reduce und den Regelsystemen `hyp2exp, trigexpand` und `trigsin` aus der Datei `trig` die Identität $u = 0$ zu erkennen.

Dasselbe Ergebnis erhielt man in Maple V.3 über

```
convert(u,exp);
expand(");
```

Um den Ausdruck auch in V.4 zu Null zu vereinfachen, müssen nun noch zusätzlich Potenzgesetze für `exp` angewendet werden:

```
combine(",exp);
```

Ähnlich ist in Macsyma und MuPAD vorzugehen, während in Mathematica wiederum ein Aufruf der Funktion `Simplify` genügt. Derive genügt ein Aufruf von Simplify, wenn man vorher `Trigonometry:=Expand` gesetzt hat.

System	Befehle	
Derive	(S)implify	(S)implify mit `Trigonometry:=Expand`
Macsyma	`rat(trigreduce(u1))`	`expand(exponentialize(u2))`
Maple	`simplify(u1)`	`combine(expand(convert(u2,exp)),exp)`
Mathematica	`u1//Simplify`	`u2//Simplify`
MuPAD	`simplify(u1)`	`combine(expand(rewrite(u2,exp)),exp)`
Reduce	`u1 where trigsin`	`u2 where hyp2exp,trigexpand,trigsin`

Tabelle 4: Simplifikation von u_1 und u_2 auf einen Blick

3. Beispiel (aus [5, S. 81])

```
u:=log(tan(x/2)+sec(x/2))-arcsinh(sin(x)/(1+cos(x)))
```

Diesen Ausdruck vermag keines der Systeme zu Null zu vereinfachen, obwohl auch hier das Vorgehen für einen Mathematiker mit geübtem Blick sehr transparent ist: Wegen $arcsinh(a) = \log(a+\sqrt{a^2+1})$ ist $arcsinh$ durch einen logarithmischen Ausdruck zu ersetzen und dann die beiden Logarithmen zusammenzufassen. Dies wird in Derive beim Vereinfachen automatisch vorgenommen, ist in Macsyma, Maple und MuPAD durch eingebaute Funktionen und in den anderen Systemen durch Angabe einer einfachen Regel realisierbar, etwa in Mathematica als

```
Arch2Log = { ArcSinh[x_] -> Log[x+Sqrt[1+x^2]] }
```

Vereinfacht man die Differenz u_1 dieser beiden Logarithmen mit `simplify`, so wartet nur Mathematica mit einem einigermaßen passablen Ergebnis auf:

$$\log\left(\sec(\frac{x}{2})+\tan(\frac{x}{2})\right)-\log\left(\sqrt{\sec(\frac{x}{2})^2}+\tan(\frac{x}{2})\right)$$

Hier greift wiederum die vorsichtige Simplifikation von $\sqrt{x^2}$, die hier allerdings aus dem Kontext heraus aufgelöst werden kann (wenn man voraussetzt, daß es sich um reelle Funktionen handelt): $\log(\sec(\frac{x}{2})+\tan(\frac{x}{2}))$ hat als Definitionsbereich $D=\{x : \cos(\frac{x}{2})>0\}$. Wendet man noch die entsprechende Regel

```
SimpSqrt = { Sqrt[x_^2] -> x }
```

an, so erhält man schließlich 0.

Um in den anderen Systemen ebenfalls die Nulläquivalenz des betrachteten Ausdrucks u_1 zu erkennen, ist es sinnvoll, die beiden Logarithmen zusammenzufassen. In Derive erfolgt dies automatisch und ist mit `logcontract` (Macsyma), `combine(u,ln)` (Maple und MuPAD) oder dem Schalter `combinelogs` (Reduce) auch in den anderen systemen auf einfache Weise möglich. Maple V.4 weigert sich allerdings, die Zusammenfassung $\ln(A)-\ln(B)\mapsto\ln(A/B)$ vorzunehmen, wenn nicht $A,B>0$ bekannt ist, da die Werte des Logarithmus sonst ja komplex werden und das Ergebnis verfälscht sein könnte. Statt dessen kann man die relevanten Teile des Ausdrucks u_1 extrahieren und daraus das gewünschte Argument u_2 komponieren:

```
u2 := op([1,1],u1)/op([2,2,1],u1);
```

In Mathematica benötigen wir für diese Transformation eine selbstdefinierte Regel

```
LogCombine = { Log[x_]-Log[y_] -> Log[x/y] }
```

Nachdem diese Hürde genommen ist entsteht als Argument ein komplizierter trigonometrischer Ausdruck

$$u_2:=\frac{\tan(\frac{x}{2})+\sec(\frac{x}{2})}{\frac{\sin(x)}{1+\cos(x)}+\sqrt{\frac{(\sin(x))^2}{(1+\cos(x))^2}+1}},$$

der mit den bisher betrachteten Regeln weiter vereinfacht werden kann. Mathematica liefert hier bereits mit `Simplify` ein verwertbares Ergebnis

$$\frac{1+\sin(\frac{x}{2})}{\cos(\frac{x}{2})\sqrt{\sec(\frac{x}{2})^2}+\sin(\frac{x}{2})},$$

das mit der Regel `simpsqrt` zu Eins vereinfacht.

Die anderen Systeme haben Schwierigkeiten, die Winkelargumente x und $x/2$ zueinander in Beziehung zu setzen. Ersetzen wir in u_2 deshalb noch $x\mapsto 2y$:

```
u3:=subs(x=2*y,u2);
```

Maple liefert nun mit

```
assume(y,real);
simplify(expand(u3));
```

```
                    sin(y~) + 1
-------------------------------------------
sin(y~) + signum(cos(y~))
```

ein Ergebnis, dem man die Nulläquivalenz wiederum fast ansieht. Dasselbe Ergebnis erhält man mit Derive, wenn man wieder `Trigonometry:=Expand` einstellt.

Macsyma liefert mit

```
trigsimp(trigexpand(u3));
```

$$\frac{\cos y\ \sin y + \cos y}{|\cos y| + \cos y\ \sin y};$$

ein Ergebnis ähnlicher Qualität. Man beachte, daß dieser einfache Ausdruck nur entsteht, wenn man, ähnlich wie in Derive, zuerst die Mehrfachwinkel auflöst und danach erst simplifiziert.

Mit Reduce kann man dieselben Umformungen durch geeignete Regelsysteme erreichen, die in der Datei `trig` enthalten sind

```
on combinelogs$
u1:=(u where archtolog)$
u2:=part(u1,1)$
u3:=sub(x=2*y,u2)$
u3 where trig;

     cos(y)*(sin(y) + 1)
----------------------------
 abs(cos(y)) + cos(y)*sin(y)
```

Bei MuPAD führt dieselbe Strategie zu keinem verwertbaren Ergebnis.

Wir sehen hieran bereits, daß man sich stets genügend komplizierte Simplifikationsprobleme ausdenken kann, die mit dem vorhandenen Instrumentarium nicht aufgelöst werden können. Es gilt jedoch noch mehr:

Satz 8 *Aus den Funktionssymbolen sin, abs und $*, +$ sowie der Konstanten π und ganzen Zahlen kann man Ausdrücke zusammenstellen, für die das Simplifikationsproblem algorithmisch unlösbar ist.*

Das allgemeine Simplifikationsproblem ist also sogar algorithmisch unlösbar.

Kapitel 5

Algebraische Zahlen

Wir hatten im vorangehenden Kapitel gesehen, daß sich Polynome mit Koeffizienten aus einem Grundbereich R, auf dem eine kanonische Form existiert, selbst stets in eine kanonische Form bringen lassen und so das Rechnen in diesen Strukturen besonders einfach ist. Dies gilt insbesondere für Polynome über den ganzen oder rationalen Zahlen, da für beide Bereiche offensichtliche kanonische Formen (etwa die Darstellung ganzer Zahlen mit Vorzeichen im Dezimalsystem oder die rationaler Zahlen als unkürzbare Brüche mit positivem Nenner) existieren.

Treten als Koeffizienten Wurzelausdrücke wie $\sqrt{2}$ oder $\sqrt{2+\sqrt{3}}$ auf, so kann man die Frage nach einer kanonischen oder wenigstens normalen Form schon nicht mehr so einfach beantworten. Eine solche Darstellung müßte ja wenigstens die bereits früher betrachteten Identitäten

$$\sqrt{2\sqrt{3}+4} = 1+\sqrt{3}$$

oder

$$\sqrt{11+6\sqrt{2}}+\sqrt{11-6\sqrt{2}} = 6$$

erkennen, wenn die entsprechenden Wurzelausdrücke als Koeffizienten auftreten. Während dies in Maple und Derive automatisch erfolgt, sind die anderen Systeme nur mit viel gutem Zureden bereit, diese Simplifikation vorzunehmen. Selbst das Normalformproblem, also jeweils die Vereinfachung der Differenz beider Seiten der Identitäten zu 0, wird weder in MuPAD noch in Mathematica oder Macsyma allein durch den Aufruf von `simplify` gelöst. MuPAD (`radsimp`), Macsyma (`denest_sqrt`) und Mathematica 3.0 (`RootReduce`) stellen spezielle Simplifikationsroutinen für geschachtelte Wurzelausdrücke zur Verfügung, was bei Mathematica im stärkeren `FullSimplify` integriert ist. Mit Reduce und Axiom habe ich keinen direkten Weg gefunden, diese Aufgabe zu lösen.[1].

Axiom	unbekannt
Derive	automatisch
Macsyma	`denest_sqrt`
Maple	automatisch
Mathematica	`RootReduce`
MuPAD	`radsimp`
Reduce	nutzerdefiniertes Regelsystem

Tabelle 1: Vereinfachung geschachtelter Wurzelausdrücke

Faktorisierte kanonische Formen können in solchen Strukturen gänzlich fehlen, weil der Satz von der eindeutigen Primfaktorzerlegung nicht gelten muß. So kann man etwa im Ring $R = \mathbf{Z}[\sqrt{-5}]$ die

[1] Dieses Wissen kann man allerdings wiederum wenigstens Reduce mit einem einfachen Regelsystem beibringen.

Zahl 6 auf zwei (wesentlich voneinander verschiedene) Arten in Primfaktoren zerlegen:

$$6 = 2 \cdot 3 = (1 + \sqrt{-5})(1 - \sqrt{-5})$$

Gleichwohl hatten wir gesehen, daß solche Wurzelausdrücke im symbolischen Rechnen eine wichtige Rolle spielen, weil durch sie "interessante" reelle Zahlen dargestellt werden, die z.B. zur exakten Beschreibung von Nullstellen benötigt werden. Wir wollen deshalb zunächst untersuchen, wie die einzelnen Systeme mit univariaten Polynomen, in deren Koeffizienten solche Wurzelausdrücke auftreten, zurechtkommen. Wir werden dazu im folgenden mit Hilfe der verschiedenen CAS zunächst exemplarisch, ausgehend vom Polynom $f(x) = x^3 + x + 1$ dessen drei Nullstellen x_1, x_2, x_3 bestimmen, danach aus diesen von dem entsprechenden System selbst erzeugten Wurzelausdrücken das Produkt $(x - x_1)(x - x_2)(x - x_3)$ formen und schließlich versuchen, dieses Produkt durch Ausmultiplizieren und Vereinfachen wieder in das Ausgangspolynom $f(x)$ zu verwandeln. Schließlich werden wir versuchen, kompliziertere polynomiale Ausdrücke in x_1, x_2, x_3 zu vereinfachen.

Diese Rechnungen, ausgeführt in verschiedenen großen CAS allgemeiner Ausrichtung, sollen zugleich ein wenig von der Art und Weise vermitteln, wie man in jedem der Systeme die entsprechenden Rechnungen veranlassen kann.

5.1 Rechnen mit Nullstellen dritten Grades

5.1.1 Nullstellen dritten Grads mit Derive

Derive stellt im Gegensatz zu den anderen großen CAS im Rahmen seines Interpreters nur einen sehr eingeschränkten Befehlssatz bereit, den der Nutzer im Normalfall auch nicht verwenden soll. Statt dessen erfolgt der Dialog über eine menügesteuerte Oberfläche, die verschiedene Aktionen auf dem Arbeitsblatt ermöglicht. Dazu ist es insbesondere auf einfache Weise möglich, über Kursortasten auf beliebige Teilausdrücke bisher erzeugter Ausgaben (entsprechend dem jeweiligen Ableitungsbaum) zuzugreifen.

Wir wollen deshalb die Elemente dieser menügesteuerten Oberfläche zur Ausführung unserer Rechnungen verwenden. Die Namen der entsprechenden Knöpfe sind im folgenden als Funktionsaufruf dargestellt. Derive (Version 3.04) liefert die folgenden Antworten:

```
1: x^3+x+1
```

```
2: factor complex #1: (0.4 s)
```

$$\begin{aligned}
&\left(x + \left(\tfrac{1}{18}\sqrt{93} + \tfrac{1}{2}\right)^{1/3} - \left(\tfrac{1}{18}\sqrt{93} - \tfrac{1}{2}\right)^{1/3}\right) \\
&\quad \cdot \left(x + \left(\tfrac{1}{144}\sqrt{93} - \tfrac{1}{16}\right)^{1/3} - \left(\tfrac{1}{144}\sqrt{93} + \tfrac{1}{16}\right)^{1/3}\right. \\
&\qquad \left. - i\left(\left(\tfrac{3}{128}\sqrt{93} + \tfrac{29}{128}\right)^{1/6} + \left(\tfrac{29}{128} - \tfrac{3}{128}\sqrt{93}\right)^{1/6}\right)\right) \\
&\quad \cdot \left(x + \left(\tfrac{1}{144}\sqrt{93} - \tfrac{1}{16}\right)^{1/3} - \left(\tfrac{1}{144}\sqrt{93} + \tfrac{1}{16}\right)^{1/3}\right. \\
&\qquad \left. + i\left(\left(\tfrac{3}{128}\sqrt{93} + \tfrac{29}{128}\right)^{1/6} + \left(\tfrac{29}{128} - \tfrac{3}{128}\sqrt{93}\right)^{1/6}\right)\right)
\end{aligned}$$

```
3: expand #2 (1.3 s)
```

$$
\begin{aligned}
x^3 + x\left(\tfrac{1}{1152}\sqrt{93} + \tfrac{29}{3456}\right)^{1/3} + x\left(\tfrac{29}{3456} - \tfrac{1}{1152}\sqrt{93}\right)^{1/3} + x\left(\tfrac{3}{128}\sqrt{93} + \tfrac{29}{128}\right)^{1/3} \\
+x\left(\tfrac{29}{128} - \tfrac{3}{128}\sqrt{93}\right)^{1/3} - x\left(\tfrac{1}{18}\sqrt{93} + \tfrac{29}{54}\right)^{1/3} - x\left(\tfrac{29}{54} - \tfrac{1}{18}\sqrt{93}\right)^{1/3} \\
+x + \left(\tfrac{7}{7776}\sqrt{93} + \tfrac{5}{576}\right)^{1/3} + \left(\tfrac{1}{31104}\sqrt{93} - \tfrac{1}{3456}\right)^{1/3} + \left(\tfrac{7}{288}\sqrt{93} + \tfrac{15}{64}\right)^{1/3} \\
+ \left(\tfrac{1}{1152}\sqrt{93} - \tfrac{1}{128}\right)^{1/3} + \left(\tfrac{1}{486}\sqrt{93} + \tfrac{1}{54}\right)^{1/3} - \left(\tfrac{1}{31104}\sqrt{93} + \tfrac{1}{3456}\right)^{1/3} \\
- \left(\tfrac{7}{7776}\sqrt{93} - \tfrac{5}{576}\right)^{1/3} - \left(\tfrac{1}{1152}\sqrt{93} + \tfrac{1}{128}\right)^{1/3} - \left(\tfrac{7}{288}\sqrt{93} - \tfrac{15}{64}\right)^{1/3} \\
- \left(\tfrac{1}{486}\sqrt{93} - \tfrac{1}{54}\right)^{1/3}
\end{aligned}
$$

Dritte Wurzeln, hier als $(U)^{\frac{1}{3}}$ geschrieben, treten als Funktionssymbole auf, so daß es sich bei diesem univariaten Polynom intern in Wirklichkeit um ein Polynom in mehreren Kernen handelt, zwischen denen aber im Gegensatz zu den trigonometrischen Ausdrücken, die wir im letzten Kapitel betrachtet hatten, algebraische Relationen bestehen. Außerdem ist der Radikand eines solchen Wurzelausdrucks selbst wieder von komplizierterer polynomialer Gestalt.

> Das Wurzelsymbol wirkt damit, wie generell das Funktionssymbol eines Kerns, wie eine Trennwand zwischen dem (äußeren) rationalen Ausdruck, in den der Wurzelausdruck als Kern eingeht, und dem (inneren) rationalen Ausdrucken, der als Radikand, d.h. als Argument des Funktionsausdrucks, auftritt.

Diese für die Simplifikation von Funktionsausdrücken typische Situation muß durch das Simplifikationssystem aufgelöst werden. Es muß also nicht nur in der Lage sein, algebraische Abhängigkeiten, die zwischen verschiedenen solchen Kernen bestehen, zu erkennen, sondern diese auch mit der inneren und äußeren rationalen Arithmetik verbinden.

Man beachte allerdings, daß in obigem Ausdruck (bis auf rationale Vielfache des Radikanden) im wesentlichen nur zwei verschiedene dritte Wurzeln auftreten. Es sollte also nicht so schwierig sein, diesen Ausdruck zu simplifizieren. Jedoch Fehlanzeige! Offensichtlich werden von Derive intern die Kerne der Form `expt(<expr>,1/3)` *sämtlich* als verschieden angesehen, was bei einer Interpretation als komplexe Zahlen, ohne die Geschichte ihrer Entstehung zu kennen, auch korrekt wäre, da die entsprechende komplexe Funktion `expt` mehrdeutig ist.

```
4: simplify #3 (16.1 s)
```

$$
\begin{aligned}
x^3 + x\big(&\left(\tfrac{29}{3456} + \tfrac{1}{1152}\sqrt{93}\right)^{1/3} + \left(\tfrac{29}{3456} - \tfrac{1}{1152}\sqrt{93}\right)^{1/3} + \left(\tfrac{3}{128}\sqrt{93} + \tfrac{29}{128}\right)^{1/3} \\
&+ \left(\tfrac{29}{128} - \tfrac{3}{128}\sqrt{93}\right)^{1/3} - \left(\tfrac{1}{18}\sqrt{93} + \tfrac{29}{54}\right)^{1/3} - \left(\tfrac{29}{54} - \tfrac{1}{18}\sqrt{93}\right)^{1/3} + 1\big) \\
&- \left(\tfrac{1}{31104}\sqrt{93} + \tfrac{1}{3456}\right)^{1/3} + \left(\tfrac{1}{31104}\sqrt{93} - \tfrac{1}{3456}\right)^{1/3} - \left(\tfrac{1}{1152}\sqrt{93} + \tfrac{1}{128}\right)^{1/3} \\
&+ \left(\tfrac{1}{1152}\sqrt{93} - \tfrac{1}{128}\right)^{1/3} - \left(\tfrac{7}{7776}\sqrt{93} - \tfrac{5}{576}\right)^{1/3} + \left(\tfrac{1}{486}\sqrt{93} + \tfrac{1}{54}\right)^{1/3} \\
&- \left(\tfrac{1}{486}\sqrt{93} - \tfrac{1}{54}\right)^{1/3} - \left(\tfrac{7}{288}\sqrt{93} - \tfrac{15}{64}\right)^{1/3} + \left(\tfrac{14}{243}\sqrt{93} + \tfrac{5}{9}\right)^{1/3}
\end{aligned}
$$

Eine numerische Näherung der Summen der hier versammelten Wurzelausdrücke

```
5: approximate #4 (0.1 s)
```

$$x^3 + x + 1$$

zeigt gleichwohl, daß unsere Rechnungen bis hierher richtig waren, also einzig das Simplifizieren der entsprechenden Ausdrücke die Fähigkeiten des CAS übersteigt, obwohl Derive beim Vereinfachen geschachtelter Quadratwurzeln noch so gut abschnitt. Das Hauptproblem beim Simplifizieren dieser Ausdrücke besteht, wie bereits angemerkt, darin, daß sich Derive offensichtlich nicht merkt, daß die im wesentlichen gleichen Kerne der Gestalt $(29 \pm 3\sqrt{93})^{\frac{1}{3}}$ von derselben Quelle herrühren. Damit wird die in diesem Fall sogar rationale Abhängigkeit, die zwischen ihnen besteht, nicht verwendet.

