Skript zum Kurs
Einführung in das symbolische Rechnen
Wintersemester 2017/18

H.-G. Gräbe, Institut für Informatik
http://bis.informatik.uni-leipzig.de/HansGertGraebe

26. Januar 2018

Inhaltsverzeichnis

Einleitung

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 in algorithmisch aufbereiteter Form über eine einheitliche Schnittstelle zur Verfügung zu stellen, eine zunehmend wichtige Rolle. Solche Systeme, die im Gegensatz zu klassischen Anwendungen des Computers auch symbolische Kalküle beherrschen, werden zugleich in absehbarer Zeit wenigstens im naturwissenschaftlich-technischen Bereich den Kern umfassenderer Wissensrepräsentationssysteme bilden. Vorreiter der Entwicklung derartiger integrierter Systeme wissenschaftlich-technischen Wissens ist Wolfram Research, Inc. mit MATHEMATICA und seiner Plattform *Wolfram Alpha*[1]. Der souveräne Umgang mit solchen Systemen gehört damit zu einer der Grundfertigkeiten, die von Hochschul-Absolventen in Zukunft erwartet werden. Grund genug, erste Kontakte mit derartigen Werkzeugen bereits im Schul-Curriculum zu verankern (wie mit den sächsischen Lehrplänen ab 2008 für das Gymnasium geschehen) und sie im universitären Studium als wichtige Hilfsmittel einzusetzen. Eine solche gewisse Vertrautheit im alltäglichen Umgang mit CAS werde ich in diesem Kurs voraussetzen.

Wie für das Programmieren in klassischen Programmiersprachen ist auch beim CAS-Einsatz 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 Schüler bereits frühzeitig eigene Erfahrungen im Umgang mit diesem *Werkzeug geistiger Arbeit* etwa im Fachunterricht sammeln, wird auf dessen *systematische* Einführung ab Klasse 8 (sächsischer Lehrplan) nicht verzichtet. Schließlich gehört der qualifizierte Umgang mit dem Taschenrechner – insbesondere die Kenntnis seiner Eigenarten, Möglichkeiten und Grenzen – zu den elementaren Kompetenzen, über welche jeder Schüler mit Verlassen der Schule verfügen 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.

Ziel dieses Kurses ist es also nicht, Erfahrungen im Umgang mit einem der großen CAS zu vermitteln, sondern die Möglichkeiten, Grenzen, Formen und Methoden des Einsatzes von Computern zur Ausführung symbolischer Rechnungen systematisch darzustellen. Im Gegensatz zur einschlägigen Literatur soll dabei nicht eines der großen Systeme im Mittelpunkt stehen, sondern deren Gesamtheit Berücksichtigung finden. Eine zentrale Rolle werden die beiden großen Systeme MATHEMATICA und MAPLE sowie darüber hinaus die freien System MAXIMA, REDUCE und SAGE spielen.

Ähnlich wie sich übergreifende Aspekte von Programmiersprachen nur aus deren vergleichender Betrachtung erschließen, ist ein solches Herangehen besser geeignet, die grundlegenden Techniken und Begriffe, die mit der Anwendung des Computers für symbolische Rechnungen verbunden sind, abzuheben. Ein solcher Zugang erscheint auch deshalb sowohl gerechtfertigt als auch wünschenswert, weil es für jedes der großen Systeme eine Fülle von einführender Literatur höchst

[1] `http://www.wolframalpha.com/about.html`

unterschiedlicher Qualität und verschiedenen Anspruchsniveaus gibt. Mehr noch, die Entwicklerteams der großen Systeme verwenden viel Energie darauf, ihre Produkte auch von der äußeren Gestalt her nach modernen softwaretechnischen und -ergonomischen Gesichtspunkten aufzubereiten[2], so dass selbst ein ungeübter Nutzer in der Lage sein sollte, sich mit wenig Mühe 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 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 *mathematische Rigorosität* mit sehr unterschiedlichem Aufwand im Nachhinein integrieren lässt. Derartige Feinheiten wurden – nach einer Reihe spektakulärer Fehlberechnungen, deren genauere Analyse auf *Ungenauigkeiten mathematischer Semantik* tief im Kern einiger Algorithmen als Fehlerquelle führte – in den 1990er Jahren intensiv untersucht und sind in einem Teil der Wester-Liste [24, ch. 3] oder in Bernardins Übersicht [24, ch. 7] enthalten. Die Bemühungen um die Integration dieser semantischen Feinheiten in die verschiedenen Architekturen bestehender Systeme führte zu unterschiedlichem Verhalten bis hin zu sehr unterschiedlichem Antwortverhalten der verschiedenen CAS. Solche Unterschiede im Detail zu studieren übersteigt die zeitlichen Möglichkeiten und auch die Intention dieses Kurses. Ich werde mich auf die Frage beschränken, wie konsistent die einzelnen Systeme ihre eigenen Konzepte verfolgen, da gerade Übersichten und Problemberichte wie die zitierten die Systementwickler manchmal dazu verleiten, bisher nicht beachtete Problemstellungen als *quick hack* und damit nur halbherzig in ihre Systeme zu integrieren.

Diese Konsistenzfrage steht als generelle Einsatzvoraussetzung für den Nutzer eines Computeralgebrasystems an zentraler Stelle. Leider ist sie nicht eindeutig zu beantworten, so lange *ein* Werkzeug *verschiedene* Nutzergruppen adressiert. Aber selbst die Einführung global einstellbarer *Nutzerprofile*, wie in [24, ch. 3] vorgeschlagen, würde das Problem kaum lösen. Es ist deshalb 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 bereits im Voraus in groben Zügen im Klaren zu sein[3].

Allerdings begegnet man diesem *Zauberlehrlingseffekt*, den Weizenbaum in seinem klassischen Werk „Die Macht der Computer und die Ohnmacht der Vernunft“ [22] 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*, geworden, dessen deutsche Übertragung *Technologiefolgenabschätzung* die

[2] Dies ist, nach dem Vorreiter MATHEMATICA, mit einer stärkeren Kommerzialisierung auch der anderen großen Systeme verbunden. Eine wichtige Erkenntnis scheint mir dabei zu sein, dass sich im Gegensatz zur unmittelbaren Forschung an Algorithmen hierfür keine öffentlichen Mittel allokieren lassen. Auch scheinen die subtilen Mechanismen, die zum Erfolg freier Softwareprojekte führen, hier nur langsam zu greifen, da das Verhältnis zwischen Größe der Nutzergemeinde und erforderlichem Programmieraufwand deutlich ungünstiger ausfällt. Andererseits stehen die geforderten Lizenzpreise gerade der Studentenversionen in keinem Verhältnis zur Leistungsfähigkeit der Systeme. Mit diesen Mitteln kann man im Wesentlichen wohl nur 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 erscheint mir auch deshalb wünschenswert, weil ähnliche Entwicklungen in anderen Bereichen der Informatik ebenfalls stattfinden.

[3] Graf schreibt dazu in [8, S. 123] über sein MATHEMATICA-Paket: *The Package can do valuable and very helpful things but also stupid things, as for example computing a result that does not exist. In general it cannot decide whether a certain computation makes sense or not, hence the user should know what he is doing.*

zu thematisierenden Inhalte nur unvollkommen wiedergibt. Eine solche kritische Distanz zu unserem immer stärker technologisch geprägten kulturellen Umfeld – jenseits der beiden Extreme *Technikgläubigkeit* und *Technikverdammung* – ist auch auf individueller Ebene erforderlich, um kollektiv die Gefahr zu bannen, vom Räderwerk dieser Maschinerie zerquetscht zu werden. Bezogen auf den Computer kann die Beschäftigung mit Computeralgebra auch hier wertvolle Einsichten vermitteln und helfen, ein am Werkzeugcharakter orientiertes kritisches Verhältnis zum Computereinsatz gegenüber oft anzutreffender Fetischisierung (wieder) mehr in den Vordergrund zu rücken.

Die Voraussetzungen

Symbolische Rechnungen sind das wohl grundlegendste methodische Handwerkszeug in der Mathematik und durchdringen alle ihre Gebiete und Anwendungen. Die Spanne symbolischer Kalküle reicht dabei von allgemein bekannten Anwendungen wie Termvereinfachung, Faktorisierung, Differential- und Integralrechnung über mächtige mathematische Kalküle mit weit verbreitetem Einsatzgebiet (etwa Analysis spezieller Funktionen, Differentialgleichungen) bis hin zu Kalkülen einzelner Fachgebiete (Gruppentheorie, Tensorrechnung, Differentialgeometrie), die vor allem für Spezialisten und Anwender der entsprechenden Fachgebiete interessant sind.

Für jeden dieser Kalküle gilt, dass dessen qualifizierter Einsatz den mathematisch entsprechend qualifizierten Nutzer voraussetzt. Moderne CAS bündeln in diesem Sinne einen großen Teil des heute algorithmisch verfügbaren mathematischen Wissens und sind damit als Universalwerkzeuge geistiger Arbeit ein zentraler Baustein einer sich herausbildenden *Wissensgesellschaft*. Diese Entwicklungen stehen heute, im Zeitalter von Vernetzung, wachsender Bedeutung semantischer Techniken und *Big Data*, erst am Anfang – eine Plattform wie *Wolfram Alpha*[4] lässt die dort schlummernden Potenzen integrierter Expertise allerdings bereits erahnen.