5.1.2 Nullstellen dritten Grades mit Maple und MuPAD

Im weiteren wollen wir die sprachlichen Mittel, die die einzelnen Interpreter anbieten, zur Formulierung von Anfragen intensiver nutzen. So werden wir etwa die in allen Systemen vorhandene Funktion `solve` verwenden, mit der man die Nullstellen von Polynomen und allgemeineren Gleichungen und Gleichungssystemen bestimmen kann. Allerdings ist das Ausgabeformat dieser Funktion von System zu System verschieden und differiert selbst zwischen unterschiedlichen Typen von Gleichungen und Gleichungssystemen.

Maple liefert etwa für univariate Polynome, die wir in diesem Abschnitt untersuchen, eine *Ausdruckssequenz* zurück, die die verschiedenen Nullstellen zusammenfaßt. Wir wollen sie durch Einklammern `[...]` gleich in eine Liste verwandeln:

```
s:=[solve(x^3+x+1,x)];
```

$$[-1/6\,\%1 + 2\,\%2,\ 1/12\,\%1 - \tfrac{1}{\%1} + 1/2\,I\sqrt{3}\,(-1/6\,\%1 - 2\,\%2),$$
$$1/12\,\%1 - \tfrac{1}{\%1} - 1/2\,I\sqrt{3}\,(-1/6\,\%1 - 2\,\%2)]$$

$$\%1 := \sqrt[3]{108 + 12\sqrt{93}}$$
$$\%2 := \frac{1}{\sqrt[3]{108 + 12\sqrt{93}}}$$

Wir sehen, daß Maple das Problem der voneinander abhängigen Kerne auf eine clevere Art und Weise löst: Das Endergebnis wird unter Verwendung von neuen Variablen %1 und %2 formuliert, mit denen ein gemeinsamer Teilausdruck, der in der Formel mehrfach vorkommt, identifiziert wird. Auf diese Information wird bei späteren Simplifikationen zurückgegriffen, was für unsere Aufgabenstellung bereits ausreichend ist:

```
product(x-op(i,s),i=1..3);
```

$$(x + 1/6\,\%1 - 2\,\%2)\left(x - 1/12\,\%1 + \%2 - 1/2\,I\sqrt{3}\,(-1/6\,\%1 - 2\,\%2)\right)$$
$$\left(x - 1/12\,\%1 + \%2 + 1/2\,I\sqrt{3}\,(-1/6\,\%1 - 2\,\%2)\right)$$

```
p:=expand(");
```

$$1/2 + x + x^3 + \frac{1}{18}\sqrt{93} - \frac{8}{108 + 12\sqrt{93}}$$

Expand ist ein polynomialer Normalformoperator, d.h. er multipliziert Produkte aus und faßt gleichartige Terme zusammen, wobei x und %1 als Kerne betrachtet werden. Die separierten Teilausdrücke können vom Nutzer angesprochen werden, so daß wir diese Umformungen nachvollziehen können:

```
%1;
```

$$\sqrt[3]{108+12\sqrt{93}}$$

```
s1:=subs(%1=u,%2=1/u,s);
```

$$s1 := [-1/6\,u+2\,u^{-1}, 1/12\,u-u^{-1}+1/2\,I\sqrt{3}\left(-1/6\,u-2\,u^{-1}\right), \\ 1/12\,u-u^{-1}-1/2\,I\sqrt{3}\left(-1/6\,u-2\,u^{-1}\right)]$$

```
expand(product(x-op(i,s1),i=1..3));
```

$$x+x^3+\frac{1}{216}\,u^3-8\,u^{-3}$$

Durch Rationalmachen, d.h. durch Multiplizieren mit dem zu u konjugierten Wurzelausdruck, kann man das Absolutglied des Polynoms p weiter vereinfachen. Diese Technik ist jedoch ein spezieller mathematischer Trick, der in der polynomialen Normalformbildung rationaler Ausdrücke nicht enthalten ist. Er wird folglich bei Anwendung von *expand* auch nicht ausgeführt. Die folgende Anweisung führt schließlich zum gewünschten Ergebnis.

```
simplify(p);
```

$$x^3+x+1$$

Experimentieren wir weiter mit den Ausdrücken aus der Sequenz s. Es stellt sich heraus, daß Potenzsummen der drei Wurzeln stets ganzzahlig sind. Wir wollen wiederum prüfen, wieweit Maple in der Lage ist, dies zu erkennen. Dazu sammeln wir in einer Liste die Ergebnisse der Vereinfachung von $x_1^k+x_2^k+x_3^k$ für $k=2,\ldots,9$ auf und versuchen Sie danach weiter auszuwerten.

```
a:=[seq(simplify(sum(s[i]^k,i=1 .. 3)),k=2 .. 9)];
```

$$[-2,-3,2,5,6\,\frac{29+3\sqrt{3}\sqrt{31}}{\left(9+\sqrt{3}\sqrt{31}\right)^2},-7,-36\,\frac{29+3\sqrt{3}\sqrt{31}}{\left(9+\sqrt{3}\sqrt{31}\right)^2},144\,\frac{135+14\sqrt{3}\sqrt{31}}{\left(9+\sqrt{3}\sqrt{31}\right)^3}]$$

```
map(evalf,a);
        [-2., -3., 2., 5., 1.000000000, -7., -6.000000001, 6.000000002]

map(normal,a,expanded);
                          [-2, -3, 2, 5, 1, -7, -6, 6]
```

Wir sehen, daß die Potenzsummen offensichtlich immer ganzzahlig sind, Maple jedoch nicht immer in der Lage ist, mit dem Aufruf von `simplify` Nenner rational zu machen. Erst die explizite Aufforderung, die Nenner der einzelnen Ausdrücke zu expandieren (die man glücklicherweise und trickreich formulieren kann), führt zum erwarteten Ergebnis. Nebenbei sei bemerkt, daß hier genau dieselben Umformungen notwendig sind wie zur Vereinfachung geschachtelter Wurzelausdrücke, die auch Maple vorhin noch so bravourös von allein bewerkstelligt hat.

Ähnlich ist in **MuPAD** vorzugehen:

```
s:=solve(x^3+x+1,x);
s1:=_mult(x-op(s,i) $i=1..3);
expand(s1);
```

$$x^3+3x\sqrt[3]{\left(\frac{\sqrt{31}\sqrt{108}}{108}+1/2\right)}\sqrt[3]{\left(\frac{\sqrt{31}\sqrt{108}}{108}-1/2\right)}+1$$

```
simplify(%);
```

$$x^3+3x\sqrt[3]{\left(\frac{\sqrt{93}}{18}+1/2\right)}\sqrt[3]{\left(\frac{\sqrt{93}}{18}-1/2\right)}+1$$

Die Ergebnisse der ersten beiden Aufrufe sind hier weggelassen; sie haben ähnlich kompliziertes Aussehen wie in Maple. Auch hier werden gemeinsame Teilausdrücke separiert, wie man mit der (undokumentierten) Funktion `rationalize` erkennt:

```
rationalize(s);
```

$$\left\{\frac{\mathrm{X73}}{2}-\frac{\mathrm{X74}}{2}+\frac{\mathrm{X72X75\,(X73+X74)}}{2},\frac{\mathrm{X73}}{2}-\frac{\mathrm{X74}}{2}-\frac{\mathrm{X72X75\,(X73+X74)}}{2},-\mathrm{X73}+\mathrm{X74}\right\},$$

$$\left\{\mathrm{X75}=\sqrt{3},\mathrm{X72}=i,\mathrm{X73}=\sqrt[3]{\frac{\sqrt{31}\sqrt{108}}{108}+1/2},\mathrm{X74}=\sqrt[3]{\frac{\sqrt{31}\sqrt{108}}{108}-1/2}\right\}$$

Das Ergebnis von `expand` zeigt eine weitere Schwierigkeit, die hier aus der mathematischen Strenge resultiert: Die beiden Wurzelausdrücke, die im übrigen aus einer gemeinsamen Quelle stammen, wie wir weiter unten noch sehen werden, werden nicht zusammengefaßt, weil das im Komplexen im Prinzip falsch werden kann. Diese Funktionstransformation, die Verwandlung der "äußeren" in eine "innere" Multiplikation, kann durch `radsimp` bewerkstelligt werden, was aber eine sehr zeitaufwendige Sache ist. Auch für die Berechnung der Potenzsummen der Nullstellen benötigen wir diese Vereinfachung und erhalten (allerdings erst nach 216 s.)

```
[radsimp(_plus(op(s,i)^k $ i=1 .. 3)) $ k=2 .. 9];
```

```
[-2, -3, 2, 5, 1, -7, -6, 6]
```

5.1.3 Nullstellen dritten Grads mit Mathematica, Macsyma und Axiom

Mathematica hat seinen Gleichungslöser in der neuen Version 3.0 gründlich überarbeitet und liefert für unsere Gleichung dritten Grades folgende Antwort:

```
s=Solve[x^3+x+1==0,x]
```

$$\left\{ \{x \to -\left(\frac{2}{3\left(-9+\sqrt{93}\right)}\right)^{\frac{1}{3}} + \frac{\left(\frac{-9+\sqrt{93}}{2}\right)^{\frac{1}{3}}}{3^{\frac{2}{3}}}\},\right.$$

$$\{x \to \frac{-\left(1+i\sqrt{3}\right)\left(\frac{-9+\sqrt{93}}{2}\right)^{\frac{1}{3}}}{2\,3^{\frac{2}{3}}} + \frac{1-i\sqrt{3}}{2^{\frac{2}{3}}\left(3\left(-9+\sqrt{93}\right)\right)^{\frac{1}{3}}}\},$$

$$\left.\{x \to \frac{-\left(1-i\sqrt{3}\right)\left(\frac{-9+\sqrt{93}}{2}\right)^{\frac{1}{3}}}{2\,3^{\frac{2}{3}}} + \frac{1+i\sqrt{3}}{2^{\frac{2}{3}}\left(3\left(-9+\sqrt{93}\right)\right)^{\frac{1}{3}}}\}\right\}$$

Das Ergebnis des `Solve`-Operators ist keine Liste oder Menge von Lösungen, sondern eine

Substitutionsliste, in der für jede Lösung die Variablen und zugehörigen Werte in einer Form zusammengefaßt sind, die sich unmittelbar als Eingabe für den Substitutionsoperator eignet.

Solche Substitutionslisten werden auch von Maple und MuPAD für Gleichungssysteme in mehreren Veränderlichen verwendet, wo man die einzelnen Lösungskomponenten verschiedenen Variablen zuordnen muß. Mathematicas Ausgabeformat ist in dieser Hinsicht also, wie auch das von Axiom, Macsyma und Reduce, konsistenter.

Allerdings wird in diesen Systemen für Substitutionen dasselbe Gleichheitszeichen wie für Gleichungen verwendet (wie etwa im Maple-Befehl `subst(x=3,f(x))`), während Mathematica auch äußerlich zwischen dem Gleichheitszeichen `==` und dem Substitutionsoperator `->` unterscheidet.

Die einzelnen Komponenten der Lösung können wir durch die Aufrufe `x /. s[[i]]` erzeugen, welche jeweils die in der Lösungsliste aufgesammelten Substitutionen ausführen. Diese darf sich im Produkt $\prod_{i=1}^{3}(x-x_i)$ natürlich nur auf die Berechnung von x_i ausdehnen. Durch entsprechende Klammerung erreichen wir, daß im Ausdruck `x-(x /. s[[i]])` das erste x als Symbol erhalten bleibt, während für das zweite x die entsprechende Ersetzung $x \mapsto x_i$ ausgeführt wird.

Hängen wir noch eine Simplifikation an, so erhalten wir bereits das Polynom $f(x)$, von dem wir ausgegangen waren, zurück.

```
Product[x-(x /. s[[i]]),{i,1,3}]//Simplify
```

$$1 + x + x^3$$

Auch für die Potenzsummen der Nullstellen, die man mit dem Sequenzierungsoperator `Table` aufsammeln kann, bekommt man auf diesem Wege eine Liste von ganzen Zahlen:

```
Table[Simplify[Sum[(x/.s[[i]])^k,{i,1,3}]],{k,2,9}]
```

$$\{-2, -3, 2, 5, 1, -7, -6, 6\}$$

Ähnlich gehen die Systeme Axiom und Macsyma vor, wobei hier oft eine genauere Kenntnis speziellerer Designmomente notwendig ist. So liefert etwa **Axiom** mit

```
s:=solve(x^3+x+1)
```

das etwas verwunderliche Ergebnis $[x^3 + x + 1 = 0]$. Erst die explizite Aufforderung nach einer Lösung in Radikalen

```
s:=radicalSolve(x^3+x+1)
```

$$\left[x = \frac{(-3\sqrt{-3}+3)\sqrt[3]{\frac{\sqrt{\frac{31}{3}}-3}{6}}^2+2}{(3\sqrt{-3}+3)\sqrt[3]{\frac{\sqrt{\frac{31}{3}}-3}{6}}},\ x = \frac{(-3\sqrt{-3}-3)\sqrt[3]{\frac{\sqrt{\frac{31}{3}}-3}{6}}^2-2}{(3\sqrt{-3}-3)\sqrt[3]{\frac{\sqrt{\frac{31}{3}}-3}{6}}},\ x = \frac{3\sqrt[3]{\frac{\sqrt{\frac{31}{3}}-3}{6}}^2-1}{3\sqrt[3]{\frac{\sqrt{\frac{31}{3}}-3}{6}}}\right]$$

wartet mit einem Ergebnis auf, das dem der anderen Systeme ähnelt. Die Vereinfachung des Produkts sowie die Berechnung der Potenzsummen geschieht ähnlich wie in Maple über die Konversion von Listen in Produkte bzw. Summen und bedarf keiner zusätzlichen Simplifikationsaufrufe:

```
reduce(*,[x-subst(x,s.i) for i in 1..3])
```

```
  3
x  + x + 1
                                    Type: Expression Integer
```

```
[reduce(+,[subst(x**k,s.i) for i in 1..3]) for k in 2..9]
```

```
[- 2,- 3,2,5,1,- 7,- 6,6]
                                    Type: List Expression Integer
```

Macsyma[2] liefert als Ergebnis des `solve`-Operators die Lösungsmenge in Form einer Substitutionsliste in ähnlicher Form wie auch Axiom.

```
s:solve(x^3+x+1,x);
```

$$\left\{x = \sqrt[3]{\frac{\sqrt{31}}{6\sqrt{3}}-\frac{1}{2}}\left(-\frac{\sqrt{3}\,i}{2}-\frac{1}{2}\right)-\frac{\frac{\sqrt{3}\,i}{2}-1/2}{3\sqrt[3]{\frac{\sqrt{31}}{6\sqrt{3}}-1/2}},\ x = \sqrt[3]{\frac{\sqrt{31}}{6\sqrt{3}}-\frac{1}{2}}\left(\frac{\sqrt{3}\,i}{2}-\frac{1}{2}\right)-\frac{-\frac{\sqrt{3}\,i}{2}-1/2}{3\sqrt[3]{\frac{\sqrt{31}}{6\sqrt{3}}-1/2}},\right.$$
$$\left. x = \sqrt[3]{\frac{\sqrt{31}}{6\sqrt{3}}-\frac{1}{2}}-\frac{1}{3\sqrt[3]{\frac{\sqrt{31}}{6\sqrt{3}}-1/2}}\right\}$$

```
s1:product(x-subst(part(s,i),x),i,1,3);
```

$$\left(x-\sqrt[3]{\frac{\sqrt{31}}{6\sqrt{3}}-\frac{1}{2}}+\frac{1}{3\sqrt[3]{\frac{\sqrt{31}}{6\sqrt{3}}-1/2}}\right)$$
$$\left(x+\frac{\frac{\sqrt{3}\,i}{2}-1/2}{3\sqrt[3]{\frac{\sqrt{31}}{6\sqrt{3}}-1/2}}-\sqrt[3]{\frac{\sqrt{31}}{6\sqrt{3}}-\frac{1}{2}}\left(-\frac{\sqrt{3}\,i}{2}-\frac{1}{2}\right)\right)$$
$$\left(x-\sqrt[3]{\frac{\sqrt{31}}{6\sqrt{3}}-\frac{1}{2}}\left(\frac{\sqrt{3}\,i}{2}-\frac{1}{2}\right)+\frac{-\frac{\sqrt{3}\,i}{2}-1/2}{3\sqrt[3]{\frac{\sqrt{31}}{6\sqrt{3}}-1/2}}\right)$$

Eine Vereinfachung muß wie in Maple und MuPAD explizit angestoßen werden, wozu es die Funktion `ev` (evaluate) gibt, die mit verschiedenen Argumenten zu verschiedenem Simplifikationsverhalten veranlaßt werden kann. Hier hilft der Zusatz `radcan` weiter, der eine Vereinfachung von Radikalen anstößt:

[2] In Macsyma ist der Doppelpunkt der Zuweisungsoperator

```
ev(s1,radcan);
```

$$x^3 + x + 1$$

Auf dieselbe Weise lassen sich die Potenzsummen berechnen:

```
s2:ev(makelist(sum(subst(part(s,i),x)^k,i,1,3),k,2,9),radcan);
```

$$[-2, -3, 2, 5, 1, -7, -6, 6]$$

An diesem Beispiel wird der funktionale Charakter der im symbolischen Rechnen verwendeten Programmiersprachen sehr deutlich. Natürlich hätte man, und sei es zur besseren Lesbarkeit, auch die einzelnen Schritte zur Konstruktion der Liste der zu evaluierenden Ausdrücke nacheinander ausführen und die jeweiligen Zwischenergebnisse in Variablen speichern können. Listen sind jedoch eine sehr bequeme Datenstruktur, mit der sich auch größere (homogene) Datenmengen wie in diesem Fall die verschiedenen Potenzsummen von einer Funktion an die nächste übergeben lassen.

5.2 Die allgemeine Lösung einer Gleichungen dritten Grades

Wir haben bei der Analyse der bisherigen Ergebnisse nur darauf geachtet, ob die verschiedenen Systeme mit den produzierten Größen auch vernünftig umgehen konnten. Obwohl das Aussehen der Lösung von System zu System sehr differierte, schienen die Ergebnisse semantisch äquivalent zu sein, da die abgeleiteten Resultate, die jeweiligen Potenzsummen, zu denselben ganzen Zahlen vereinfacht werden konnten. Um zu beurteilen, wie angemessen die jeweiligen Antworten ausfielen, müssen wir deshalb zunächst Erwartungen an die Gestalt der jeweiligen Antwort formulieren.

Wir wollen deshalb als Intermezzo jetzt das Verfahren zur exakten Berechnung der Nullstellen von Gleichungen dritten Grades als geschachtelte Wurzelausdrücke kennenlernen und dann vergleichen, inwiefern die verschiedenen Systeme auch inhaltlich zufriedenstellende Antworten geben.

Aus dem Grundkurs Algebra ist bekannt, daß jedes Polynom dritten Grades $f(x) = x^3 + a\,x^2 + b\,x + c$ mit reellen Koeffizienten drei komplexe Nullstellen besitzt, von denen aus Stetigkeitsgründen wenigstens eine reell ist. Die beiden anderen Nullstellen sind entweder ebenfalls reelle Zahlen oder aber zueinander konjugierte komplexe Zahlen.