Dabei stellt sich heraus, dass zur Implementierung und Nutzung der Vielfalt unterschiedlicher Kalküle ein kleines, wiederkehrendes programmiertechnisches 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.

Für einen Einführungskurs ergibt sich aus diesen Überlegungen eine doppelte Schwierigkeit, da einerseits die zu demonstrierenden Prinzipien erst in nichttrivialen Anwendungen zu überzeugender Entfaltung kommen, andererseits solche Anwendungen ohne Kenntnis ihres Kontextes nur schwer nachzuvollziehen sind. Dies verlangt eine wohlüberlegte Auswahl zu treffen.

Im Folgenden werden wir uns deshalb nach einer allgemeinen Einführung in Probleme, Rahmenbedingungen, Stellung und Geschichte des symbolischen Rechnens zunächst auf die Darstellung wichtiger Designaspekte eines CAS der zweiten Generation sowie der verwendeten Datenorganisations- und Sprachkonzepte konzentrieren. Daran schließt eine genauere Betrachtung theoretischer und praktischer Aspekte der Simplifikationsproblematik als besonderem Charakteristikum des symbolischen Rechnens an. Schließlich erleben wir am Beispiel der symbolischen Behandlung algebraischer Zahlen diese Konzepte in ihrem komplexen Zusammenspiel, stellen Anforderungen und Lösungen gegenüber und lernen dabei einige auf den ersten Blick merkwürdige Ansätze besser verstehen.

In einem Addendum des Skripts sind weitere Ausführungen zu algebraischen Zahlen, Nullstellenberechnungens sowie zum Unterschied der mathematischen und computeralgebraischen Behandlung des Integrierens rationaler Funktionen zu finden. Weiter werden dort übergreifende Aspekte eher philosophischer Natur diskutiert, um die Stellung des symbolischen Rechnens als einer zentralen technologischen Entwicklungslinie im Wissenschaftsgebäude besser zu verstehen.

Nicht berühren werden wir das für viele praktische Anwendungen in den Natur- und Ingenieurwissenschaften zentrale Feld der *Differentialgleichungen*, da die Schwierigkeiten der damit verbundenen Mathematik in keinem Verhältnis zu den gewinnbaren neuen Einsichten in grundlegen-

[4] `http://www.wolframalpha.com/about.html`

de Zusammenhänge des symbolischen Rechnens im Umfang der Konzepte dieses Kurses stehen. Gleichfalls ausgeklammert bleiben Fragen der graphischen Möglichkeiten der CAS, die ein wichtiges Mittel der Visualisierung wissenschaftlicher Zusammenhänge sind und für viele Nutzer den eigentlichen Reiz eines großen CAS darstellen, aber nicht zum engeren Gebiet des symbolischen Rechnens gehören. Dasselbe gilt für die mittlerweile ausgezeichneten Präsentationsmöglichkeiten der integrierten graphischen Oberflächen der meisten der betrachteten Systeme zur Organisation und Darstellung technischer Texte.

Kapitel 1

Computeralgebrasysteme im Einsatz

In diesem Kapitel wollen wir Computeralgebrasysteme in verschiedenen Situationen als Rechenhilfsmittel kennen lernen und dabei erste Besonderheiten eines solchen Systems gegenüber rein numerisch basierten Rechenhilfsmitteln heraus arbeiten. Die folgenden Rechnungen basieren auf MATHEMATICA 11.0, hätten aber mit jedem der großen CAS in ähnlicher Weise ausgeführt werden können. An einigen Stellen sind entsprechende Rechnungen zum Vergleich auch mit anderen CAS ausgeführt.

1.1 Computeralgebrasysteme als Taschenrechner für Zahlen

Mit einem CAS kann man natürlich zunächst alle Rechnungen ausführen, die ein klassischer Taschenrechner beherrscht. Das umfasst neben den Grundrechenarten auch eine Reihe mathematischer Funktionen.

```
1.23+2.25
```

3.48

```
12+23
```

35

```
10!
```

3628800

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

```
100!
```

```
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
```

Die (im Kernbereich ausgeführten) Rechnungen eines CAS sind grundsätzlich exakt.

Dies wird auch bei der Behandlung rationaler Zahlen sowie irrationaler Zahlen deutlich.

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

$$7/6$$

Diese werden nicht durch numerische Näherungswerte ersetzt, sondern bleiben als *symbolische Ausdrücke* in einer Form stehen, die uns aus einem streng mathematischen Kalkül wohlbekannt ist. Im Gegensatz zu numerischen Approximationen, bei denen sich die Zahl `3.1415926535` qualitativ kaum von der Zahl `3.1426968052` unterscheidet (letzteres ist $\frac{20}{9} \cdot \sqrt{2}$, auf 10 Stellen nach dem Komma genau ausgerechnet), verbirgt sich hinter einem solchen Symbol eine wohldefinierte *mathematische Semantik*.
So ist etwa „$\sqrt{2}$ diejenige (eindeutig bestimmte) positive reelle Zahl, deren Quadrat gleich 2 ist“.

```
Sum[1/i,{i,1,50}]
```

$$\frac{13943237577224054960759}{3099044504245996706400}$$

```
% // N
```

$$4.499205338$$

```
Sqrt[2]
```

$$\sqrt{2}$$

```
Pi
```

$$\pi$$

Ein CAS verfügt über Mittel, diese Semantik (bis zu einem gewissen Grade) darzustellen. So „weiß“ MATHEMATICA einiges über die Zahl π.

```
Sin[Pi]
```

$$0$$

```
Sin[Pi/4]
```

$$\frac{1}{\sqrt{2}}$$

```
Sin[Pi/5]
```

$$\sqrt{\frac{5}{8} - \frac{\sqrt{5}}{8}}$$

Ein Programm, das nur einen (noch so guten) Näherungswert von π kennt, kann die Information $\sin(\pi) = 0$ prinzipiell nicht *exakt* ableiten.

```
p=N[Pi]; {p, Sin[p]}
```

$$\left\{3.14159, 1.22465 \cdot 10^{-16}\right\}$$

Dieselben Rechnungen mit MAXIMA. Beachten Sie die Möglichkeit, eine Rechnung in einem speziellen (Komma separiert angegebenen) *Kontext* auszuführen. Beachten Sie, dass allein die Zuweisung an p mit der angegebenen Präzision erfolgt, nicht aber die Berechnung von `sin(p)`. Dafür hätte `fpprec:20` global gesetzt werden müssen.

```
p:bfloat(%pi), fpprec:20;
sin(p);
```

$$3.14159265358979323850$$
$$1.144237745221966\text{b-}17$$

Ähnlich der Ansatz von SAGE, der sich aber stärker an der objektorientierten Notation der Hostsprache `Python` orientiert und die Genauigkeitsinformation auf dem Objekt p speichert.

```
p = pi.n(digits=100)
sin(p)
```

$$3.141592\ldots2117068$$
$$2.86889\ldots846969\text{e-}102$$

Ohne hier auf Details der inneren Darstellung einzugehen, halten wir fest, dass CAS symbolischen Ausdrücken *Eigenschaften* zuordnen, die deren mathematischen Gehalt widerspiegeln.

Wie eben gesehen, kann eine dieser Eigenschaften insbesondere darin bestehen, dass dem CAS auch ein Verfahren zur Bestimmung eines *numerischen Näherungswerts* bekannt ist. Dieser kann mit einer beliebig vorgegebenen Präzision berechnet werden, die softwaremäßig auf der Basis der Langzahlarithmetik operiert. Damit ist die Numerik zwar oft deutlich langsamer als die auf Hardware-Operationen zurückgeführte Numerik klassischer Programmiersprachen, erlaubt aber genauere und insbesondere adaptive Approximationen.

Ähnlich wie auf dem Taschenrechner stehen auch wichtige mathematische Funktionen und Operationen zur Verfügung, allerdings in einem wesentlich größeren Umfang als dort. So wissen CAS wie MAXIMA, 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!)
```

$$2^{10}\,3^5\,5^2\,7\,11$$

```
primep(12!-1)
```

$$True$$

Damit lassen sich bereits umfangreichere rechnerische Experimente aufsetzen. So kann etwa genauer untersucht werden, welche Regelmäßigkeiten in den Ausdrücken $g(n) = \gcd(2^{3\,n}-1, 3^{2\,n}-1)$ für verschiedene n zu finden sind. Wir speichern dazu zunächst eine Reihe von Paaren $(n, g(n))$ in einer Liste l.

```
l:makelist([n,gcd(2^(3*n)-1,3^(2*n)-1)],n,1,30);
```

$$[[1,1],[2,1],[3,7],[4,5],[5,1],[6,511],[7,1],[8,85],[9,7],[10,11],[11,23],[12,33215],[13,1],$$
$$[14,1],[15,217],[16,85],[17,103],[18,9709],[19,1],[20,687775],[21,49],[22,1541],[23,47],$$
$$[24,564655],[25,151],[26,1],[27,7],[28,145],[29,1],[30,174251]]$$

Nun können wir zum Beispiel prüfen, für welche n der Wert $g(n)$ eine Primzahl ist