Um eine allgemeine Darstellung der Nullstellen durch Wurzelausdrücke in a, b, c zu erhalten, überführt man das Polynom f zunächst durch die *Tschirnhausen-Transformation* $x \mapsto y - \frac{a}{3}$ in die *reduzierte Form* (alle Rechnungen mit MuPAD)

```
f:=x^3+a*x^2+b*x+c:
collect(subs(f,x=y-a/3),y);
```

$$y^3 + y\left(b - \frac{a^2}{3}\right) + \left(c - \frac{ab}{3} + \frac{2a^3}{27}\right) = y^3 + p\,y + q$$

Diese Form hängt nur noch von den zwei Parametern $p = b - \frac{a^2}{3}$ und $q = c - \frac{ab}{3} + \frac{2a^3}{27}$ ab. Die Lösung der reduzierten Gleichung $g = y^3 + p\,y + q$ suchen wir in der Form $y = u + v$, wobei wir die Aufteilung in zwei Summanden so vornehmen wollen, das eine noch zu spezifizierende Nebenbedingung erfüllt ist.

```
g:=y^3+p*y+q;
expand(subs(g,y=u+v));
```

$$q + pu + pv + u^3 + v^3 + 3uv^2 + 3u^2v$$

Diesen Ausdruck formen wir um zu

$$(u + v)(3uv + p) + (u^3 + v^3 + q)$$

und wählen u und v so, daß

$$3uv + p = u^3 + v^3 + q = 0$$

gilt. Aus der ersten Beziehung erhalten wir $v = \frac{-p}{3u}$, welches, in die zweite Gleichung eingesetzt, nach kurzer Umformung eine quadratische Gleichung für u^3 ergibt:

$$\begin{aligned} u^3 + v^3 + q = u^3 - \tfrac{p^3}{27u^3} + q = 0 \\ \iff (u^3)^2 + q \cdot u^3 - \tfrac{p^3}{27} = 0 \end{aligned}$$

Die Diskriminante D dieser Gleichung ist

$$D = \left(\frac{q}{2}\right)^2 + \left(\frac{p}{3}\right)^3 \qquad \left(= -\frac{abc}{6} + \frac{b^3}{27} + \frac{c^2}{4} + \frac{a^3c}{27} - \frac{a^2b^2}{108}\right)$$

und je nach Vorzeichen von D erhalten wir für u^3 zwei reelle Lösungen $u^3 = -\frac{q}{2} \pm \sqrt{D}$ (falls $D > 0$), eine reelle Doppellösung $u^3 = -\frac{q}{2}$ (falls $D = 0$) oder zwei zueinander konjugierte komplexe Lösungen $u^3 = -\frac{q}{2} \pm i \cdot \sqrt{-D}$ (falls $D < 0$).

Betrachten wir zunächst den Fall $\mathbf{D \geq 0}$. Zur Ermittlung der reellen Lösung können wir die (eindeutige) reelle dritte Wurzel ziehen und erhalten $u_1 = \sqrt[3]{-\frac{q}{2} \pm \sqrt{D}}$ und nach geeigneter Erweiterung des Nenners $v_1 = \frac{-p}{3u} = \sqrt[3]{-\frac{q}{2} \mp \sqrt{D}}$. Beide Lösungen liefern also ein und denselben Wert

$$y_1 = \sqrt[3]{-\frac{q}{2} + \sqrt{D}} + \sqrt[3]{-\frac{q}{2} - \sqrt{D}}.$$

Diese Formel nennt man die *Cardanosche Formel* für die Gestalt der reellen Lösung einer reduzierten Gleichung dritten Grades. Die beiden komplexen Lösungen erhält man, wenn man mit den entsprechenden komplexen dritten Wurzeln für u startet. Diese unterscheiden sich von u_1 nur durch eine dritte Einheitswurzel $\omega = -\frac{1}{2} + \frac{i}{2}\sqrt{3}$ oder $\omega^2 = \bar{\omega} = -\frac{1}{2} - \frac{i}{2}\sqrt{3}$. Wegen $u_1v_1 = u_2v_2 = u_3v_3$ erhalten wir

$$\begin{aligned} u_2 = \omega u_1,\ v_2 = \bar{\omega} v_1 \\ u_3 = \bar{\omega} u_1,\ v_3 = \omega v_1 \end{aligned}$$

Zusammen ergeben sich die beiden komplexen Lösungen als

$$y_{2,3} = -\frac{u_1 + v_1}{2} \pm \frac{i}{2}\sqrt{3}(u_1 - v_1).$$

Im Falle $D = 0$ gilt $u_1 = v_1$, so daß die beiden komplexen Lösungen zu einer reellen Doppellösung zusammenfallen.

Wesentlich interessanter fällt der Fall $\mathbf{D < 0}$ aus. Man beachte zunächst, daß wegen $D = \left(\frac{q}{2}\right)^2 + \left(\frac{p}{3}\right)^3$ dann $p < 0$ gilt. Wollen wir aus obiger Gleichung für u^3 die Lösungen für u berechnen, so haben wir aus einer komplexen Zahl die dritte Wurzel zu ziehen. Kombinieren wir die Ergebnisse wie im ersten Fall, so stellt sich heraus, daß wir über den Umweg der komplexen Zahlen drei *reelle* Lösungen bekommen. Dieser Fall wird deshalb auch als *casus irreducibilis* bezeichnet, da er vor Einführung der komplexen Zahlen nicht aus den Cardanoschen Formeln hergeleitet werden konnte.

Führen wir die Rechnungen genauer aus. Zunächst überführen wir den Ausdruck für u^3 in die für das Wurzelziehen geeignetere trigonometrische Form.

$$\begin{aligned} & u^3 = -\tfrac{q}{2} \pm i \cdot \sqrt{-D} = R \cdot (\cos(\phi) \pm i \cdot \sin(\phi)) \\ & \text{mit} \qquad R := \sqrt{\left(\tfrac{q}{2}\right)^2 - D} = \sqrt{-\left(\tfrac{p}{3}\right)^3} \\ & \text{und} \qquad \phi := \arccos\left(\tfrac{-q}{2R}\right) \end{aligned}$$

Wir erhalten daraus die drei komplexen Lösungen

$$u_{k+1} = r \cdot (\cos(\tfrac{\phi+k\pi}{3}) \pm i \cdot \sin(\tfrac{\phi+k\pi}{3})),\ k = 0, 1, 2$$
$$\text{mit} \qquad r := \sqrt[3]{R} = \sqrt{-\tfrac{p}{3}}$$

Da $uv = -\frac{p}{3}$ reell ist, stellt sich ähnlich wie im ersten Fall heraus, daß u und v zueinander konjugierte komplexe Zahlen sind, womit sich bei der Berechnung von y die Imaginärteile der beiden Summanden wegheben. Wir erhalten schließlich die *trigonometrische Form* der drei reellen Nullstellen von g

$$y_{k+1} = 2r \cdot \cos(\tfrac{\phi+k\pi}{3}),\ k = 0, 1, 2.$$

Daß es sich bei den drei reellen Zahlen y_1, y_2, y_3 wirklich um Nullstellen von g handelt, kann man allerdings auch ohne den Umweg über komplexe Zahlen sehen: Für $y = 2\sqrt{-\frac{p}{3}}\cos(\alpha)$ gilt wegen der Produkt-Summe-Regel für trigonometrische Funktionen

$$\begin{aligned} g(y) &= \tfrac{-8p}{3}\sqrt{-\tfrac{p}{3}}\cos^3(\alpha) + 2p\sqrt{-\tfrac{p}{3}}\cos(\alpha) + q \\ &= \tfrac{-8p}{3}\sqrt{-\tfrac{p}{3}}\left(\tfrac{\cos(3\alpha)+3\cos(\alpha)}{4}\right) + 2p\sqrt{-\tfrac{p}{3}}\cos(\alpha) + q \\ &= \tfrac{-2p}{3}\sqrt{-\tfrac{p}{3}}\cos(3\alpha) + q = 2R\cos(3\alpha) + q. \end{aligned}$$

Damit verschwindet $g(y_k)$ für $k = 0, 1, 2$, da ja gerade $\phi = \arccos\left(\frac{-q}{2R}\right)$ war.

Fassen wir unsere Ausführungen in dem folgenden Satz zusammen.

Satz 9 *Sei $g := y^3 + py + q$ eine reduziertes Polynom dritten Grades und $D = \left(\frac{q}{2}\right)^2 + \left(\frac{p}{3}\right)^3$ dessen Diskriminante. Dann gilt*

1. *Ist $D > 0$, so hat g eine reelle und zwei komplexe Nullstellen. Diese ergeben sich mit*

   $$u_1 = \sqrt[3]{-\frac{q}{2} + \sqrt{D}}, \qquad v_1 = \sqrt[3]{-\frac{q}{2} - \sqrt{D}}$$

   aus der Cardanoschen Formel

   $$y_1 = u_1 + v_1$$

   bzw. als

   $$y_{2,3} = -\frac{u_1 + v_1}{2} \pm \frac{i}{2}\sqrt{3}\,(u_1 - v_1).$$

2. *Ist $D = 0$, so hat g eine einfache reelle Nullstelle $y_1 = 2\sqrt[3]{-\frac{q}{2}}$ und eine reelle Doppelnullstelle $y_{2,3} = -\sqrt[3]{-\frac{q}{2}}$.*

3. *Ist $D < 0$, so hat g mit $\phi = \arccos\left(\frac{-q}{2R}\right)$ drei reelle Nullstellen*

   $$y_{k+1} = 2\sqrt{-\frac{p}{3}} \cdot \cos\left(\frac{\phi + k\pi}{3}\right),\ k = 0, 1, 2.$$

5.3 * Die allgemeine Lösung einer Gleichungen vierten Grades

Eine Darstellung durch Wurzelausdrücke existiert auch für die Nullstellen von Polynomen vierten Grades. Die nachstehenden Ausführungen folgen [14, Kap. 16].

Mit der *Tschirnhausen-Transformation* $x \mapsto y - \frac{a}{4}$ kann man das Polynom vierten Grades $f = x^4 + ax^3 + bx^2 + cx + d$ zunächst wieder in die *reduzierte Form* $g = y^4 + py^2 + qy + r$ ohne kubischen Term überführen. Sind nun (u, v, w) drei komplexe Zahlen, die die Bedingungen

$$\begin{aligned} uvw &= -\tfrac{q}{8} \\ u^2 + v^2 + w^2 &= -\tfrac{p}{2} \\ u^2v^2 + u^2w^2 + v^2w^2 &= \tfrac{p^2-4r}{16} \end{aligned}$$

erfüllen, so ist $y = u + v + w$ eine Nullstelle von g. In der Tat, ersetzen wir in g die Variable y durch die angegebene Summe und gruppieren die Terme entsprechend, so erhalten wir den Ausdruck

```
g:=y^4+p*y^2+q*y+r:
expand(subs(g,y=u+v+w));
```

$$\begin{aligned} & r + qu + qv + qw + 2puv + 2puw + 2pvw + u^4 + v^4 + w^4 + pu^2 + pv^2 + pw^2 + 4uv^3 + 4u^3v + \\ & 4uw^3 + 4u^3w + 4vw^3 + 4v^3w + 12uvw^2 + 12uv^2w + 12u^2vw + 6u^2v^2 + 6u^2w^2 + 6v^2w^2 \\ & = (u + v + w)(8uvw + q) + (uv + uw + vw)(4(u^2 + v^2 + w^2) + 2p) + (u^2 + v^2 + w^2)^2 + \\ & 4(u^2v^2 + u^2w^2 + v^2w^2) + p(u^2 + v^2 + w^2) + r, \end{aligned}$$

der für alle Tripel (u, v, w), die die angegebenen Bedingungen erfüllen, verschwindet. Für ein solches Tripel sind aber nach dem Vietaschen Wurzelsatz (u^2, v^2, w^2) die drei Nullstellen der kubischen Gleichung

$$z^3 + \frac{p}{2}z^2 + \frac{p^2 - 4r}{16}z - \frac{q^2}{64} = 0$$

Sind umgekehrt z_1, z_2, z_3 Lösungen dieser kubischen Gleichung, so erfüllt jedes Tripel (u, v, w) mit $u = \pm\sqrt{z_1}, v = \pm\sqrt{z_2}, w = \pm\sqrt{z_3}$ nach dem Vietaschen Wurzelsatz die Gleichungen

$$\begin{aligned} uvw &= \pm\tfrac{q}{8} \\ u^2 + v^2 + w^2 &= -\tfrac{p}{2} \\ u^2v^2 + u^2w^2 + v^2w^2 &= \tfrac{p^2-4r}{16}, \end{aligned}$$

wobei vier der Vorzeichenkombinationen in der ersten Gleichung das Vorzeichen + und die restlichen das Vorzeichen − ergeben. Ist (u, v, w) ein Tripel, das als Vorzeichen in der ersten Gleichung − ergibt, so auch die Tripel $(u, -v, -w)$, $(-u, v, -w)$ und $(-u, -v, w)$. Damit sind

$$y_1 = u + v + w,\ y_2 = u - v - w,\ y_3 = -u + v - w,\ y_4 = -u - v + w$$

die vier Nullstellen des Polynoms g vierten Grades. Die allgemeine Lösung, die man ohne Mühe aus der entsprechenden allgemeinen Lösung der zugehörigen Gleichung zusammenstellen kann, ist allerdings bereits wesentlich umfangreicher, so daß es noch schwieriger als bei Gleichungen dritten Grades ist, mit den so generierten Wurzelausdrücken weiterzurechnen. Für eine Klassifizierung an Hand der Parameter p, q, r nach der Anzahl reeller Nullstellen sei etwa auf [14] verwiesen.

Nachdem die Lösungsverfahren für Gleichungen dritten und vierten Grades wenigstens seit dem 16. Jahrhundert bekannt waren, versuchten die Mathematiker lange Zeit, auch für Gleichungen höheren Grades solche allgemeinen Formeln für die entsprechenden Nullstellen zu finden. Erst in den Jahren 1824 und 1826 gelang es N.H. ABEL den Beweis zu erbringen, daß es solche allgemeinen Formeln nicht geben kann. Heute gibt die Theorie der Galoisgruppen, die mit jeder solchen Gleichung eine entsprechende Permutationsgruppe verbindet, Antwort, ob und wie sich die Nullstellen eines bestimmten Polynoms fünften oder höheren Grades durch Radikale ausdrücken lassen.

5.4 Die RootOf-Notation

Doch kehren wir zu den Polynomen dritten Grades zurück und untersuchen, wie die einzelnen CAS die verschiedenen Fälle der allgemeinen Lösungsformel behandeln.

Es stellt sich heraus, daß von den betrachteten CAS nur Derive und MuPAD die beiden Fälle $D > 0$ und $D < 0$ bei der Gestaltung der Ausgabe unterscheiden, während sonst (außer Reduce) die Cardanosche Formel auch für komplexe Radikanden angesetzt wird. So antwortet MuPAD im *casus irreducibilis* wie erwartet

```
solve(x^3-3*x+1,x);
```

$$\left\{2\cos\left(\tfrac{2\pi}{9}\right), -\cos\left(\tfrac{2\pi}{9}\right) + \sqrt{3}\sin\left(\tfrac{2\pi}{9}\right), -\cos\left(\tfrac{2\pi}{9}\right) - \sqrt{3}\sin\left(\tfrac{2\pi}{9}\right)\right\}$$

Maples Antwort dagegen lautet

```
s:=[solve(x^3-3*x+1,x)];
```

$$\left[1/2\ \sqrt[3]{-4+4\sqrt{-3}} + 2\ \frac{1}{\sqrt[3]{-4+4\sqrt{-3}}},\right.$$
$$-1/4\ \sqrt[3]{-4+4\sqrt{-3}} - \frac{1}{\sqrt[3]{-4+4\sqrt{-3}}} + 1/2\sqrt{-3}\left(1/2\ \sqrt[3]{-4+4\sqrt{-3}} - 2\ \frac{1}{\sqrt[3]{-4+4\sqrt{-3}}}\right),$$
$$\left.-1/4\ \sqrt[3]{-4+4\sqrt{-3}} - \frac{1}{\sqrt[3]{-4+4\sqrt{-3}}} - 1/2\sqrt{-3}\left(1/2\ \sqrt[3]{-4+4\sqrt{-3}} - 2\ \frac{1}{\sqrt[3]{-4+4\sqrt{-3}}}\right)\right]$$

Letzteres hat den Nachteil, daß man der Lösung selbst nach einer näherungsweisen Berechnung nicht ansieht, daß es sich um reelle Zahlen handelt:

```
evalf(s);
                                  -9
        [1.532088886 - .1 10   I, -1.879385242, .3472963560]
```

Reduce erlaubt, mit dem Schalter `trigform` zwischen beiden Darstellungsformen zu wechseln, wobei standardmäßig die trigonometrische Form eingestellt ist, was zwar für $D < 0$ zu dem gewünschten Ergebnis führt

```
on fullroots;
s:=solve(x^3-3*x+1,x);
```

$$\{x = 2\cdot\cos(\tfrac{2\cdot\pi}{9}), x = -\cos(\tfrac{2\cdot\pi}{9}) + \sqrt{3}\cdot\sin(\tfrac{2\cdot\pi}{9}), x = -\cos(\tfrac{2\cdot\pi}{9}) - \sqrt{3}\cdot\sin(\tfrac{2\cdot\pi}{9})\},$$

aber im Fall $D > 0$ das unverständliche Ergebnis

```
s:=solve(x^3+3*x+1,x);
```

$$\{x = 2\cdot\cosh(\tfrac{\operatorname{arctanh}(\sqrt{5})}{3})\cdot i, x = -\cosh(\tfrac{\operatorname{arctanh}(\sqrt{5})}{3})\cdot i + \sqrt{3}\cdot\sinh(\tfrac{\operatorname{arctanh}(\sqrt{5})}{3}),$$
$$x = -\cosh(\tfrac{\operatorname{arctanh}(\sqrt{5})}{3})\cdot i - \sqrt{3}\cdot\sinh(\tfrac{\operatorname{arctanh}(\sqrt{5})}{3})\}$$

liefert.

Die explizite Verwendung der Lösungsformel für kubische Gleichungen birgt allerdings auch Schwierigkeiten für die weitere Vereinfachung der dabei entstehenden Ausdrücke. Vergessen der Entstehungsgeschichte von Teilausdrücken führt zu dem bei Derive beobachteten Verhalten. Erzeugt man wie in MuPAD die beiden dritten Wurzeln u, v in der Cardanoschen Formel getrennt, so bekommt man zwar im Gegensatz zu den anderen Systemen etwas angenehmere Wurzelausdrücke

(insbesondere solche mit rationalen Nennern), hat aber im Nachhinein Probleme mit Ausdrücken, die Produkte dieser Wurzeln enthalten, obwohl wir wissen, daß $u\,v = -\frac{p}{3}$ gilt, d.h. die beiden Wurzelausdrücke im wesentlichen zueinander invers sind.

Noch schwieriger wird die Entscheidung, welche Art der Darstellung der Lösung gewählt werden soll, wenn die zu betrachtende Gleichung Parameter enthält. Betrachten wir etwa die Aufgabe

```
s:=solve(x^3+a*x+1,x);
```

die von einem Parameter a abhängt. Die Lösungsdarstellung sollte mit möglichen späteren Substitutionen konsistent sein, d.h. bei einer Ersetzung

```
subs(a=3,s);
```

zur Cardanoschen Formel und bei

```
subs(a=-3,s);
```

zur trigonometrischen Darstellung verzweigen. Um dies zu erreichen, müßte man aber die Entscheidung bis zur Substitution aufschieben, d.h. `s` müßte eine Liste sein, die drei neue Symbole enthält, von denen nur bekannt ist, daß sie Lösung einer bestimmten Gleichung drittes Grades sind.