```
sublist(l,lambda([u],primep(part(u,2))));
```

$$[[3,7],[4,5],[9,7],[10,11],[11,23],[17,103],[23,47],[25,151],[27,7]]$$

oder die Werte $g(n)$ in Faktoren zerlegen

```
map(lambda([u],[part(u,1),factor(part(u,2))]),l);
```

$$[[1,1],[2,1],[3,7],[4,5],[5,1],[6,7\cdot 73],[7,1],[8,5\cdot 17],[9,7],[10,11],\ldots]$$

1.2 Computeralgebrasysteme als Taschenrechner für Formeln und symbolische Ausdrücke

Die wichtigste Besonderheit eines Computeralgebrasystems gegenüber Taschenrechner und klassischen Programmiersprachen ist ihre

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 vier 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))
```

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

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 ermöglichen die Funktionen `Simplify` oder `Together`. Das Ergebnis mag verblüffen; jedenfalls sieht man das dem ursprünglichen Ausdruck nicht ohne weiteres an.

```
Simplify[u]
```

$$0$$

Eine solche normalisierte Darstellung ist nicht immer zweckmäßig, weshalb diese Umformung nicht automatisch vorgenommen wurde. So erhalten wir etwa

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

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

```
Together[u]
```

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

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.

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

Wir verwenden dazu die Substitutionsfunktion `ReplaceAll`, die lokal eine Variable durch einen anderen Ausdruck, in diesem Fall eine Zahl, ersetzt.

```
Simplify[ReplaceAll[u,n->2]]
```

$$1$$

Wie viele zweistellige MATHEMATICA-Funktionen existiert für `ReplaceAll` auch eine Infix-Notation `A /. B` , die wie folgt verwendet werden kann

```
Simplify[u /. n → 3]
```

$$a + b + c$$

```
Simplify[u /. n → 4]
```

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

```
Simplify[u /. n → 5]
```

$$a^3 + b^3 + b^2\,c + b\,c^2 + c^3 + a^2\,(b + c) + a\left(b^2 + b\,c + c^2\right)$$

Wir sehen, dass sich in jedem der betrachteten Fälle die rationale Funktion u_n zu einem Polynom vereinfacht. Alternativ hätten wir u auch als Funktion $f[n]$ vereinbaren können:

```
f[n_Integer] := a^n/((a-b)*(a-c)) + b^n/((b-c)*(b-a)) + c^n/((c-a)*(c-b));
Table[Simplify[f[n]],{n,1,5}]
```

$$\{0,1, a + b + c, a^2 + b^2 + b\,c + c^2 + a\,(b + c),\\ a^3 + b^3 + b^2\,c + b\,c^2 + c^3 + a^2\,(b + c) + a\,(b^2 + b\,c + c^2)\}$$

Ähnlich kann man auch in anderen CAS vorgehen, etwa in MAXIMA. Beachten Sie, dass hierbei `:` der Zuweisungsoperator ist und MAXIMA-Befehle grundsätzlich mit `;` abzuschließen sind, während `;` am Ende eines MATHEMATICA-Kommandos die Ergebnisausgabe unterdrückt[1]. In MAXIMA kann die Ausgabe mit `$` unterdrückt werden.

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

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

[1] Genauer: `;` ist in MAXIMA ein *Terminationssymbol*, in MATHEMATICA dagegen ein *Separator* – MATHEMATICA gibt immer das Ergebnis des letzten Kommandos aus, das in diesem Fall die leere Anweisung ist.

```
ratsimp(ev(u,n=4));
```

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

Um dieselben Rechnungen mit SAGE auszuführen ist etwas mehr Aufwand erforderlich. Zunächst müssen die symbolischen Variablen explizit deklariert werden. Dann lassen sich die zu untersuchenden symbolischen Ausdrücke wie eben schon eingeben. Eine von n abhängende Schar von Ausdrücken kann entweder wie oben mit einem symbolischen Wert n angeschrieben werden, der später durch konkrete Werte ersetzt wird, oder aber als Funktion mit Parameter n. Im Folgenden wird diese zweite Notationsmöglichkeit (die auch in jedem anderen CAS möglich ist) verwendet.

```
a,b,c,n = var('a b c n')
u(n) = a^n/((a-b)*(a-c))+b^n/((b-a)*(b-c))+c^n/((c-a)*(c-b))
u(3).simplify_rational()
```

$$a + b + c$$

`simplify_rational` ist eine spezielle Methode von `sage.symbolic.expression.Expression`, zu dem sowohl u als auch $u(3)$ gehört, wie `type(u(3))` herausbringt. Auch in SAGE können mehrere Ergebnisse in einer Liste aggregiert werden.

```
[u(i).simplify_rational() for i in (2..4)]
```