Genau ein solches Verhalten zeigt Reduce in der Standardeinstellung, d.h. ohne den Schalter `fullroots` einzuschalten:

```
solve(x^3+x+1,x);
```

liefert die auf den ersten Blick verblüffende Antwort

```
                3
{x=root_of(x_  + x_ + 1,x_,tag_1)}
```

Es wird ein Funktionsausdruck mit dem Funktionssymbol $x\_ = \text{RootOf}(\ldots)$ und dem zugehörigen Polynom als Parameter erzeugt, das stellvertretend für die einzelnen Nullstellen dieses Polynoms steht und von dem nichts weiter bekannt ist als die Gültigkeit der Ersetzungsregel $x\_^3 \Rightarrow -(x\_+1)$.

Neben einer höheren Konsistenz der Darstellung spricht für ein solches Vorgehen auch die Tatsache, daß die Wurzeln einer Gleichung dritten Grades schon ein recht kompliziertes Aussehen haben können, aus dem sich die genannte Ersetzungsinformation nur noch schwer rekonstruieren läßt. Dies gilt erst recht für Nullstellen einer Gleichung vierten Grades. Schließlich gibt es für Nullstellen allgemeiner Gleichungen höheren Grades gar keine Darstellung in Radikalen, so daß für solche Zahlen überhaupt nur auf eine RootOf-Darstellung zurückgegriffen werden kann.

Neben diesen praktischen Erwägungen ist eine solche Darstellung auch aus theoretischen Gründen günstig, denn es gilt der folgende

Satz 10 *Sei R ein Körper mit kanonischer Form und a eine Nullstelle des über R irreduziblen Polynoms $p(x) = x^k - r(x) \in R[x]$, wobei deg $r < k$ sei. Eine solche Nullstelle nennt man eine* algebraische Zahl.
Stellt man polynomiale Ausdrücke in $R[a]$ in distributiver Form dar und wendet zusätzlich die algebraische Regel `{ a^k => r(a) }` *an, so erhält man eine kanonische Form.*

Beweis : Mit dem beschriebenen Verfahren kann man jeden Ausdruck $U \in R[a]$ in der Form $U = \sum_{i=0}^{k-1} r_i a^i$ darstellen. Es bleibt zu zeigen, daß diese Darstellung eindeutig ist.
Wie früher auch genügt es zu zeigen, daß aus $0 = \sum_{i=0}^{k-1} r_i a^i$ bereits $r_i = 0$ für alle $i < k$ folgt. Wäre das nicht so, so wäre a eine Nullstelle des Polynoms $q(x) := \sum_{i=0}^{k-1} r_i x^i$ vom Grad

$deg\ q < k$. Berechnen wir etwa mit dem Euklidschen Algorithmus $g(x) = gcd(p(x), q(x))$ und eine entsprechende Darstellung

$$g(x) = u(x)\,p(x) + v(x)\,q(x)$$

des größten gemeinsamen Teilers als Linearkombination von p und q, so sehen wir, daß a auch Nullstelle von $g(x)$ ist. $g(x)$ ist also ein nichttrivialer Teiler von $p(x)$, das aber als irreduzibel vorausgesetzt wurde. □

Es stellt sich heraus, daß $R[a]$ nicht nur ein Ring, sondern seinerseits wieder ein Körper ist, wie wir weiter unten noch genauer untersuchen werden. Wendet man das beschriebene Verfahren rekursiv an, so erhält man eine für Rechnungen sehr brauchbare Darstellung für solche algebraischen Zahlen. Zur Einführung einer neuen algebraischen Zahl a muß man dazu jeweils "nur" ein über dem bisherigen Bereich irreduzibles Polynom finden, dessen Nullstelle a ist, d.h. im wesentlichen in solchen Bereichen faktorisieren können.

Die Verwendung eines RootOf-Symbols ist deshalb keine Besonderheit von Reduce, sondern wird inzwischen auch von anderen Systemen verwendet. Der `Solve`-Operator von **Maple** und **MuPAD** unterscheidet zwischen Nullstellen einzelner Gleichungen und Nullstellen von Gleichungssystemen. Für erstere wird angenommen, daß der Nutzer nach einer möglichst expliziten Lösung sucht und die RootOf-Notation erst ab Grad 4 (Maple) bzw. Grad 5 (MuPAD) eingesetzt. Für zweiteren Fall werden in Maple standardmäßig alle nichtrationalen Nullstellen durch RootOf-Ausdrücke dargestellt, in MuPAD dagegen werden auch Quadratwurzel-Ausdrücke erzeugt. Dieses Verhalten kann man durch Setzen eines entsprechenden Parameters (`_EnvExplicit` in Maple, `MaxDegree` in MuPAD) ändern. Ähnlich gehen auch **Reduce** (RootOf ab Grad 3, wie wir in obigem Beispiel gesehen hatten) und **Mathematica** (RootOf ab Grad 5, was man aber durch die Optionen `Cubics -> False` und `Quartics -> False` abstellen kann) vor. Mit **Derive** kann man keine nichtlinearen Gleichungssysteme lösen. **Macsyma** (Version 420) löste die Probleme mit algebraischen Nullstellen auf seine Weise: Wenn die Nullstellen nicht mehr durch entsprechende Radikale dargestellt werden konnten, so wurde zu näherungsweisen Lösungen übergegangen. Dieses Verhalten hat sich in neueren Versionen durch die Bereitstellung zweier neuer Pakete (`algsys` und `triangsys`) bereits geändert, so daß auch Macsyma mittlerweile eine RootOf-Notation unterstützt.

Eine Besonderheit des RootOf-Symbols im Vergleich zu anderen Funktionssymbolen ist der Umstand, daß es in den meisten Zusammenhängen eigentlich für eine zusammengesetzte Datenstruktur, etwa die Liste (oder gar Substitutionsliste) aller Nullstellen steht. Die derzeit in den verschiedenen Systemen verwendete Notation ist in dieser Hinsicht unzureichend. So ist etwa die Antwort von Reduce auf

```
solve(x^3+x+1,x);

              3
{x=root_of(x_  + x_ + 1,x_,tag_1)}
```

unverständlich, denn die Gleichung hat ja nicht eine, sondern drei Lösungen. Auch die Antwort von Maple auf die Aufgabe

```
s:=[solve({x^2+y+z-3,y^2+x+z-3,z^2+x+y-3},{x,y,z})];
```

$$\begin{aligned}[{} & \{z = 2\,RootOf(\_Z^2+1), y = 2\,RootOf(\_Z^2+1), x = 1 - 2\,RootOf(\_Z^2+1)\},\\ & \{z = RootOf(2\,\_Z + \_Z^2 + 3), y = RootOf(2\,\_Z + \_Z^2 + 3),\\ & \qquad\qquad x = RootOf(2\,\_Z + \_Z^2 + 3)\},\\ & \{z = 2\,RootOf(\_Z^2+1), y = 1 - 2\,RootOf(\_Z^2+1), x = 2\,RootOf(\_Z^2+1)\},\\ & \{z = RootOf(5 - 2\,\_Z + \_Z^2), y = 1 - RootOf(5 - 2\,\_Z + \_Z^2),\\ & \qquad\qquad x = 1 - RootOf(5 - 2\,\_Z + \_Z^2)\}]\end{aligned}$$

ist mathematisch wenig zufriedenstellend, denn *RootOf* steht ja für *mehrere* Wurzeln, die nacheinander und teilweise nicht einmal unabhängig voneinander an den entsprechenden Stellen einzusetzen sind.

Zwar besitzen sowohl Reduce[3] als auch Maple[4] entsprechende Funktionen, um diese Mehrdeutigkeiten ggf. aufzulösen, allerdings setzt deren Anwendung eine gehörige Portion Wissen über interne Abläufe voraus.

Die mit dem mathematischen Gebrauch am besten übereinstimmenden Lösungen bieten derzeit Mathematica (ab Version 3.0) und Axiom an. **Mathematica** erzeugt für jede Nullstelle einen eigenen Funktionsausdruck, die in einer Liste in mit der sonstigen Notation konsistenter Form aufgesammelt werden:

```
Solve[x^5+a*x+1==0,x]

  {{x -> Root[1 + a*#1 + #1^5 & , 1]}, {x -> Root[1 + a*#1 + #1^5 & , 2]},
   {x -> Root[1 + a*#1 + #1^5 & , 3]}, {x -> Root[1 + a*#1 + #1^5 & , 4]},
   {x -> Root[1 + a*#1 + #1^5 & , 5]}}
```

Es werden für die einzelnen (vom Parameter a abhängenden) Nullstellen fünf verschiedene Funktionsausdrücke mit jeweils zwei Argumenten gebildet. Eines davon ist das Polynom, das die Nullstelle beschreibt, und zwar in einer Notation als reine Funktion (die eine Beschreibung, aber keinen Namen hat; `#1` steht hierbei für einen formalen Parameter), das zweite ein Index[5]. Ersetzt man in dieser Liste $a = 1$, so lassen sich die fünf Nullstellen in die zwei Nullstellen eines Polynoms zweiten und die drei Nullstellen eines Polynoms dritten Grades zerlegen. Diese Umformungen ergeben sich hier auf natürliche Weise:

```
s/.a->1

  {{x -> (-1 - I*Sqrt[3])/2}, {x -> (-1 + I*Sqrt[3])/2},
   {x -> Root[1 - #1^2 + #1^3 & , 1]}, {x -> Root[1 - #1^2 + #1^3 & , 2]},
   {x -> Root[1 - #1^2 + #1^3 & , 3]}}
```

Will man dasselbe in Maple erreichen

```
s:=[solve(x^5+a*x+1,x)];
                                   5
                        s := [RootOf(_Z  + a _Z + 1)]

s1:=subs(a=1,s);
                                    5
                        s1 := [RootOf(_Z  + _Z + 1)]
```

so bedarf es eines auf die spezielle Datenstruktur zugeschnittenen Aufrufs der Funktion `allvalues`, die in der Lage ist, ein einzelnes RootOf-Symbol in eine Ausdruckssequenz zu expandieren, d.h. aus *einem* RootOf-Eintrag in einer *Liste* mehrere sich dahinter verbergende Nullstellen zu erzeugen. Mit `map` können wir den Aufruf dieser Funktion an die richtigen Stellen unserer Liste s plazieren, was aber schon einige Kenntnisse der Interna von Maple voraussetzt.

```
map(allvalues,s1);
```

[3] Mit `on fullroots;` wird der obige Ausdruck in eine Liste von drei Nullstellen verwandelt, die durch das Funktionssymbol `one_of` geklammert sind. Dieses kann man durch die Funktion `expand_cases` expandieren.

[4] Hier hilft `map(allvalues,s)` weiter, das auf jeden einzelnen Eintrag in s die Funktion `allvalues` anwendet. Diese Funktion versucht, Nullstellen bis zum Grad 4 durch entsprechende Radikale zu ersetzen und sonst wie in Macsyma mit Näherungslösungen weiterzuhelfen.

[5] Die Nullstellen werden in konkreten Anwendungen an Hand von Näherungswerten unterschieden.

$$[-1/2+1/2\,\sqrt{-1}\sqrt{3},-1/2-1/2\,\sqrt{-1}\sqrt{3},-1/6\,\%1-2/3\,\frac{1}{\%1}+1/3,$$
$$1/12\,\%1+1/3\,\frac{1}{\%1}+1/3+1/2\,\sqrt{-1}\sqrt{3}\left(-1/6\,\%1+2/3\,\frac{1}{\%1}\right),$$
$$1/12\,\%1+1/3\,\frac{1}{\%1}+1/3-1/2\,\sqrt{-1}\sqrt{3}\left(-1/6\,\%1+2/3\,\frac{1}{\%1}\right)]$$

mit

$$\%1=\sqrt[3]{100+12\,\sqrt{69}}$$

In Reduce erfolgt eine solche Konversion automatisch

```
s:=solve(x^5+a*x+1,x);
```

$$\{x=\mathrm{root\_of}(a\cdot x\_+x\_^5+1,x\_)\}$$

```
sub(a=1,s);
```

$$\{x=\mathrm{one\_of}(\{\frac{\sqrt{3}\cdot i-1}{2},x=\frac{-(\sqrt{3}\cdot i+1)}{2},x=\mathrm{root\_of}(x\_^3-x\_^2+1,x\_)\})\}$$

Die auf diese Weise durch Expandieren aus einem einzelnen RootOf-Symbol entstandene Liste von Lösungen wird allerdings noch durch ein (weiteres) Funktionssymbol `one_of` zusammengehalten, das im weiteren vom Nutzer mit Hilfe der Funktion `expand_cases` aufgelöst werden kann.

Axioms Ansatz besteht schließlich darin, überhaupt kein RootOf-Symbol einzuführen, sondern die Lösung nur soweit aufzuspalten, wie dies im gegebenen Kontext möglich ist.

```
s:=solve(x^5+a*x+1,x)

  [x^5  + a x + 1 = 0]

       Type: List Equation Fraction Polynomial Integer

s1:=solve(x^5+x+1,x)

  [x^2  + x + 1 = 0, x^3  - x^2  + 1 = 0]

        Type: List Equation Fraction Polynomial Integer
```

Die weitere Bearbeitung erfolgt durch Aufruf entsprechender Funktionen auf zu selektierenden Teilausdrücken. So liefert etwa

```
radicalSolve(s1.1)
```

$$\left[x=\frac{-\sqrt{-3}-1}{2},\ x=\frac{\sqrt{-3}-1}{2}\right]$$

die beiden Nullstellen des ersten Ausdrucks. Das strenge Typkonzept verlangt aber auch hier ein gehöriges Wissen des Nutzers über die Interna des CAS, da die Daten nicht wie in den anderen Systemen alle auf gleiche Weise *Ausdrucksbäume* sind, sondern einen *strengen Typ* besitzen, der beim Aufruf einer Funktion mit deren Signatur verglichen wird. So führt etwa der Aufruf

```
subst(s,a=1)
```

zu einem Fehler

```
   Cannot find a definition or applicable library operation named subst
      with argument type(s)
                  List Equation Fraction Polynomial Integer
                         Equation Polynomial Integer
```

während eine elementweise Substitution

```
[subst(x,a=1) for x in s]
```

```
          5
   (3)  [x  + x + 1= 0]
                           Type: List Equation Expression Integer
```

das gewünschte Ergebnis liefert.

Wir sehen an diesem Beispiel, daß ein strenges Typsystem beim Arbeiten mit symbolischen Ausdrücken, die oftmals eine recht verzwickte Datenstruktur bilden, auch seine Nachteile hat.

5.5 Das Rechnen mit algebraischen Zahlen

Wir wollen nun einige aus einem Kurs zur höheren Algebra wohlbekannte Eigenschaften algebraischer Zahlen zusammentragen und zu deren Beweis und Veranschaulichung Mittel der Computeralgebra einsetzen. Da hierbei die Demonstration eines CAS als Arbeitsinstrument im Mittelpunkt steht, werden wir uns in den folgenden Ausführungen auf Maple und Reduce beschränken. Jedes andere der Systeme wäre aber sicher ähnlich gut geeignet, die erforderlichen symbolischen Transformationen zu vollziehen.

5.5.1 Definitionen und Beispiele

Wir hatten bereits weiter oben den Begriff der algebraischen Zahl erwähnt und wollen an dieser Stelle eine formale Definition nachholen.

DEFINITION: Seien $k \subset K$ zwei Körper. $\alpha \in K$ bezeichnet man als *algebraisch über* k, wenn es ein Polynom $p(x) \in k[x]$ mit $p(\alpha) = 0$ gibt.

Satz 11 *Sei $\alpha \in K$ eine algebraische Zahl und*

$$P := \{q(x) \in k[x] \,:\, q(\alpha) = 0 \text{ und } lc(q) = 1\}$$

In P gibt es genau ein Polynom $p(x) \in k[x]$ minimalen Grades. $p(x)$ ist irreduzibel in $k[x]$ und alle Polynome $q(x) \in P$ enthalten $p(x)$ als Faktor.

Dieses Polynom heißt charakteristisches Polynom *oder* Minimalpolynom, *sein Grad der* Grad *der algebraischen Zahl α.*

Der Beweis dieses Satzes ergibt sich unmittelbar aus dem Satz über Division mit Rest in $k[x]$.

Beispiele (über $k = \mathbf{Q}$):

$$\alpha_1 = \sqrt{2} \qquad p_1(x) = x^2 - 2$$
$$\alpha_2 = \sqrt[3]{5} \qquad p_2(x) = x^3 - 5$$
$$\alpha_3 = \sqrt{1 - \sqrt{2}} \qquad p_3(x) = x^4 - 2\,x^2 - 1$$

Die Irreduzibilität von p_3 ist nicht ganz offensichtlich, kann aber leicht mit Maple getestet werden:

```
factor(x^4-2*x^2-1);
```

$$x^4 - 2\,x^2 - 1$$

Analog erhalten wir für $\alpha_4 = \sqrt{9 + 4\sqrt{5}}$ ein Polynom $p_4(x) = x^4 - 18 * x^2 + 1$, dessen Nullstelle α_4 ist. Allerdings ist p_4 nicht irreduzibel

```
factor(x^4-18*x^2+1);
```

$$(\,x^2 - 4\,x - 1\,)\,(\,x^2 + 4\,x - 1\,)$$

und α_4 als Nullstelle des ersten Faktors in Wirklichkeit eine algebraische Zahl vom Grad 2. Das weiß Maple auch:

```
sqrt(9+4*sqrt(5));
```

$$\sqrt{5} + 2$$

Andere Beispiele algebraischer Zahlen hängen mit Winkelfunktionen spezieller Argumente zusammen. Aus der Schule bekannt sind die Werte von $\sin(x)$ und $\cos(x)$ für $x = \frac{\pi}{n}$ mit $n = 3, 4, 6$. Maple kennt jedoch auch für $n = 5$ interessante Ausdrücke:

```
cos(Pi/5); cos(2*Pi/5);
```

$$\frac{1}{4}\,\sqrt{5} + \frac{1}{4}, \frac{1}{4}\,\sqrt{5} - \frac{1}{4}$$

Zur Bestimmung des entsprechenden charakteristischen Polynoms benutzen wir $\cos(5 \cdot \frac{\pi}{5}) = \cos(\pi) = -1$. Wenden wir unsere Mehrfachwinkelformeln auf $\cos(5\,x) + 1$ an, so erhalten wir ein Polynom in $\cos(x)$, das für $x = \frac{\pi}{5}$ verschwindet.

```
expand(cos(5*x)+1);
```

$$16\cos(\,x\,)^5 - 20\cos(\,x\,)^3 + 5\cos(\,x\,) + 1$$

```
subs(cos(x)=z,");
```

$$16\,z^5 - 20\,z^3 + 5\,z + 1$$

Diese Polynom ist allerdings noch nicht das Minimalpolynom.

```
factor(");
```

$$(\,z + 1\,)\,(\,-1 - 2\,z + 4\,z^2\,)^2$$

Dasselbe Programm kann man für $\sin(\frac{\pi}{5})$ absolvieren.

```
sin(Pi/5);
```

$$\frac{1}{4}\,\sqrt{2}\,\sqrt{5 - \sqrt{5}}$$

Zur Bestimmung der charakteristischen Gleichung gehen wir wie oben vor:

```
expand(sin(5*x));
```

$$16\,\sin(x)\cos(x)^4 - 12\,\sin(x)\cos(x)^2 + \sin(x)$$

```
simplify(",{cos(x)^2=1-sin(x)^2});
```

$$5\,\sin(x) - 20\,\sin(x)^3 + 16\,\sin(x)^5$$

```
subs(sin(x)=z,");
```

$$5\,z - 20\,z^3 + 16\,z^5$$

```
factor(");
```

$$z\left(5 - 20\,z^2 + 16\,z^4\right)$$

Also ist der zweite Faktor $p(z) := 5 - 20\,z^2 + 16\,z^4$ das irreduzible Polynom von $\sin(\frac{\pi}{5})$.