$$\left[1, a + b + c, (a+b)\,c + a^2 + a\,b + b^2 + c^2\right]$$

Wir sehen an diesem Beispiel, dass Polynome wie im unterliegenden CAS MAXIMA, das von SAGE aufgerufen wird, in rekursiver Form gespeichert werden. Mit `expand` kann das Ergebnis in die distributive Form umgewandelt werden.

```
[u(i).simplify_rational().expand() for i in (2..4)]
```

$$\left[1, a + b + c, a^2 + a\,b + a\,c + b^2 + b\,c + c^2\right]$$

Tragen die in die Formeln eingehenden Symbole weitere semantische Information, so sind auch komplexere Vereinfachungen möglich. So werden etwa Ausdrücke, die die imaginäre Einheit I enthalten, von MATHEMATICA wie erwartet vereinfacht.

```
Table[(1+I)^n, {n,1,10} ]
```

$$1+\mathrm{i},\ 2\,\mathrm{i},\ -2+2\,\mathrm{i},\ -4,\ -4-4\,\mathrm{i},\ -8\,\mathrm{i},\ 8-8\,\mathrm{i},\ 16,\ 16+16\,\mathrm{i},\ 32\,\mathrm{i}$$

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

$$1+\mathrm{i}$$

In MAXIMA dagegen müssen die entsprechenden Umformungen explizit angestoßen werden.

```
l1:makelist((1+%i)^n,n,1,10); map(expand,l1);
```

```
(3+%i)/(2-%i); rectform(%);
```

Das ist für etwas kompliziertere Aufgaben auch in anderen CAS erforderlich wie in den folgenden Rechnungen mit MATHEMATICA:

```
Table[(Sqrt[2]+Sqrt[3])^n,{n,2,4}]
```

$$\left\{\left(\sqrt{2}+\sqrt{3}\right)^2,\ \left(\sqrt{2}+\sqrt{3}\right)^3,\ \left(\sqrt{2}+\sqrt{3}\right)^4\right\}$$

```
Expand[Table[(Sqrt[2]+Sqrt[3])^n,{n,2,5}]]
```

$$\left\{5+2\sqrt{6},\ 11\sqrt{2}+9\sqrt{3},\ 49+20\sqrt{6},\ 109\sqrt{2}+89\sqrt{3}\right\}$$

AXIOM verfolgt stärker ein Konzept der Darstellung der Ergebnisse in Normalformen und führt alle diese Umformungen automatisch aus.

Manchmal ist es nicht einfach, das System zu „überreden“, genau das zu tun, was man will. MAXIMA liefert auf obiger Eingabe zum Beispiel mit `expand` kein vollständig zusammengefasstes Ergebnis, sondern erst mit `ratsimp` – und das auch in einer eigenartig Mischung von Potenz- und `sqrt`-Notation für die beteiligten Wurzelausdrücke.

```
l:makelist((sqrt(2)+sqrt(3))^n,n,2,5);
```

```
map(expand,l);
```

$$\left[2^{\frac{3}{2}}\sqrt{3}+5, 3^{\frac{5}{2}}+2^{\frac{3}{2}}+9\sqrt{2}, 2^{\frac{5}{2}}3^{\frac{3}{2}}+2^{\frac{7}{2}}\sqrt{3}+49, 3^{\frac{5}{2}}+20\,3^{\frac{3}{2}}+20\sqrt{3}+2^{\frac{13}{2}}+45\sqrt{2}\right]$$

```
map(ratsimp,l);
```

$$\left[2^{\frac{3}{2}}\sqrt{3}+5, 3^{\frac{5}{2}}+11\sqrt{2}, 5\,2^{\frac{5}{2}}\sqrt{3}+49, 89\sqrt{3}+109\sqrt{2}\right]$$

Rechnen mit Wurzelausdrücken kann oft unerwartet schwierig sein, etwa die Umwandlung des Ausdrucks $\frac{1}{\sqrt{2}+\sqrt{3}}$ in die gleichwertige Form $\sqrt{3}-\sqrt{2}$. Mit MATHEMATICA oder MAXIMA kann das mit den Standardinstrumenten nicht erreicht werden, obwohl das Vorgehen – Erweitern des Bruchs mit $\sqrt{3}-\sqrt{2}$, dem Konjugierten des Nenners – weitgehend klar ist.

Auch in MUPAD liefern weder `expand` noch `normal` oder `simplify` eine Ergebnis mit rationalem Nenner. Erst die aufwändige Funktion `radsimp` liefert das erwartete Ergebnis, das man auch schnell im Kopf ausrechnen kann. Dasselbe Ergebnis liefert die MAPLE-Funktion `evala`.

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

$$\sqrt{3}-\sqrt{2}$$

Beide Funktionen können auch kompliziertere Wurzelausdrücke „rational machen“ wie etwa

```
evala(1/(sqrt(2)+sqrt(3)+sqrt(5)));
```

$$-\frac{1}{12}\sqrt{2}\sqrt{3}\sqrt{5}+\frac{1}{6}\sqrt{3}+\frac{1}{4}\sqrt{2}$$

Beide CAS haben entsprechende Algorithmen implementiert, die in den anderen Systemen nicht bzw. nicht direkt zur Verfügung steht. Der Grund für eine solche Designentscheidung liegt darin, dass es aus Effizienzgründen nicht klug ist, Nenner rational zu machen. Wir kommen auf diese Frage später zurück. AXIOM führt auch diese Umformungen automatisch aus.

1.3 CAS als Problemlösungs-Umgebungen

Wir haben in obigen Beispielen bereits in bescheidenem Umfang programmiersprachliche Mittel eingesetzt, um unsere speziellen Wünsche zu formulieren.

> Das Vorhandensein einer voll ausgebauten Programmiersprache, mit der man den Interpreter des jeweiligen CAS gut steuern kann, ist ein weiteres Charakteristikum der Mehrzahl der betrachteten Systeme. Dabei werden alle gängigen Sprachkonstrukte einer imperativen Programmiersprache unterstützt und noch um einige Spezifika erweitert, die aus der Natur des symbolischen Rechnens folgen und über die weiter unter zu sprechen sein wird.

Mit diesem Instrumentarium und dem eingebauten mathematischen Wissen werden CAS so zu einer vollwertigen Problemlösungs-Umgebung, in der man neue Fragestellungen und Vermutungen (mathematischer Natur) ausgiebig testen und untersuchen kann. Selbst für zahlentheoretische Fragestellungen reichen die Möglichkeiten dabei weit über die des Taschenrechners hinaus.

Betrachten wir etwa Aufgaben der Art:

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

wie sie in Schülerarbeitsgemeinschaften vor Einführung von CAS zum Kennenlernen des Rechnens mit Resten gern gestellt wurden.

Die Originalaufgabe ist für ein CAS gegenstandslos, da man die ganze Zahl leicht ausrechnen kann.

```
2^100;
```

```
1267650600228229401496703205376
```

Erst im Bereich von Exponenten in der Größenordnung 1 000 000 wird der Verbrauch von Rechenzeit ernsthaft spürbar und die dann über 300 000-stelligen Zahlen zunehmend unübersichtlich.

Wirkliche Probleme bekommt der Nutzer, wenn er die dem Computer „angemessene“ Frage nach letzten Ziffern der Zahl $2^{10^{10}}$ stellt.

Zunächst ist zu beachten, dass `2^10^10` sowohl als `(2^10)^10` als auch als `2^(10^10)` verstanden werden kann. Da `^` nicht assoziativ ist, kommt es hier auf die Klammersetzung an. Mit Blick auf die Potenzgesetze ist allein die zweite – die rechtsassoziative – Auslegung sinnvoll, denn es gilt $\left(2^{10}\right)^{10} = 2^{100}$. Maple lässt solche Ausdrücke ohne Klammern deshalb gar nicht erst zu, Mathematica, Maxima und Sage parsen den Ausdruck `2^10^10` (korrekt) rechtsassoziativ, Reduce und Matlab (falsch) linksassoziativ.

Bei der korrekten Eingabe `2^(10^10)` gibt MuPAD nach endlicher Zeit auf. Ähnlich reagieren Mathematica, Maple und Sage, während Maxima losrechnet und nur durch einen Interrupt wieder anzuhalten ist.

```
2^10^10
```

```
Error: Overflow/underflow in
arithmetical operation
```

Der ursprüngliche Sinn der Aufgabe bestand darin, zu erkennen, dass man zur Berechnung der letzten Ziffer nicht die gesamte Potenz, sondern nur deren Rest (mod 10) berechnen muss und dass Potenzreste $2^k \pmod{10}$ eine periodische Folge bilden. Mit einem CAS kann man den Rechner für eine wesentlich weitergehende Untersuchung derselben Problematik einsetzen. Da dieses in der Lage ist, die ermüdende Berechnung der Potenzreste zu übernehmen (nachdem dieser Gegenstand ggf. genügend geübt worden ist), können wir nach Regelmäßigkeiten für die Potenzreste für beliebige Moduln fragen.

```
Table[Mod[2^k, 10],{k,1,15}]
```

$$\{2, 4, 8, 6, 2, 4, 8, 6, 2, 4, 8, 6, 2, 4, 8\}$$

```
Table[Mod[2^k, 3],{k,1,15}]
```

$$\{2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2\}$$

```
Table[Mod[2^k, 11],{k,1,15}]
```

$$\{2, 4, 8, 5, 10, 9, 7, 3, 6, 1, 2, 4, 8, 5, 10\}$$

```
Table[Mod[2^k, 17],{k,1,15}]
```

$$\{2, 4, 8, 16, 15, 13, 9, 1, 2, 4, 8, 16, 15, 13, 9\}$$

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.