Eine algebraische Zahl ist jedoch durch ihr charakteristisches Polynom nicht eindeutig bestimmt, da ja jedes (irreduzible) Polynom vom Grad n genau n verschiedene komplexe Nullstellen hat. Es ergibt sich also z.B. die Frage, welche weiteren drei Nullstellen $p(z)$ hat und welche interssanten Beziehungen zwischen diesen vier algebraischen Zahlen bestehen. Da man für $x = \frac{k\,\pi}{5}$, $k \in \mathbf{Z}$ ebenfalls $\sin(5\,x) = 0$ hat, sind die Nullstellen des biquadratischen Polynoms neben $\alpha_{1,2} = \pm\sin(\frac{\pi}{5})$ gerade $\alpha_{3,4} = \pm\sin(\frac{2\pi}{5})$. Interessanterweise ergibt sich für das Produkt der beiden algebraischen Zahlen vom Grad 4

```
sin(Pi/5)*sin(2*Pi/5);
```

$$\frac{1}{8}\sqrt{5-\sqrt{5}}\sqrt{5+\sqrt{5}}$$

```
combine(");
```

$$\frac{1}{4}\sqrt{5}$$

eine algebraische Zahl vom Grad 2.

Untersuchen wir nun den Fall $n = 7$ und betrachten zunächst das Produkt der drei betragsmäsig verschiedenen Werte von $\cos(\frac{k\,\pi}{7}), k \in \mathbf{Z}$ (alle anderen Werte ergeben sich daraus über Quadrantenbeziehungen):

```
cos(Pi/7)*cos(2*Pi/7)*cos(3*Pi/7);
```

$$\cos(\frac{\pi}{7})\cos(\frac{4\,\pi}{7})\cos(\frac{5\,\pi}{7})$$

Hiermit kann Maple also nichts mehr anfangen. Gleichwohl erhalten wir näherungsweise

```
evalf(",20);
```

$$.12499999999999999999$$

und analog für

```
cos(Pi/7)-cos(2*Pi/7)+cos(3*Pi/7);
```

$$\cos(\frac{3\,\pi}{7}) - \cos(\frac{2\,\pi}{7}) + \cos(\frac{\pi}{7})$$

```
evalf(",20);
```

$$.50000000000000000000$$

Bestimmen wir nun nach dem bereits bekannten Schema das charakteristische Polynom von $\cos(\frac{\pi}{7})$.

```
factor(subs(cos(x)=z,expand(cos(7*x)+1)));
```

$$(z+1)\,(1-4\,z-4\,z^2+8\,z^3)^2$$

$\cos(\frac{\pi}{7})$ ist damit als Nullstelle des Polynoms $p(x) = 1 - 4\,z - 4\,z^2 + 8\,z^3$ eine algebraische Zahl vom Grad 3. Die anderen beiden Nullstellen dieses Polynoms sind $\cos(\frac{3\pi}{7})$ und $\cos(\frac{5\pi}{7}) = -\cos(\frac{2\pi}{7})$, da wir nur $\cos(7\,x) + 1 = 0$ ausgenutzt haben. Obige Beziehungen ergeben sich nun unmittelbar aus dem bereits aus Kapitel 2 bekannten *Vietaschen Wurzelsatz*

Satz 12 *Sind x_1, x_2, x_3 die drei Nullstellen des Polynoms $p(x) = x^3 - a\,x^2 + b\,x - c$, so gilt*

$$\begin{aligned} a &= x_1 + x_2 + x_3 \\ b &= x_1\,x_2 + x_2\,x_3 + x_1\,x_3 \\ c &= x_1\,x_2\,x_3 \end{aligned}$$

5.5.2 Der Ring der algebraischen Zahlen

Für die beiden algebraischen Zahlen $\sqrt{2}$ und $\sqrt{3}$ ist es nicht schwer, durch geeignete Umformungen ein Polynom zu finden, das deren Summe $a := \sqrt{2} + \sqrt{3}$ als Nullstelle hat:

$$a^2 = 5 + 2\sqrt{6} \;\Rightarrow\; (a^2-5)^2 = 24 \;\Rightarrow\; a^4 - 10\,a^2 + 1 = 0$$

a ist also Nullstelle des (in diesem Fall bereits irreduziblen) Polynoms $p(x) = x^4 - 10x^2 + 1$. Es stellt sich heraus, daß dies auch allgemein gilt:

Satz 13 *Die Summe und das Produkt zweier algebraischer Zahlen ist wieder eine algebraische Zahl.*

Vor dem allgemeinen Beweis des Satzes wollen wir die Aussage an einem etwas komplizierteren Beispiel studieren, in dem die auszuführenden Umformungen nicht so offensichtlich sind.

Betrachten wir dazu die beiden Zahlen $a = \sqrt{2}$ und $b = \sqrt[3]{5}$ und versuchen, ein Polynom $p := \sum_{i=0}^{n} r_i\,x^i$ zu konstruieren, das $c = a + b$ als Nullstelle hat, d.h. so daß $\sum_{i=0}^{n} r_i\,c^i = 0$ gilt. Um geeignete Koeffizienten r_i zu finden, berechnen wir zunächst die Potenzen $c^i = (a+b)^i$ als Ausdrücke in a und b. Dazu sind die binomischen Formeln sowie die Ersetzungen `{a^2=>2, b^3=>5}` anzuwenden, die sich aus den charakteristischen Polynomen von a bzw. b unmittelbar ergeben. Eine Rechnung mit Reduce ergibt

```
for n:=0:10 collect (a+b)^n where {a^2=>2, b^3=>5};
```

$$\begin{aligned} &\{1, \\ &a+b, \\ &2\,a\,b + b^2 + 2, \\ &3\,a\,b^2 + 2\,a + 6\,b + 5, 8\,a\,b + 20\,a + 12\,b^2 + 5\,b + 4, \\ &20\,a\,b^2 + 25\,a\,b + 4\,a + 5\,b^2 + 20\,b + 100, \\ &30\,a\,b^2 + 24\,a\,b + 200\,a + 60\,b^2 + 150\,b + 33, \\ &84\,a\,b^2 + 350\,a\,b + 183\,a + 210\,b^2 + 81\,b + 700, \\ &560\,a\,b^2 + 264\,a\,b + 1120\,a + 249\,b^2 + 1400\,b + 1416, \\ &513\,a\,b^2 + 2520\,a\,b + 4216\,a + 2520\,b^2 + 1944\,b + 3485, \\ &5040\,a\,b^2 + 6160\,a\,b + 6050\,a + 2970\,b^2 + 8525\,b + 21032\} \end{aligned}$$

Wir sehen, daß sich alle Potenzen als Linearkombinationen von sechs Termen 1, a, b, b^2, ab, ab^2 darstellen lassen. Mehr als 6 solcher Ausdrücke sind dann sicher linear abhängig. Finden wir für unser Beispiel eine solche Abhängigkeitsrelation, indem wir eine Linearkombination von 7 verschiedenen Potenzen $(a+b)^i$ mit unbestimmten Koeffizienten aufstellen und diese dann so bestimmen, daß die 6 Koeffizienten vor 1, a, b, b^2, ab, ab^2 alle verschwinden:

```
factor a,b;
u:=for n:=0:6 sum mkid(r,n)*((a+b)^n where {a^2=>2, b^3=>5});
```

$$ab^2(3r_3+20r_5+30r_6)+ab(2r_2+8r_4+25r_5+24r_6)+a(r_1+2r_3+20r_4+4r_5+200r_6)+b^2(r_2+12r_4+5r_5+60r_6)+b(r_1+6r_3+5r_4+20r_5+150r_6)+(r_0+2r_2+5r_3+4r_4+100r_5+33r_6)$$

```
v:=for each s in coeff(u,a) join coeff(s,b);
```

$$\begin{aligned}\{\quad & r_0+2r_2+5r_3+4r_4+100r_5+33r_6,\\ & r_1+6r_3+5r_4+20r_5+150r_6,\\ & r_2+12r_4+5r_5+60r_6,\\ & r_1+2r_3+20r_4+4r_5+200r_6,\\ & 2r_2+8r_4+25r_5+24r_6,\\ & 3r_3+20r_5+30r_6\}\end{aligned}$$

Das ist ein homogenes lineares Gleichungssystem, deshalb fixieren wir eines der r_i, ehe wir die Lösung bestimmen.

```
s:=solve(sub(r6=1,v));
```

$$\{\{r_0=17, r_1=-60, r_4=-6, r_2=12, r_5=0, r_3=-10\}\}$$

Ein Polynom mit der Nullstelle $c=a+b$ lautet also

```
f:=sub(first s,r6=1,for i:=0:6 sum mkid(r,i)*x^i);
```

$$x^6 - 6\,x^4 - 10\,x^3 + 12\,x^2 - 60\,x + 17$$

Testen wir schließlich noch, ob dieses Polynom irreduzibel ist:

```
factorize f;
```

Damit wissen wir, daß es sich bei diesem Polynom sogar um das Minimalpolynom von $c=a+b$ handelt.

BEWEIS : Der Beweis des Satzes geht vollkommen analog. Sind $\alpha_1,\ldots,\alpha_k$ algebraische Zahlen über k vom Grad $d_1,\ldots,d_k$, so ergeben sich aus den entsprechenden Minimalpolynomen

$$p_i(x) = x^{d_i} - q_i(x)$$

Ersetzungsformeln

$$\{\alpha_i^{d_i} \Rightarrow q_i(\alpha_i),\ i=1,\ldots,k\},$$

die es erlauben, jeden polynomialen Ausdruck aus $R := k[\alpha_1,\ldots,\alpha_k]$ als Linearkombination der $D := d_1\cdots d_k$ Produkte $T_{red} := \{\alpha_1^{j_1}\cdots\alpha_k^{j_k} \ :\ 0\le j_i<d_i\}$ zu schreiben.

Ist nun $c\in R$ ein solcher polynomialer Ausdruck (also etwa Summe oder Produkt zweier algebraischer Zahlen), so kann man wie in obigem Beispiel eine nichttriviale lineare Abhängigkeitsrelation zwischen den Potenzen c^i, $i=0,\ldots,D$ finden und erhält damit ein Polynom $p(x)\in k[x]$, dessen Nullstelle c ist. □

Das gefundene Polynom muß allerdings nicht unbedingt das Minimalpolynom sein, da es in Faktoren zerfallen kann.

Aus dem im Beweis verwendeten konstruktiven Ansatz kann man sogar eine weitergehende Aussage ableiten:

Korollar 1 *Sind $\alpha_1, \ldots, \alpha_k$ algebraische Zahlen über k vom Grad $d_1, \ldots, d_k$, so bildet die Menge der k-linearen Kombinationen von Elementen aus T_{red} einen Ring.*

5.5.3 Die Inverse einer algebraischen Zahl

Von einfachen algebraischen Zahlen wie etwa $1+\sqrt{2}$ oder $\sqrt{2}+\sqrt{3}$ wissen wir, daß man die jeweilige Inverse dazu recht einfach darstellen kann, wenn man mit einer auf geeignete Weise definierten *konjugierten* Zahl erweitert. So gilt etwa

$$\frac{1}{1+\sqrt{2}} = \frac{1-\sqrt{2}}{1-2} = \sqrt{2}-1$$

und

$$\frac{1}{\sqrt{2}+\sqrt{3}} = \frac{\sqrt{3}-\sqrt{2}}{3-2} = \sqrt{3}-\sqrt{2}$$

Damit kann man in diesen Fällen auch die Inverse einer algebraischen Zahl (und damit beliebige Quotienten) als k-lineare Kombination der Produkte aus T_{red} darstellen. Es stellt sich die Frage, ob man auch kompliziertere rationale Ausdrücke mit algebraischen Zahlen auf ähnliche Weise vereinfachen kann. Wie sieht es z.B. mit

$$\frac{1}{\sqrt{2}+\sqrt{3}+\sqrt{5}}$$

aus?

Axiom und Derive liefern als Ergebnis sofort

$$\frac{1}{6}\sqrt{3}+\frac{1}{4}\sqrt{2}-\frac{1}{12}\sqrt{30}$$

Für die anderen Systeme sind dazu spezielle Funktionen, Schalter und/oder Pakete notwendig, so in Maple die Funktion `rationalize` aus der gleichnamigen Bibliothek (die in Version V.3 vorher aber geladen werden muß) und in MuPAD die Funktion `radsimp`. In Reduce (ab Version 3.5[6]) muß der Schalter `rationalize` eingeschaltet sein. In Macsyma benötigt man noch intimere Systemkenntnisse: Es ist der Schalter `algebraic` einzuschalten und die CRE-Form zu verwenden:

```
algebraic:true$
a:sqrt(2)+sqrt(3)+sqrt(5)$
1/a;
```

liefert

$$\frac{1}{\sqrt{5}+\sqrt{3}+\sqrt{2}}$$

Erst

```
rat(1/a);
```

ergibt das gewünschte Ergebnis

$$-\frac{\sqrt{2}\sqrt{3}\sqrt{5}-2\sqrt{3}-3\sqrt{2}}{12}$$

Mathematica konnte ich nicht dazu veranlassen, eine entsprechende Darstellung zu finden.

[6] In einigen Patch-Varianten der Version 3.6 (vor Februar 1998) wurde dies versehentlich wieder abgestellt.

Untersuchen wir, auf welchem Wege sich eine solche Darstellung finden ließe. Wie man leicht ermittelt, hat $a = \sqrt{2}+\sqrt{3}+\sqrt{5}$ das charakteristische Polynom $p(x) = x^8-40x^6+352x^4-960x^2+576$, d.h. es gilt

$$a^8 - 40a^6 + 352a^4 - 960a^2 + 576 = 0$$

oder

$$a^{-1} = \frac{-a^7 + 40a^5 - 352a^3 + 960a}{576}$$

Kennt das System das charakteristische Polynom, ist es also nicht schwer, a^{-1} zu berechnen. Es werden einzig noch Ringoperationen benötigt, um den Ausdruck zu vereinfachen:

$$a^{-1} = sub\left(a = \sqrt{2} + \sqrt{3} + \sqrt{5}, \frac{-a^7 + 40a^5 - 352a^3 + 960a}{576}\right)$$

Es erhebt sich die Frage, ob die "schlauen" Systeme dasselbe Programm auch mit komplizierteren algebraischen Zahlen abspulen können. Betrachten wir dazu den Ausdruck

$$A := \frac{\sqrt[3]{4} - 2\sqrt[3]{2} + 1}{\sqrt[3]{4} + 2\sqrt[3]{2} + 1} = \frac{a^2 - 2a + 1}{a^2 + 2a + 1}$$

mit $a = \sqrt[3]{2}$, d.h. $a^3 = 2$.

Derive ist dazu nicht mehr in der Lage. Maple und MuPAD liefern mit obiger Bibliothek, ebenso wie Reduce und Macsyma

```
subs(a=2^(1/3),(a^2-2*a+1)/(a^2+2*a+1));
rationalize(");
expand(");
```

$$\frac{4\sqrt[3]{2}}{3} - 5/3$$

In Axiom hängt das Ergebnis von der Vorgehensweise bei der Eingabe ab.

```
subst((a^2-2*a+1)/(a^2+2*a+1),a=2^(1/3))
```

liefert einen Ausdruck vom Typ `Expression Integer`, der nicht weiter vereinfacht wird:

$$\frac{\sqrt[3]{2}^2 - 2\ \sqrt[3]{2} + 1}{\sqrt[3]{2}^2 + 2\ \sqrt[3]{2} + 1}$$

Geht man dagegen schrittweise vor

```
a:=2^(1/3);
(a^2-2*a+1)/(a^2+2*a+1)
```

$$\frac{4\ \sqrt[3]{2} - 5}{3},$$

so erhält man das gewünschte Ergebnis, allerdings vom Typ `AlgebraicNumber`. Der Grund liegt wiederum im Vorhandensein eines strengen Typsystems. Während im ersten Fall nach Substitution in einen rationalen Ausdruck aus $\mathbf{Z}(a)$ über den so entstehenden Datentyp nichts gefolgert worden ist und damit das Resultat als etwas vom allgemeinsten Typs "irgendwas mit ganzen Zahlen" interpretiert wird, erkennt das System im zweiten Fall, daß a eine algebraische Zahl ist und ruft demzufolge im weiteren die für `AlgebraicNumber` definierten arithmetischen Operationen auf. Nach

dem weiter unter zu beweisenden Satz kann man dabei jede algebraische Zahl als Linearkombination von Termen aus T_{red} darstellen.

Die für eine solche Darstellung notwendigen Rechnungen wollen wir nun mit Reduce an unserem Beispiel nachvollziehen. Wir wollen untersuchen, ob bzw. wie sich der rationale Ausdruck A als Linearkombination von $a^i,\ i = 0, 1, 2$ mit $a = \sqrt[3]{2}$ darstellen läßt. Wir machen dazu den Ansatz

$$A = \frac{a^2 - 2a + 1}{a^2 + 2a + 1} = a^2 r_2 + a r_1 + r_0$$

mit unbestimmten Koeffizienten r_2, r_1, r_0. Ausmultiplizieren und Koeffizientenvergleich liefern

```
u:=(a^2*r2+a*r1+r0);
(a^2-2*a+1)-(a^2+2*a+1)*u where {a^3=>2};
coeff(ws,a);
solve(ws);
```

$$\{\{r_2 = 0, r_1 = \frac{4}{3}, r_0 = \frac{-5}{3}\}\}$$

und damit als Ergebnis unserer Vereinfachung

```
sub(first ws, u);
```

$$\frac{4a - 5}{3}$$

Allgemein gilt der folgende Satz

Satz 14 *Ist α eine algebraische Zahl über k vom Grad d, so bildet die Menge $R := k[\alpha]$ der k-linearen Kombinationen von Termen aus $T_{red} := \{\alpha^i,\ i = 0, \ldots, d-1\}$ einen Körper.*

Kann man in k effektiv rechnen, so auch in R. Genauer: Existiert in k eine kanonische Normalform und ist das Minimalpolynom von α bekannt, so existiert auch in R eine kanonische Normalform.

BEWEIS : Sei $p(x) = x^d - q(x),\ deg\ q < d$ das Minimalpolynom von α.

Wir müssen zum Beweis des Satzes nur zeigen, daß unser Verfahren zur Berechnung der Inversen eines Ausdrucks $A(\alpha) \in R$, das wir oben an einem Beispiel demonstriert haben, stets zum Ziel führt. Es ist dazu das lineare Gleichungssystem in $r_0, \ldots, r_{d-1}$, das man aus

$$A(\alpha) \cdot \sum_{i=0}^{d-1} r_i \alpha^i = 1$$

nach (algebraischer) Ersetzung $\{\alpha^d \ \Rightarrow\ q(\alpha)\}$ durch Koeffizientenvergleich erhält, zu lösen.

Bei diesem Gleichungssystem handelt es sich um ein inhomogenes System von d Gleichungen mit d Unbekannten, dessen zugehöriges homogenes System nur die triviale Lösung besitzt, da jede nichttriviale Lösung des Systems

$$A(\alpha) \cdot \sum_{i=0}^{d-1} r_i \alpha^i = 0$$

zu einem Polynom vom Grad $< d$ mit Nullstelle α führt.

Folglich hat das inhomogene System für jede rechte Seite genau eine Lösung, d.h. die Koeffizienten $r_0, \ldots, r_{d-1}$ lassen sich eindeutig bestimmen. $\square$

5.6 Algebraische Erweiterungstürme

Der letzte Satz zeigt, daß man auch mit mehreren algebraischen Zahlen $\alpha_1, \alpha_2, \ldots, \alpha_n$ effektiv rechnen kann, indem man Schritt für Schritt auf die oben angegebene Art die Erweiterungen $k_i = k_{i-1}[\alpha_i]$ mit $k_0 = k$ konstruiert. Eine solche Folge aufeinander aufbauender algebraischer Körpererweiterungen bezeichnet man als *algebraischen Körperturm*. Zu seiner Konstruktion muß man einzig in der Lage sein, das Faktorisierungsproblem in $k_i[x]$ über den jeweils entstehenden Zwischenkörpern k_i zu lösen.

Dies wird insbesondere dann wichtig, wenn Rechnungen ausgeführt werden sollen, die mehrere algebraische Zahlen mit demselben Minimalpolynom enthalten. Sind nämlich α_1 und α_2 algebraische Zahlen (etwa über $k = \mathbf{Q}$) mit dem gemeinsamen Minimalpolynom $p(x)$, so ist α_2 zwar auch Nullstelle von $p(x)$ über $k_1 = k[\alpha_1]$, aber es gilt

$$p(x) = (x - \alpha_1)\, q(x)$$

für ein geeignetes Polynom $q(x) \in k_1[x]$. α_2 ist dann Nullstelle dieses zweiten Faktors, welcher aber noch immer nicht irreduzibel über k_1 sein muß.

Über entsprechende Algorithmen zur Faktorisierung über algebraischen Erweiterungen verfügen in der einen oder anderen Form außer Derive alle Systeme. Allerdings unterscheidet sich die Art, wie solche algebraischen Zahlen in die Rechnungen eingeführt werden, grundlegend.

MuPAD verfolgt zum Beispiel einen stärker objektorientierten Ansatz, in dem man einen algebraischen Erweiterungskörper als Grundbereich konstruieren und an eine Variable binden kann, die man dann als Parameter an entsprechende Faktorisierungsroutinen übergibt.

```
Q1:=Dom::AlgebraicExtension(Dom::Rational,a1^5-a1+1);
```

```
                                                  5
Dom::AlgebraicExtension(Dom::Rational, - a1 + a1  + 1 = 0, a1)
```

führt etwa den algebraischen Erweiterungskörper $Q_1 = \mathbf{Q}[a_1]$ ein, dessen erzeugendes Element a_1 das Minimalpolynom $p := x^5 - x + 1$ besitzt. Faktorisieren wir dieses Polynom über Q_1, so zerfällt es in der Tat in zwei Faktoren:

```
Factor(poly(x^5-x+1,Q1));
```

$$(x - a_1)\left(x^4 + a_1^4 + x a_1^3 + x^3 a_1 + x^2 a_1^2 - 1\right)$$

Eine zweite Nullstelle a_2 von p hat also über Q_1 den zweiten Faktor p_1 als Minimalpolynom, so daß wir die entsprechende Erweiterung

```
Q2:=Dom::AlgebraicExtension(Q1,subs(p1,x=a2)):
```

als Körperturm erhalten, über dem wir wiederum versuchen können, die Faktorzerlegung unseres Ausgangspolynom $p(x)$ zu finden, die diesmal die beiden Linearfaktoren $(x - a_1)$ und $(x - a_2)$ enthalten sollte.

```
factor(poly(x^5-x+1,Q2));
```

Allerdings hatte ich nicht die Geduld, auf das Resultat zu warten.

Maple gestattet die Definition algebraischer Zahlen über einen alias-Mechanismus. Auf diese Weise wird mit einem entsprechenden RootOf-Ausdruck ein neues Symbol verbunden, das wie in MuPAD zugleich als Printname[7] dient, wodurch die entstehenden Formeln kompakt notiert werden können. Für obiges Beispiel erhält man

[7] Leider nicht bei der LaTeX-Ausgabe.

```
alias(a1=RootOf(x^5-x+1)):
s:=factor(x^5-x+1,a1);
```

$$(x^4 + a_1\,x^3 + a_1^2\,x^2 + a_1^3\,x - 1 + a_1^4)\,(x - a_1)$$

```
p1:=op(1,s):
alias(a2=RootOf(p1)):
s1:=factor(x^5-x+1,{a1,a2});
```

Nach bereits 36 s. (auf einer Sun UltraSparc mit Maple V.4) erhält man

$$(a_2\,x^2 + a_1^2\,x + a_1\,x^2 + x^3 + a_1^3 + a_1\,a_2\,x + a_2^3 + a_2^2\,x + a_1^2\,a_2 + a_1\,a_2^2)\,(x - a_2)\,(x - a_1)$$

das mit dem speziellen Befehl zur Simplifikation von Ausdrücken mit algebraischen Zahlen

```
evala(s1);
```

wieder zu

$$x^5 - x + 1$$

vereinfacht.

Mathematica verfolgt einen ähnlichen Zugang, wobei allerdings für die langen RootOf-Ausdrücke nicht automatisch kurze Printsymbole eingeführt werden. Definieren wir wie oben a_1 als algebraische Zahl mit dem Minimalpolynom $x^5 - x + 1$ und geben sie beim Faktorisieren explizit an, so erhalten wir dieselbe Faktorzerlegung wie mit Maple

```
a1=Root[x^5-x+1,1];
u=Factor[x^5-x+1,Extension->{a1}]
```

$$(x - a_1)\left(-1 + x^4 + x^3\,a_1 + x^2\,a_1^2 + x\,a_1^3 + a_1^4\right)$$

Die zweite Nullstelle a_2 kann man nicht als Körperturm wie in Maple definieren:

```
a2=Root[u[[2]],1]
```

führt nämlich auf

$$\mathrm{Root}[1 - \#1 + \#1^5\&, 2],$$

also zu keinem anderen Ergebnis als wenn man die zweite Nullstelle a_2 des Polynoms p direkt hinzugefügt hätte. Faktorisiert man erneut

```
a2=Root[x^5-x+1,2];
s=Factor[x^5-x+1,Extension->{a1,a2}]
```

so erhält man nach geduldigem Warten als Ergebnis einen riesigen Ausdruck mit sehr langen Koeffizienten, der zwei lineare und einen kubischen Faktor enthält. Die beiden linearen Faktoren haben allerdings nicht die (syntaktische) Gestalt $x - a_1$ bzw. $x - a_2$, sondern enthalten als Absolutglied jeweils kompliziertere polynomiale Audrücke, in denen a_1 und a_2 jeweils bis zum Grad 4 vertreten sind. Das ist nicht verwunderlich, da die im Abschnitt 5.5.2 angegebene Menge T_{red} für mehr als eine algebraische Zahl nicht mehr linear unabhängig sein muß, d.h. nicht unbedingt eine kanonische Darstellung liefert. In der Tat hat a_2 über $\mathbf{Q}[a_1]$ ein Minimalpolynom vom Grad 4, d.h. auch Potenzen a_2^4 könnten weiter normiert werden.

Aus demselben Grund führt Ausklammern und Simplifizieren mit `Simplify@Expand[s]`[8] nicht zum Ausgangspolynom zurück.

Einzig **Axiom** ist in der Lage, in diesem Beispiel schnell und zuverlässig die Minimalpolynome der einzelnen Nullstellen a_i, $i = 1, \ldots, 5$, des Polynoms $x^5 - x + 1$ jeweils über $k_0 = \mathbf{Q}, k_1 = k_0[a_1]$ etc. auszurechnen, d.h. das sogenannte *Zerfällungskörper-Problem* zu lösen:

[8] Nach Information von M. Trott ist wegen der internen Gestaltung von `Simplify` der Befehl `RootReduce@Expand[s]` angemessener.

```
l:=rootsOf(p,x)
  [%x8,%x9,%x10,%x11,- %x11 - %x10 - %x9 - %x8]

                                Type: List Expression Integer
```

Wir erhalten eine Liste der 5 Nullstellen, für die von Axiom selbsttätig neue Symbole eingeführt worden sind. Auch die zugehörigen Minimalpolynome im Sinne eines Körperturms sind bekannt:

```
definingPolynomial %x8
```

$$x_8^5 - x_8 + 1$$

```
definingPolynomial %x9
```

$$x_8^4 + x_9 x_8^3 + x_9^2 x_8^2 + x_9^3 x_8 + x_9^4 - 1$$

```
definingPolynomial %x10
```

$$x_9^3 + (x_8 + x_{10})x_9^2 + \left(x_8^2 + x_{10}\, x_8 + x_{10}^2\right) x_9 + x_8^3 + x_{10}\ x_8^2 + x_{10}^2\ x_8 + x_{10}^3$$

```
definingPolynomial %x11
```

$$x_{10}^2 + (x_9 + x_8 + x_{11})\, x_{10} + x_9^2 + (x_8 + x_{11})\, x_9 + x_8^2 + x_{11}\, x_8 + x_{11}^2$$

Die fünfte Nullstelle kann schließlich durch die anderen rational ausgedrückt werden, so daß keine weitere Körpererweiterung notwendig ist.

Die in den anderen Systemen explizit ausgeführte Faktorisierung über algebraischen Zahlen läßt sich allerdings nicht ohne weiteres ausführen.

```
factor(p,%x8)
```

ergibt einen Typfehler; die vom System selbst erzeugte algebraische Zahl x_8 ist vom Typ `Expression Integer`, während beim Faktorisieren als Zusatzparameter eine algebraische Zahl, also eine Variable vom Typ `AlgebraicNumber` (kurz `AN`) benötigt wird. Solche Zahlen können mit der `rootOf`-Funktion von Axiom erzeugt werden. Allerdings liefert

```
a1:=rootOf(subst(p,x=a1))
```

wieder den falschen Typ `Expression Integer`, während der explizite Aufruf

```
a1:=rootOf(a1^5-a1+1)
```

endlich den richtigen Datentyp ergibt, mit dem

```
u1:=factor(p,[a1])
```

dasselbe Ergebnis wie die anderen Systeme liefert:

$$(x - a_1)\ \left(x^4 + a_1\, x^3 + a_1^2\, x^2 + a_1^3\, x + a_1^4 - 1\right)$$

Um den zweiten Faktor aus diesem Ergebnis vom Typ `Factored Polynomial AlgebraicNumber` zu extrahieren, ist ein Blick in die Dokumentation unumgänglich, denn es muß dafür eine entsprechende Methode des Datentyp(konstruktor)s `Factored` verwendet werden, in diesem Fall die Funktion `nthFactor`:

```
p1:=nthFactor(u1,2)
```

Nun gilt es, a_2 als Nullstelle dieses Polynoms zu vereinbaren. Leider geht das nicht mehr auf die oben angegebene Weise, da sich der Typ `Expression Integer` bei Verwendung zweier Variabler, a_1 und a_2, kaum noch vermeiden läßt. Statt dessen ist ein *sehr* tiefer Blick in die Interna notwendig. Neben dem Datentypkonstruktor `Polynomial` gibt es in Axiom noch die spezielleren Konstruktoren `UnivariatePolynomial` (kurz UP) und `SparseUnivariatePolynomial` (kurz SUP). Unter den vielen Syntaxmöglichkeiten für `rootOf`[9]

```
-> )display operations rootOf

There are 5 exposed functions called rootOf :
   [1] (SparseUnivariatePolynomial D,Symbol) -> D from D if D has ACF

   [2] SparseUnivariatePolynomial D -> D from D if D has ACF
   [3] Polynomial D -> D from D if D has ACF
   [4] (D,Symbol) -> D from D
            if D has ACFS D2 and D2 has Join(OrderedSet,IntegralDomain)

   [5] D -> D from D
            if D has ACFS D1 and D1 has Join(OrderedSet,IntegralDomain)
```

ist für unsere Zwecke nur die erste von Interesse. Wir müssen also unser Polynom p_1 vom Typ `Polynomial AN` erst in den Typ `SUP AN` verwandeln, um die entsprechende Funktion

```
   rootOf : (SUP AN, Symbol) -> AN
```

zum Erzeugen einer neuen algebraischen Zahl[10] aufrufen zu können. Glücklicherweise kennt Axiom wenigstens die entsprechende Konversionsroutine, so daß zum Schluß alles wieder ganz einfach aussieht:

```
a2:=rootOf(p1::SUP AN,a2)
```

Nun können wir unser Ausgangspolynom faktorisieren

```
factor(p,[a1,a2])
```

und erhalten schließlich (nach ebenfalls geduldigem Warten) ähnlich wie in Maple

$$(x-a_2)\,(x-a_1)\left(x^3+(a_2+a_1)\,x^2+\left(a_2^2+a_1\,a_2+a_1^2\right)x+a_2^3+a_1\,a_2^2+a_1^2\,a_2+a_1^3\right)$$

vom Typ `Factored Polynomial AlgebraicNumber`.

Macsyma vermag Polynome nur über einer einfachen Erweiterung der ganzen Zahlen zu faktorisieren. Dazu kann man als zweiten Parameter der Funktion `factor` das entsprechende Minimalpolynom angeben.

```
factor(x^5-x+1,a1^5-a1+1);
```

$$(x-a_1)\,\left(x^4+a_1\,x^3+a_1^2\,x^2+a_1^3\,x+a_1^4-1\right)$$

[9] Diese werden, wie in polymorphen Ansätzen üblich, an Hand des Namens *und* der Signatur, in Axiom sogar unter Einbeziehung des Rückgabetyps, unterschieden

[10] AN = AlgebraicNumber hat glücklicherweise die Eigenschaft ACF = AlgebraicallyClosedField

Reduce verfügt standardmäßig nur über rudimentäre Fähigkeiten, kompliziertere algebraische Zahlen einzuführen und mit ihnen zu rechnen. Definiert man etwa eine solche Zahl a über einen entsprechenden RootOf-Ausdruck `a:=root_of(x^d-q(x),x)`, so wird in nachfolgenden Rechnungen die algebraische Ersetzung $a^d \Rightarrow q(a)$ ausgeführt, aber bereits die oben beschriebene Inversenberechnung erfolgt nicht (selbst unter `on rationalize;`). Um solche Rechnungen doch noch auszuführen, steht das Reduce-Paket `arnum` zur Verfügung. Allerdings ist dieses Paket nur ungenügend integriert und führt schnell zu relativ unverständlichen Fehlermeldungen. Außerdem ist es in seiner Leistungsfähigkeit nur steckenweise mit den bisher besprochenen Systemen vergleichbar.

5.7 Algebraische Zahlen mit gemeinsamem Minimalpolynom und symmetrische Funktionen

In diesem Abschnitt wollen wir die Beziehungen zwischen den verschiedenen Nullstellen eines (irreduziblen) Polynoms genauer studieren.

Wir hatten weiter oben gesehen, daß es durchaus interessante solche Beziehungen geben kann, deren Kenntnis für das qualifizierte Rechnen mit algebraischen Zahlen folglich wichtig ist. So hatten wir etwa rationale Ausdrücke für die Potenzsummen der Nullstellen eines Polynoms dritten Grades gefunden oder Beziehungen zwischen verschiedenen algebraischen Zahlen trigonometrischer Natur entdeckt.

Derartige Beziehungen bestehen auch zwischen algebraischen Zahlen höheren als dritten Grades und sind einigen der betrachteten CAS wohlbekannt. Betrachten wir dazu zwei Beispiele in Maple. Berechnen wir zuerst die Potenzsummen über die fünf Wurzeln des Polynoms $p = x^5 - x + 1$ für $k = 1, \ldots, 9$:

```
u:=solve(x^5-x+1,x):
[seq(sum(v^k,'v'=u),k=1..9)];
```

$$[0, 0, 0, 4, -5, 0, 0, 4, -9]$$

Bis auf die aus mathematischer Sicht etwas komische Schreibweise ist das Ergebnis klarer ausgefallen als für ein Polynom dritten Grades, wo explizit mit Wurzelausdrücken gerechnet wurde. Allerdings konnte die Funktion `sum` hier das RootOf-Symbol explizit entdecken und so einen entsprechenden Algorithmus aufrufen.

Noch interessanter fällt das Ergebnis der folgenden Vereinfachung aus, wenn wir die Summe $\sum \frac{1}{x-\alpha}$ über die fünf Wurzeln obigen Polynoms bilden:

```
sum(1/(x-v),'v'=u);
```

$$\frac{5\,x^4 - 1}{1 - x + x^5}$$

In diese Summe rationaler Ausdrücke gehen die verschiedenen Nullstellen in den Summanden auf gleiche Weise ein und es ergibt sich ein Ausdruck ganz ohne algebraische Zahlen!

Im folgenden wollen wir eine Begründung für diese auf den ersten Blick vielleicht verblüffenden Beziehungen zwischen algebraischen Zahlen sehr komplizierter Bauart geben. Wir sehen daran noch einmal, daß die RootOf-Darstellung gegenüber einer solchen durch Radikale den großen Vorteil bietet, daß sich aus ihr wichtige Eigenschaften der eingeführten Symbole auf relativ einfache Weise extrahieren lassen.

All diesen Ausdrücken ist gemeinsam, daß die verschiedenen Nullstellen $Z = (z_1, \ldots, z_n)$, die sich hinter einem gemeinsamen RootOf-Symbol vom Grad n mit dem Minimalpolynom

$$p(x) = x^n + a_1 x^{n-1} + \ldots + a_{n-1} x + a_n$$

verbergen, in diese auf symmetrische Weise eingehen, d.h. sich beim Ausmultiplizieren und Bilden eines Hauptnenners als Koeffizienten symmetrische Polynome in diesen Nullstellen ergeben. Nach dem **Vietaschen Wurzelsatz** (Kapitel 2) sind die elementarsymmetrischen Polynome $e_k = e_k(X)$ eng mit den Koeffizienten von $p(x)$ verbunden: Es gilt

$$a_i = (-1)^i \, e_i(Z), \quad i = 1, \ldots, n$$

Wir wollen deshalb an dieser Stelle zunächst untersuchen, wie *beliebige* symmetrische Polynome mit den elementarsymmetrischen Polynomen zusammenhängen.

Sei dazu $h = h(X)$ ein symmetrisches Polynom in $X = (x_1, \ldots, x_n)$ mit Koeffizienten aus dem Körper k, das in distributiver Form mit lexikographisch angeordneten Termen dargestellt sei. Wegen der Symmetrie hat der erste Summand von h in einer solchen Darstellung die Gestalt

$$c \, x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n}$$

mit $c \in k$ und $a_1 \geq a_2 \geq \ldots \geq a_n$. Da das ebenfalls symmetrische Polynom

$$c \, e_1^{a_1 - a_2} e_2^{a_2 - a_3} \cdots e_{n-1}^{a_{n-1} - a_n} e_n^{a_n}$$

denselben höchsten Term hat (wegen $e_i = x_1 \cdot \ldots \cdot x_i + \langle kleinere \; Terme \rangle$), so ergibt sich als Differenz beider Polynome ein neues symmetrisches Polynom, dessen höchster Term lexikographisch kleiner ist.

Betrachten wir zur Veranschaulichung dieses Vorgehens zunächst einige Beispiele:

$$p_1 := \sum_{i=1}^{5} x_i^2 = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2$$

Hier ist $a_1 = 2$, $a_2 = \ldots = a_5 = 0$. Es gilt $e_1^2 = x_1^2 + \langle kleinere \; Terme \rangle$. Als Differenz der beiden symmetrischen Polynome erhalten wir das symmetrische Polynom

$$p_1 - e_1^2 = -2(x_1x_2 + x_1x_3 + x_1x_4 + x_1x_5 + x_2x_3 + x_2x_4 + x_2x_5 + x_3x_4 + x_3x_5 + x_4x_5),$$

dessen Leitterm x_1x_2 lexikographisch kleiner als x_1^2 ist. Da e_2 denselben Leitterm hat, erhalten wir schließlich

$$p_1 - e_1^2 + 2e_2 = 0,$$

also $p_1 = e_1^2 - 2e_2$.

Auf diese Weise kann man von jedem symmetrischen Polynom Schritt für Schritt Produkte elementarsymmetrischer Polynome abziehen, so daß sich der Leitterm jeweils verkleinert, bis sich das Nullpolynom ergibt.

$$p_2 := x_1^3 + x_2^3 + x_3^3 + x_4^3 + x_5^3$$

$$\begin{aligned} p_2 - e_1^3 = \; & 3(-x_1^2x_2 - x_1^2x_3 - x_1^2x_4 - x_1^2x_5 - x_1x_2^2 - 2x_1x_2x_3 - 2x_1x_2x_4 - 2x_1x_2x_5 - x_1x_3^2 - \\ & 2x_1x_3x_4 - 2x_1x_3x_5 - x_1x_4^2 - 2x_1x_4x_5 - x_1x_5^2 - x_2^2x_3 - x_2^2x_4 - x_2^2x_5 - x_2x_3^2 - \\ & 2x_2x_3x_4 - 2x_2x_3x_5 - x_2x_4^2 - 2x_2x_4x_5 - x_2x_5^2 - x_3^2x_4 - x_3^2x_5 - x_3x_4^2 - 2x_3x_4x_5 - \\ & x_3x_5^2 - x_4^2x_5 - x_4x_5^2) \end{aligned}$$

$$\begin{aligned} p_2 - e_1^3 + 3e_1e_2 = \; & 3(x_1x_2x_3 + x_1x_2x_4 + x_1x_2x_5 + x_1x_3x_4 + x_1x_3x_5 + x_1x_4x_5 + x_2x_3x_4 + x_2x_3x_5 + \\ & x_2x_4x_5 + x_3x_4x_5) \end{aligned}$$

$$p_2 - e_1^3 + 3e_1e_2 - 3e_3 = 0,$$

also $p_2 = e_1^3 - 3e_1e_2 + 3e_3$.

Dieses Verfahren terminiert in jedem Fall, weil die lexikographische Ordnung noethersch ist. Damit haben wir den folgenden Satz bewiesen:

Satz 15 (Hauptsatz über symmetrische Polynome)
Jedes symmetrische Polynom $P(\mathbf{x}) \in k[x_1, \ldots, x_n]$ läßt sich (eindeutig) als polynomialer Ausdruck $Q(e_1, \ldots, e_n)$ in den elementarsymmetrischen Polynomen darstellen.

Korollar 2 *Sind $z_1, \ldots, z_n$ die Nullstellen eines (nicht unbedingt irreduziblen) Polynoms $p(x) \in \mathbf{Q}[x]$ mit rationalen Koeffizienten, so ist jeder symmetrische polynomiale Ausdruck in $z_1, \ldots, z_n$ eine rationale Zahl.*

Dies gilt insbesondere für die Potenzsummen und allgemeiner für Ausdrücke der Form

$$\sum_{a \in RootOf(p(x))} f(a),$$

wenn $f(x)$ ein polynomialer Ausdruck ist.

Ausdrücke dieser Art, in denen die verschiedenen Nullstellen *separiert* in die einzelnen Summanden eingehen, lassen sich mit der von Maple angebotenen Kombination der Funktionssymbole `sum` und RootOf vereinfachen.

Allerdings gibt es auch Fragestellungen, in denen die Nullstellen auf kompliziertere Weise miteinander verbunden sind. Seien z.B. z_1, z_2, z_3, z_4 die vier Nullstellen des Polynoms $x^4 + x + 1$. Wir wollen ein Polynom $p(x)$ mit den sechs Nullstellen $\{z_i z_j \ : \ 1 \leq i < j \leq 4\}$ bestimmen. Dieses Polynom sechsten Grades ergibt sich als

$$p(x) = \prod_{1 \leq i < j \leq 4} (y - z_i z_j).$$

Expandieren wir die Faktoren und sammeln die Koeffizienten vor den entsprechenden Potenzen y^i zusammen, so zeigt sich, daß diese symmetrisch in $z_1, \ldots, z_4$ sind:

$$\begin{aligned}
p(x) = \; & \\
& y^6 - y^5(z_1z_2 + z_1z_3 + z_1z_4 + z_2z_3 + z_2z_4 + z_3z_4) \\
& + y^4(z_1^2z_2z_3 + z_1^2z_2z_4 + z_1^2z_3z_4 + z_1z_2^2z_3 + z_1z_2^2z_4 + z_1z_2z_3^2 + 3z_1z_2z_3z_4 \\
& \qquad + z_1z_2z_4^2 + z_1z_3^2z_4 + z_1z_3z_4^2 + z_2^2z_3z_4 + z_2z_3^2z_4 + z_2z_3z_4^2) \\
& + y^3(-z_1^3z_2z_3z_4 - z_1^2z_2^2z_3^2 - 2z_1^2z_2^2z_3z_4 - z_1^2z_2^2z_4^2 - 2z_1^2z_2z_3^2z_4 - 2z_1^2z_2z_3z_4^2 - z_1^2z_3^2z_4^2 \\
& \qquad - z_1z_2^3z_3z_4 - 2z_1z_2^2z_3^2z_4 - 2z_1z_2^2z_3z_4^2 - z_1z_2z_3^3z_4 - 2z_1z_2z_3^2z_4^2 - z_1z_2z_3z_4^3 - z_2^2z_3^2z_4^2) \\
& + y^2z_1z_2z_3z_4(z_1^2z_2z_3 + z_1^2z_2z_4 + z_1^2z_3z_4 + z_1z_2^2z_3 + z_1z_2^2z_4 + z_1z_2z_3^2 + 3z_1z_2z_3z_4 + z_1z_2z_4^2 \\
& \qquad + z_1z_3^2z_4 + z_1z_3z_4^2 + z_2^2z_3z_4 + z_2z_3^2z_4 + z_2z_3z_4^2) \\
& - yz_1^2z_2^2z_3^2z_4^2(z_1z_2 + z_1z_3 + z_1z_4 + z_2x_3 + z_2z_4 + z_3z_4) + z_1^3z_2^3z_3^3z_4^3 \\
= \; & y^6 - y^5e_2 + y^4h_1 - y^3h_2 + y^2e_4h_1 - ye_2e_4^2 + e_4^3,
\end{aligned}$$

wobei h_1 und h_2 für die entsprechenden symmetrischen Klammerausdrücke stehen. Mit unserer Expansionstechnik erhält man

$$\begin{aligned}
h_1 &= e_1e_3 - e_4, \\
h_2 &= e_1^2e_4 + e_3^2 - 2e_2e_4
\end{aligned}$$

und damit

$$p(x) = y^6 - y^5e_2 + y^4(e_1e_3 - e_4) - y^3(e_1^2e_4 + e_3^2 - 2e_2e_4) + y^2e_4(e_1e_3 - e_4) - ye_2e_4^2 + e_4^3.$$

Nach dem Vietaschen Wurzelsatz folgt $e_1 = e_2 = 0, e_3 = e_4 = 1$, also insgesamt

$$p(x) = y^6 - y^4 - y^3 - y^2 + 1.$$

5.7.1 Symmetrische Summen rationaler Ausdrücke

Wir hatten oben in einem Beispiel gesehen, daß es Fälle gibt, in denen sich auch eine Summe von rationalen Ausdrücken, in die die verschiedenen Nullstellen ein und desselben Polynoms auf symmetrische Weise eingehen, auf rationale Weise durch die Koeffizienten der beteiligten Polynome ausdrücken lassen kann. So liefert etwa Maple für die Summe

$$\sum \frac{1}{x-\alpha} = \frac{5\,x^4-1}{1-x+x^5},$$

wobei über $\alpha \in \text{RootOf}(x^5 - x + 1)$ summiert wurde.

Wir wollen zum Abschluß dieses Abschnitts allgemeiner die Summation von Ausdrücken der Gestalt

$$\sum_{g(\alpha)=0} \frac{c(\alpha)}{x-\alpha}$$

mit Polynomen $c(x)$, $g(x) \in k[x]$ betrachten (in obigem Beispiel ist $c(x) = 1$).

Satz 16 *Sei $g(x) \in k[x]$ ein Polynom mit einfachen Nullstellen. Dann läßt sich jede Summe*

$$S := \sum_{g(\alpha)=0} \frac{c(\alpha)}{x-\alpha}$$

in der Form $\frac{f(x)}{g(x)}$ mit einer polynomialen Funktion $f(x) \in k[x]$ vom Grad deg $f <$ deg g darstellen.

Beweis : OBdA sei der Leitkoeffizient von g gleich 1. Da g nur einfache Nullstellen hat, gilt $g(x) = \prod_b (x-b)$, wobei sich das Produkt über alle Nullstellen von g erstreckt. Multiplizieren wir S mit $g(x)$, so erhalten wir ein Polynom vom Grad $<$ *deg* g, das als einziger Kandidat für f in Frage kommt:

$$f(x) = \sum c(a)\frac{g(x)}{x-a} = \sum c(a) \prod_{b\neq a} (x-b)$$

Es bleibt zu zeigen, daß die Koeffizienten dieses Polynoms bereits in k liegen. Nun gilt aber $g'(a) = \prod_{b\neq a}(a-b)$ und obige Formel ist gerade die Lagrange-Interpolationsformel für ein Polynom, das an den Stellen $a \in \text{RootOf}(g(x))$ die Werte $f(a) = c(a)\,g'(a)$ annimmt. Da einerseits $c(x)\,g'(x)\ (mod\ g(x))$ ein solches Polynom ist und sich andererseits das Interpolationspolynom eindeutig ergibt, schließen wir $f(x) = c(x)\,g'(x)\ (mod\ g(x)) \in k[x]$. □

Ähnlich kann man auch mit beliebigen Summen rationaler Ausdrücke verfahren, in die die verschiedenen Nullstellen eines (quadratfreien) Polynoms symmetrisch eingehen. Ist etwa wie oben

```
a:=RootOf(x^5-x+1,x);
```

$$RootOf(\_Z^5 - \_Z + 1)$$

die Menge der Nullstellen dieses Polynoms fünften Grades, so gilt

```
sum(1/(x-u^2),u=a);
```

$$\frac{5\,x^4 - 6\,x^2 + 1}{x - 2\,x^3 + x^5 - 1}$$

```
sum(1/(x^2-u),u=a);
```

$$\frac{5\,x^8-1}{1-x^2+x^{10}}$$

```
sum((x+u)/(x-u^2),u=a);
```

$$\frac{5\,x^5-6\,x^3+x+1-5\,x^2}{x-2\,x^3+x^5-1}$$

Allgemein gilt der folgende Satz:

Satz 17 *Ist $f(x,y) \in k(x,y)$ eine rationale Funktion und $g(x) \in k[x]$ ein Polynom mit einfachen Nullstellen, so läßt sich*

$$S(x) := \sum_{g(\alpha)=0} f(x,\alpha)$$

als rationale Funktion $S(x) \in k(x)$ mit Koeffizienten aus dem Grundkörper k darstellen.

Kapitel 6

Die Stammfunktion einer rationalen Funktion

Aus dem Grundkurs Analysis ist bekannt, daß jede rationale Funktion $r(x) = \frac{f(x)}{g(x)}$ mit teilerfremden f, g eine Stammfunktion besitzt, die sich durch elementare Funktionen (genauer: rationale Funktionen und Logarithmen) ausdrücken läßt. Dort wird gewöhnlich auch ein Verfahren zur Bestimmung dieser Stammfunktion erläutert, das auf NEWTON und LEIBNIZ zurückgeht und erstmals von J.BERNOULLI (1703) vollständig begründet wurde. Es geht von der Zerlegung des Nenners $g(x)$ in komplexe Linearfaktoren bzw. reelle lineare und quadratische Faktoren aus und verwendet eine *Partialbruchzerlegung*, deren Summanden bekannte Stammfunktionen besitzen. Für den komplexen Fall, auf den wir uns im weiteren der Einfachheit halber beschränken wollen, gilt dann folgender Satz

Satz 18 (Newton-Leibniz-Bernoulli) *Seien $\alpha_1, \ldots, \alpha_d \in \mathbf{C}$ die verschiedenen Nullstellen von $g(x)$ mit den Vielfachheiten $\nu_1, \ldots, \nu_d$. Dann existieren Polynome $p(x), q_1(x), \ldots, q_d(x) \in \mathbf{C}[x]$ und Konstanten $c_1, \ldots, c_d \in \mathbf{C}$ mit*

$$\int r(x)\,dx = p(x) + \sum_i \frac{q_i(x)}{(x-\alpha_i)^{\nu_i - 1}} + \sum_i c_i \log(x - \alpha_i)$$

und $deg(p) \leq deg(r) := deg(f) - deg(g)$ sowie $deg(q_i) < \nu_i - 1$.

Die Stammfunktion besteht also aus einem *rationalen Anteil*

$$r_1(x) = p(x) + \sum_i \frac{q_i(x)}{(x-\alpha_i)^{\nu_i - 1}}$$

und einem *transzendenten Anteil*

$$r_2(x) = \sum_i c_i \log(x - \alpha_i).$$

Um eine solche Stammfunktion bei vorgegebener rationaler Funktion $r(x)$ zu konstruieren, könnte man die einzelnen Parameter p, q_i, c_i in obiger Formel mit unbestimmten Koeffizienten ansetzen, die abgeleitete Gleichung

$$f(x) = \left(p'(x) + \left(\sum_i \frac{q_i(x)}{(x-\alpha_i)^{\nu_i - 1}} \right)' + \sum_i \frac{c_i}{(x-\alpha_i)} \right) g(x)$$

aufstellen, in der der Ausdruck auf der rechten Seite ebenfalls polynomial ist, und durch Koeffizientenvergleich ein lineares Gleichungssystem gewinnen, dem die unbestimmten Koeffizienten genügen müssen.

Unter der Annahme, daß $deg(f) < deg(g) =: n$ gilt, also $p(x) = 0$ gesetzt werden kann, sind die verbleibenden Koeffizienten sogar eindeutig bestimmt. In der Tat, dann ergeben sich nichttriviale Bedingungen auf die $\nu_1 + \ldots + \nu_d = n$ unbestimmten Koeffizienten genau für die n Potenzen $x^0, \ldots, x^{n-1}$. Wir erhalten also ein lineares Gleichungssystem mit n Unbestimmten und n Gleichungen, das nach obigem Satz für jede rechte Seite $f(x)$ eine Lösung besitzt, welche demzufolge eindeutig bestimmt ist.

Allerdings ist ein solcher Ansatz wenig praktikabel, da stets mit allen Nullstellen der Gleichung $g(x)$ zu rechnen ist, d.h. komplizierte und voneinander stark abhängige algebraische Zahlen einzuführen sind, während z.B. in der Stammfunktion

$$\int \frac{3x^2 - 1}{x^3 - x + 1}\, dx = \ln(x^3 - x + 1)$$

überhaupt keine algebraischen Zahlen auftreten.

Wir wollen deshalb, ausgehend von einem Körper k mit $\mathbf{Q} \subseteq k \subset \mathbf{C}$, in dem effektiv gerechnet werden kann, und einer rationalen Funktion $r(x) = \frac{f(x)}{g(x)} \in k(x)$, im folgenden die einzelnen Schritte des Algorithmus daraufhin untersuchen, inwieweit sie ohne die Einführung (neuer) algebraischer Zahlen ausgeführt werden können.

Für eine detailliertere Diskussion der Integration rationaler Funktionen, insbesondere auch die Behandlung des reellen Falls, sei auf [1, Kap. 2] verwiesen.

6.1 Der rationale Anteil der Stammfunktion

Wir können zunächst $deg(f) < deg(g)$ voraussetzen, da man andernfalls von $r(x)$ durch Division mit Rest einen polynomialen Anteil abspalten kann, dessen Stammfunktion sich leicht berechnen läßt wie in folgendem Beispiel:

$$\int \frac{x^5 - x + 1}{x^3 - x + 1}\, dx = \int (x^2 + 1)\, dx - \int \frac{x^2}{x^3 - x + 1}\, dx = \frac{x^3}{3} + x - \int \frac{x^2}{x^3 - x + 1}\, dx$$

Als zweiten Schritt können wir ohne Einführung zusätzlicher algebraischer Zahlen die Berechnung von $\int r(x)\, dx$ auf die Berechnung eines Integral mit *quadratfreiem* Nenner reduzieren, d.h. $g(x)$ durch ein Polynom ersetzen, das nur einfache Nullstellen besitzt. Es stellt sich heraus, daß bei diesem auf M.W.Ostrogradski zurückgehenden Ansatz bereits der gesamte rationale Anteil der Stammfunktion berechnet wird, d.h. dafür im Gegensatz zur Darstellung im Satz von Newton-Leibniz-Bernoulli keine neuen algebraischen Zahlen eingeführt werden müssen.

Wir wollen o.B.d.A. annehmen, daß $g(x)$ monisch ist, also

$$g(x) = (x - \alpha_1)^{\nu_1} \cdot \ldots \cdot (x - \alpha_d)^{\nu_d}$$

gilt. Dann ist

$$g_0(x) = (x - \alpha_1) \cdot \ldots \cdot (x - \alpha_d)$$

ein Polynom, das dieselben Nullstellen besitzt, jedoch jede nur mit der Vielfachheit 1, und schließlich

$$g_1(x) = \frac{g(x)}{g_0(x)} = (x - \alpha_1)^{\nu_1 - 1} \cdot \ldots \cdot (x - \alpha_d)^{\nu_d - 1}$$

der zugehörige Kofaktor.

Satz 19 *Es gilt* $\gcd(g(x), g'(x)) = g_1(x)$ *und damit* $g_0, g_1 \in k[x]$.

BEWEIS : Bezeichnen wir mit

$$g^{(i)}(x) = \frac{g_0(x)}{x - \alpha_i},\ i = 1, \ldots, d$$

das Produkt der Linearfaktoren $(x - \alpha_j)$, $j \neq i$, so gilt nach der Produktregel für die Ableitung

$$g'(x) = \left(\sum_i \nu_i g^{(i)}(x) \right) g_1(x).$$

Da andererseits $g^{(i)}(x) = 0$ für $x = \alpha_j$, $j \neq i$, und $g^{(i)}(\alpha_i) \neq 0$ gilt, ist keine der Nullstellen von $g(x)$ eine Nullstelle des ersten Faktors. Folglich gilt $\gcd(g(x), g'(x)) = g_1(x)$.

Da der größte gemeinsame Teiler z.B. mit dem Euklidschen Algorithmus berechnet werden kann, entstehen seine Koeffizienten durch arithmetische Operationen aus denen von $g(x)$ und $g'(x)$, liegen also ebenfalls in k. □

Ein Vergleich mit dem rationalen Anteil $r_1(x)$ der Stammfunktion von $r(x)$ im Satz von Newton-Leibniz-Bernoulli zeigt, daß als Hauptnenner von r_1 gerade $g_1(x)$ auftritt. Es mag deshalb nicht verwundern, daß $r_1 = \frac{f_1}{g_1} \in k(x)$ gilt und die obige Darstellung durch Partialbrüche an dieser Stelle unnötige algebraische Zahlen einführt. Den Zähler f_1 kann man als Lösung eines entsprechenden linearen Gleichungssystems bestimmen. Dazu machen wir den Ansatz

$$r(x) = \frac{f(x)}{g(x)} = \left(\frac{f_1(x)}{g_1(x)} \right)' + \frac{f_0(x)}{g_0(x)}$$

mit unbestimmten Polynomen f_0, f_1 vom Grad $deg(f_0) < deg(g_0)$ und $deg(f_1) < deg(g_1)$.

Betrachten wir wiederum zunächst ein Beispiel. Sei

$$r(x) = \frac{x^5 - x + 1}{(x^3 - x + 1)^2 (x^2 + x + 1)^3}$$

also

$$\begin{aligned} g(x) &= (x^3 - x + 1)^2 (x^2 + x + 1)^3, \\ g_1(x) &= gcd(g, g') = (x^3 - x + 1)\,(x^2 + x + 1)^2, \\ g_0(x) &= \frac{g}{g_1} = (x^3 - x + 1)\,(x^2 + x + 1) \end{aligned}$$

Formulieren wir die Aufgabenstellung in Reduce:

```
g:=(x^3-x+1)^2*(x^2+x+1)^3;
g1:=gcd(g,df(g,x));
g0:=g/g1;
f:=x^5-x+1;
f1:=for i:=0:deg(g1,x)-1 sum mkid(a,i)*x^i;
f0:=for i:=0:deg(g0,x)-1 sum mkid(b,i)*x^i;
```

Die beiden letzten Anweisungen erzeugen die Polynome f_1 und f_0 mit unbestimmten Koeffizienten

$$\begin{aligned} f_1 &= a_0 + a_1\,x + a_2\,x^2 + a_3\,x^3 + a_4\,x^4 + a_5\,x^5 + a_6\,x^6 \\ f_0 &= b_0 + b_1\,x + b_2\,x^2 + b_3\,x^3 + b_4\,x^4 \end{aligned}$$

Statt obige rationale Funktionen zu verwenden wollen wir gleich mit $g(x)$ durchmultiplizieren und die Differenz der beiden Seiten bilden. Zuvor ist jedoch noch auf die von Reduce nicht standardmäßig verwendete *gekürzte Normalform* für rationale Funktionen umzuschalten.

```
on gcd;
s:=f-g*(df(f1/g1,x)+f0/g0);
```

s ist tatsächlich polynomial. Koeffizientenvergleich liefert ein lineares Gleichungssystem in den unbestimmten Koeffizienten von f_1 und f_0

```
coeff(s,x);
s1:=solve ws;
```

$$\{\{a_0 = \tfrac{2295}{7889}, a_1 = \tfrac{10394}{23667}, a_2 = -\tfrac{470}{3381}, a_3 = -\tfrac{137}{3381}, a_4 = \tfrac{5548}{23667}, a_5 = \tfrac{4908}{7889}, a_6 = \tfrac{11143}{23667},$$
$$b_0 = \tfrac{20158}{23667}, b_1 = -\tfrac{19966}{23667}, b_2 = -\tfrac{1327}{7889}, b_3 = \tfrac{11143}{23667}, b_4 = 0\}\}$$

Setzen wir diese Lösung in f_0 und f_1 ein, so erhalten wir die gesuchte Zerlegung.

```
f1a:=sub(first s1,f1);
f0a:=sub(first s1,f0);
```

$$f_{1a} = \frac{11143\,x^6 + 14724\,x^5 + 5548\,x^4 - 959\,x^3 - 3290\,x^2 + 10394\,x + 6885}{23667}$$

$$f_{0a} = \frac{11143\,x^3 - 3981\,x^2 - 19966\,x + 20158}{23667}$$

Die Probe zeigt, daß wir bis hierher richtig gerechnet haben:

```
f/g-(df(f1a/g1,x)+f0a/g0);
```

$$0$$

Auf dieselbe Weise können wir auch im allgemeinen Fall vorgehen.

Satz 20 *Sei $r(x) = \frac{f(x)}{g(x)} \in k(x)$ eine rationale Funktion mit $deg(f) < n := deg(g)$ und $g_0, g_1 \in k[x]$ wie oben definiert. Dann gibt es eindeutig bestimmte Polynome $f_0, f_1 \in k[x]$ vom Grad $deg(f_0) < deg(g_0)$, $deg(f_1) < deg(g_1)$, so daß*

$$\int r(x)\,dx = \frac{f_1(x)}{g_1(x)} + \int \frac{f_0(x)}{g_0(x)}\,dx$$

gilt.

Die Koeffizienten der Polynome f_0, f_1 kann man durch Ansatz mit unbestimmten Koeffizienten aus einem eindeutig lösbaren linearen Gleichungssystem gewinnen.

BEWEIS : Wir betrachten die Gleichung

$$f(x) = \left(\frac{f_1(x)}{g_1(x)}\right)' g(x) + f_0(x)\,g_1(x),$$

die äquivalent zu der Gleichung ist, deren Existenz es zu beweisen gilt. Der gebrochen rationale erste Summand ist in Wirklichkeit auch polynomial. In der Tat, die Quotientenregel und $g = g_0\,g_1$ ergeben

$$\left(\frac{f_1}{g_1}\right)' g = f_1'\,g_0 - f_1\,g_2 \quad \text{mit} \quad g_2 := \frac{g_1'\,g_0}{g_1} \in k(x)$$

Wegen

$$g_1' = \sum_i (\nu_i - 1) g^{(i)}(x) \prod_i (x - \alpha_i)^{\nu_i - 2}$$

gilt sogar $g_2 \in k[x]$.

Offensichtlich ist weiterhin

$$f = f_1' \, g_0 - f_1 \, g_2 + f_0 \, g_1$$

Wir erhalten also bei einem Ansatz mit unbestimmten Koeffizienten wiederum ein Gleichungssystem mit n Gleichungen in $deg(g_0) + deg(g_1) = n$ Unbestimmten, das nach dem Satz von Newton-Leibniz-Bernoulli für jedes f wenigstens eine komplexe Lösung hat. Damit besitzt das entsprechende lineare Gleichungssystem eine eindeutige Lösung, die bereits in k^n liegt, da sie z.B. mit dem Gauß-Algorithmus berechnet werden kann. □

Beispiel: Berechnen wir den rationalen Anteil von

$$u := \int \frac{1}{(x^5 - x + 1)^3} \, dx.$$

Die entsprechenden Schritte sind dieselben wie oben

```
g:=(x^5-x+1)^3;
g1:=gcd(g,df(g,x));
g0:=g/g1;
f1:=for i:=0:deg(g1,x)-1 sum mkid(a,i)*x^i;
f0:=for i:=0:deg(g0,x)-1 sum mkid(b,i)*x^i;
on gcd;
s:=1-g*(df(f1/g1,x)+f0/g0);
coeff(s,x);
s1:=solve ws;
f1:=sub(first s1,f1);
f0:=sub(first s1,f0);
```

Als Endergebnis erhalten wir

$$f_1 = \frac{\left(\begin{array}{c} 6000000\,x^9 + 6122880\,x^8 + 5645300\,x^7 + 4187625\,x^6 - 10800000\,x^5 + \\ 795200\,x^4 + 1625180\,x^3 + 2892175\,x^2 + 10780750\,x - 5534464 \end{array} \right)}{16462322}$$

$$f_0 = \frac{3000000\,x^3 + 6122880\,x^2 + 8467950\,x + 8375250}{8231161}$$

und auch die Probe zeigt die Richtigkeit der bisherigen Rechnung

```
1/g-(df(f1a/g1,x)+f0a/g0);
```

so daß wir schlußfolgern können

$$u = \frac{f_1}{(x^5 - x + 1)^2} + \int \frac{f_0}{x^5 - x + 1} \, dx$$

6.2 Der transzendente Anteil der Stammfunktion

Für die weiteren Betrachtungen können wir den Nenner $g = g_0$ der zu integrierenden rationalen Funktion als quadratfrei voraussetzen. Sei wie oben $\{\alpha_1, \ldots, \alpha_d\}$ die Menge der Nullstellen von g und somit o.B.d.A. $g(x) = \prod_i (x - \alpha_i)$.

Wir wollen zunächst eine Darstellung für die Koeffizienten $c_i \in \mathbf{C}$ in der Formel

$$\int \frac{f(x)}{g(x)} \, dx = \sum_i c_i \, \ln(x - \alpha_i)$$

finden. Diese ist äquivalent zu

$$\frac{f(x)}{g(x)} = \sum_i \frac{c_i}{x - \alpha_i}$$

oder

$$f(x) = \sum_i c_i \, g^{(i)}(x).$$

Letztere Formel hat starke Ähnlichkeit zur Lagrange-Interpolationsformel, mit der ein Polynom $h(x)$ vom Grad $deg(h) < d$ konstruiert werden kann, das an vorgegebenen Stellen $\alpha_1, \ldots, \alpha_d$ vorgegebene Werte $b_1, \ldots, b_d$ annimmt:

$$h(x) = \sum_i b_i \prod_{j \neq i} \frac{x - \alpha_j}{\alpha_i - \alpha_j} = \sum_i b_i \frac{g^{(i)}(x)}{g^{(i)}(\alpha_i)}.$$

Ein Vergleich beider Formeln zeigt, daß

$$c_i = \frac{f(\alpha_i)}{g^{(i)}(\alpha_i)}$$

und wegen $g'(x) = \sum_i g^{(i)}(x)$ sowie $g^{(i)}(\alpha_j) = 0$ für $j \neq i$ schließlich

$$c_i = \frac{f(\alpha_i)}{g'(\alpha_i)}$$

gesetzt werden kann. Wir haben damit den folgenden Satz bewiesen:

Satz 21 *Ist $g(x)$ quadratfrei und $deg(f) < deg(g)$, so gilt*

$$\int \frac{f(x)}{g(x)} \, dx = \sum_\alpha \frac{f(\alpha)}{g'(\alpha)} \ln(x - \alpha),$$

wobei über $\alpha \in \text{RootOf}(g(x))$, die Menge der Nullstellen von g, summiert wird.

Korollar 3

$$\int \frac{1}{x^n - 1} \, dx = \sum_\alpha \frac{\alpha}{n} \ln(x - \alpha),$$

wobei über alle $\alpha \in \text{RootOf}(x^n - 1)$ zu summieren ist.

Dies ergibt sich sofort aus obiger Formel für $f = 1$ und $g = x^n - 1$ unter Beachtung, daß $\alpha^n = 1$ gilt.

Wir haben im letzten Kapitel gesehen, daß man den Quotienten zweier algebraischer Zahlen, hier also $c_i = \frac{f(\alpha)}{g'(\alpha)}$, stets als Linearkombination von Elementen aus T_{red}, hier also von Potenzen $1, \alpha^1, \ldots, \alpha^{d-1}$ darstellen kann. Eine solche polynomiale Darstellung von c_i kann man explizit berechnen, ohne dabei bereits neue algebraische Zahlen einzuführen:

Da g nur einfache Nullstellen besitzt, sind g und g' teilerfremd. Wir finden (etwa mit dem erweiterten Euklidschen Algorithmus) also zwei Polynome $u(x), v(x) \in k[x]$ mit

$$1 = u(x) \, g(x) + v(x) \, g'(x).$$

Da α eine Nullstelle von g ist, gilt mithin $g'(\alpha) \, v(\alpha) = 1$. Als Kofaktor c_i in obiger Integrationsformel kann man also auch den Wert des Polynoms $f(x) \, v(x)$ an der Stelle α einsetzen. Wegen $g(\alpha) = 0$ kann man dieses Polynom sogar noch $(mod \; g(x))$ reduzieren:

Satz 22 *Ist $g(x)$ quadratfrei und $deg(f) < deg(g)$, so existiert ein (effektiv berechenbares) Polynom $c(x) \in k[x]$ vom Grad $deg(c) < deg(g)$, für welches*

$$\int \frac{f(x)}{g(x)}\,dx = \sum_{\alpha} c(\alpha)\ln(x-\alpha)$$

gilt, wobei wieder über $\alpha \in \text{RootOf}(g(x))$ zu summieren ist.

Das Polynom $c(x)$ kann man wiederum über einen Ansatz mit unbestimmten Koeffizienten und Koeffizientenvergleich aus einem linearen Gleichungssystem gewinnen. Erfahrungsgemäß sind die Koeffizienten, die in einem solchen Polynom auftreten, wesentlich größer als die in der rationalen Darstellung. Betrachten wir etwa das Beispiel

$$\int \frac{1}{(x^5-x+1)}\,dx = \sum_{\alpha} \frac{1}{5\alpha^4-1}\ln(x-\alpha),$$

wobei über $\alpha \in \text{RootOf}(x^5-x+1)$ summiert wird. Das entsprechende Polynom $c(x)$ können wir aus $1 - c(x)\,g'(x) \equiv 0 \ (mod\ g(x))$, wobei $c(x)$ mit unbestimmten Koeffizienten anzusetzen ist, gewinnen. Mit Reduce erhalten wir

```
g:=(x^5-x+1);
c:=for i:=0:deg(g,x)-1 sum mkid(c,i)*x^i;
s:=remainder(1-c*df(g,x),g);
coeff(s,x);
s1:=solve ws;
c:=sub(first s1,c);
```

$$c(x) = \frac{-320\,x^4 - 400\,x^3 - 500\,x^2 - 625\,x + 256}{2869},$$

also

$$\int \frac{1}{(x^5-x+1)}\,dx = \sum_{\alpha} \frac{-320\,\alpha^4 - 400\,\alpha^3 - 500\,\alpha^2 - 625\,\alpha + 256}{2869}\ln(x-\alpha)$$

Die Darstellung des transzendenten Teils in der bisherigen Form hat allerdings immer noch den Nachteil, daß er etwa im Fall $f(x) = g'(x)$ algebraische Zahlen einführt, während zur Darstellung des Ergebnisses solche nicht notwendig sind. Der Satz liefert für $g(x) = x^5 - x + 1$

$$\int \frac{5x^4-1}{(x^5-x+1)}\,dx = \sum_{\alpha\,:\,g(\alpha)=0} \ln(x-\alpha) = \ln(x^5-x+1),$$

wenn man nach den Logarithmengesetzen die Summe von Logarithmen in den Logarithmus eines Produkts verwandelt. Ähnlich kann man vorgehen, wenn einige der Kofaktoren $c(\alpha)$ übereinstimmen, wobei die Hoffnung berechtigt ist, daß dabei einfachere algebraische Zahlen entstehen.

Da es sich bei den $\beta = c(\alpha) = \frac{f(\alpha)}{g'(\alpha)}$ wiederum um algebraische Zahlen handelt, wollen wir zunächst die Frage nach deren Minimalpolynom untersuchen. Offensichtlich ist ein solches Paar (α, β) eine Lösung des Gleichungssystems $\{g(x) = 0, f(x) = y \cdot g'(x)\}$ und umgekehrt, so daß wir nach einem Polynom $p(y)$ suchen können, das alle y-Komponenten von derartigen Lösungspaaren erfüllen. Das Minimalpolynom von β ist dann ein Faktor dieses Polynoms.

Die Berechnung eines solchen Polynoms führt auf die Elimination der Variablen x aus obigem Gleichungssystem. Diese Eliminationsaufgabe kann man z.B. über die Berechnung der *Resultante* lösen (wobei es genügt, deren quadratfreien Teil zu nehmen):

$$p(y) = \text{res}_x(g(x), f(x) - y \cdot g'(x)).$$

Für jede Nullstelle β von $p(y)$ haben die beiden Polynome $g(x)$ und $f(x) - \beta \cdot g'(x)$ die gemeinsamen Nullstellen

$$\mathcal{R}_\beta := \{\alpha \ : \ g(\alpha) = 0 \ \wedge \ \frac{f(\alpha)}{g'(\alpha)} = \beta\}.$$

Offensichtlich gilt

$$\prod_{\alpha \in \mathcal{R}_\beta} (x - \alpha) = \gcd\left(g(x), f(x) - \beta \cdot g'(x)\right),$$

so daß wir insgesamt folgende dritte Darstellung für den transzendenten Anteil des Integrals einer rationalen Funktion erhalten, siehe [1, Thm 2.4.1.]

Satz 23 (Rothstein-Trager) *Ist $g(x)$ quadratfrei, $deg(f) < deg(g)$, $\gcd(f,g) = 1$ und $p(y)$ der quadratfreie Teil der Resultante $\mathrm{res}_x(g(x), f(x) - y \cdot g'(x))$, so gilt*

$$\int \frac{f(x)}{g(x)}\,dx = \sum_\beta \beta \cdot \ln\bigl(\gcd\left(g(x), f(x) - \beta \cdot g'(x)\right)\bigr),$$

wobei über die Nullstellen $\beta \in \textsc{RootOf}(p(y))$ zu summieren ist.

Im Fall $f(x) = g'(x)$ gilt $p(y) = (y - 1)$, also $\beta = 1$ und wir erhalten nun unmittelbar

$$\int \frac{g'(x)}{g(x)}\,dx = \ln(g(x)).$$

Doch auch für andere Integrale können sich deutliche Vereinfachungen ergeben wie etwa im folgenden Beispiel [1, 2.4.1.]:

Gesucht ist die Stammfunktion von

$$r(x) = \frac{x^4 - 3x^2 + 6}{x^6 - 5x^4 + 5x^2 + 4}.$$

Der Nenner $g(x) = x^6 - 5x^4 + 5x^2 + 4$ ist quadratfrei und irreduzibel, so daß nach Satz 21 die Antwort algebraische Zahlen vom Grad 6 enthalten würde. Die Berechnung der Resultante $p(y)$ liefert

```
f:=x^4-3x^2+6;
g:=x^6-5x^4+5x^2+4;
p:=resultant(g,f-y*df(g,x),x);
```

$$p(y) = 45796\,(4y^2 + 1)^3$$

und damit $\beta = \pm\frac{i}{2}$, also nur eine quadratische Irrationalität. Von den 6 Summanden, die nach Satz 21 entstehen würden, haben je 3 denselben Kofaktor und können deshalb zusammengefaßt werden. Die Berechnung der größten gemeinsamen Teiler über der entsprechenden algebraischen Erweiterung liefert

```
on complex;
gcd(g,f-i/2*df(g,x));
gcd(g,f+i/2*df(g,x));
```

$$x^3 + i\,x^2 - 3\,x - 2\,i$$

$$x^3 - i\,x^2 - 3\,x + 2\,i$$

und schließlich

$$\int r(x)\,dx = \frac{i}{2}\ln(x^3 + i\,x^2 - 3\,x - 2\,i) - \frac{i}{2}\ln(x^3 - i\,x^2 - 3\,x + 2\,i)$$

Wir sehen an diesem Beispiel noch einmal deutlich, welche Rolle algebraische Zahlen im symbolischen Rechnen spielen, wie schwierig der Umgang mit ihnen stellenweise ist und welche Methoden angewendet werden können, um den Einsatz algebraischer Zahlen auf das notwendige Minimun zu beschränken.

Literaturverzeichnis

[1] M. Bronstein. *Symbolic integration I — transcendental functions.* Algorithms and computations in mathematics I. Springer, Berlin, 1997.

[2] B. Buchberger, G.E. Collins, and R. Loos, editors. *Computer Algebra, Symbolic and Algebraic Computation.* Springer, Wien - New York, second edition, 1983.

[3] B. Buchberger and R. Loos. *Computer algebra*, pages 11–43. In Buchberger et al. [2], second edition, 1983.

[4] *Duden Informatik.* Dudenverlag, Mannheim, 1993.

[5] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra.* Kluwer Acad. Publisher, 2 edition, 1992.

[6] J. Grabmeier. Computeralgebra – eine Säule des Wissenschaftlichen Rechnens. *it + ti*, 6:5 – 20, 1995.

[7] J. Grabmeier and V. Weispfenning, editors. *Computeralgebra in Deutschland – Bestandsaufnahme, Möglichkeiten, Perspektiven.* Fachgruppe der GI, DMV, GAMM, Passau and Heidelberg, 1993.

[8] D. Gruntz and M. Monagan. Introduction to Gauss. *Maple Technical Newsletter*, 9, 1993.

[9] A. Heck. *Introduction to Maple.* Springer, 1993.

[10] D.E. Knuth. *The art of computer programming*, volume 1. Addison Wesley, 1991.

[11] F. Mušmann. Die entzauberte Moderne. Zur Neudefinition des Verhältnisses von Wissenschaft und Gesellschaft. *Forum Wissenschaft*, 14(3), 1996. Dossier.

[12] T. Ottmann and P. Widmeyer. *Algorithmen und Datenstrukturen.* BI Wissenschaftsverlag, 2 edition, 1993.

[13] R. Pavelle, M. Rothstein, and J.P. Fitch. Computer algebra. *Scientific American*, 245(6):102 – 113, dec 1981.

[14] H. Pieper. *Die komplexen Zahlen.* Dt. Verlag der Wissenschaften, Berlin, 1988.

[15] U. Schwardmann. *Computeralgebrasysteme.* Addison-Wesley, 1995.

[16] B. Simon. Comparative CAS review. *Notices AMS*, 39:700 – 710, sept 1992.

[17] J. Weizenbaum. *Die Macht der Computer und die Ohnmacht der Vernunft*, volume 274 of *Taschenbuch Wissenschaft.* Suhrkamp, 9 edition, 1994.

[18] M. Wester. A review of CAS mathematical capabilities. *CAN Nieuwsbrief*, 13:41–48, dec 1994.

[19] N. Wirth. *Algorithmen und Datenstrukturen.* B.G. Teubner Verlag Stuttgart, 4 edition, 1986.