Auch die modifizierte Aufgabe zur Bestimmung letzter Ziffern von $2^{10^{10}}$ können wir nun lösen.

Der erste Versuch schlägt allerdings fehl: Die Auswertungsregeln der Informatik führen dazu, dass *vor* Auswertung der `Mod`-Funktion die inneren Argumente ausgewertet werden, d. h. zunächst der Versuch unternommen wird, die Potenz vollständig auszurechnen.

```
Mod[2^10^10, 10^5]
```

```
Overflow[]
```

Wir brauchen statt dessen eine spezielle Potenzfunktion, die bereits in den Zwischenrechnungen die Reste reduziert und damit Zwischenergebnisse überschaubarer Größe produziert. In Mathematica bzw. Maxima steht dafür die spezielle Funktion `PowerMod` bzw. `power_mod` zur Verfügung, die in diesem Fall direkt aufgerufen werden muss.

```
PowerMod[2,10^10,10^20]
```

```
power_mod(2,10^10,10^20)
```

$$46374549681787109376$$

Andere CAS verwenden Eingabeinformationen, um nach einer solchen Funktion zu suchen, so dass die Aufgabe in „natürlicher“ Notation angeschrieben werden kann und der Nutzer nicht nach Funktionsnamen suchen muss, unter denen die spezielle Funktionalität implementiert ist. Ein solches Vorgehen – Auflösung des Funktionsnamens zur Laufzeit nach dem Typ der Aufrufargumente – ist aus dem objektorientierten Programmieren gut bekannt.

Einen solchen Ansatz verfolgt Axiom, das funktionale Polymorphie (hier der Potenzfunktion) an Hand der Argumenttypen auflösen kann. Die korrekte Potenzfunktion wird also ausgewählt, wenn bereits die Zahl 2 als Element von $\mathbb{Z}_{10^8}$ erzeugt wird.

```
++ Axiom
a:IntegerMod(10^8):= 2
a^(10^10)
```

```
87109376    Type: IntegerMod(100000000)
```

Eine ähnliche Notation ist mit SAGE und auch der alten Version von MUPAD möglich.

```
# Sage
u = Mod(2, 10^20)
u^10^10
```

$$463745496817871093 76$$

In MAPLE erreichen wir denselben Effekt über den *inerten Operator* `&^` (auf solche inerten Funktionen werden wir später noch eingehen).

```
2 &^(10^10) mod 10^20;
```

$$463745496817871093 76$$

Wir wollen die auch aus didaktischen Gesichtspunkten interessante Möglichkeit, CAS als Problemlösungs- und Experimentierumgebung einzusetzen, zur Erforschung von Eigenschaften ganzer Zahlen an einem weiteren Beispiel verdeutlichen:

Definition 1 *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. Jahrhundert 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. MATHEMATICA kennt eine Funktion `DivisorSigma`, mit der wir die Teilersumme der natürlichen Zahl n berechnen können. Mit einer Schleife verschaffen wir uns erst einmal einen Überblick über die perfekten Zahlen bis 1000.

```
For[i=2,i<1000,i++,
  If[DivisorSigma[1,i]==2*i,Print[i]]
]
```

```
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.

Es ist meist nicht sinnvoll, die Ergebnisse wie oben mit einer `Print`-Anweisung anzuzeigen, da die entsprechenden Ausdrücke dann zwar auf dem Bildschirm zu sehen sind, jedoch nicht direkt weiter verwendet werden könnnen. Wir wollen die Ergebnisse der folgenden Rechnung deshalb aggregieren und unter einem Bezeichner zum Zweck der weiteren Verwendung speichern.

```
uhu = Table[ n=2^(k-1)*(2^k-1); {k,n,DivisorSigma[1,n]==2*n}, {k,2,40} ]
```

$$\{\{2, 6, \text{True}\}, \{3, 28, \text{True}\}, \{4, 120, \text{False}\}, \ldots, \{40, 604462909806764831539200, \text{False}\}\}$$

Diese Liste von Listen lässt sich nun einfach als Tabelle ausgeben, da MATHEMATICA Matrizen als Listen von Listen speichert und deshalb umgekehrt auch Listen von Listen als Matrizen anzeigen kann.

```
uhu // MatrixForm
```

$$\begin{pmatrix}
2 & 6 & \text{True} \\
3 & 28 & \text{True} \\
4 & 120 & \text{False} \\
5 & 496 & \text{True} \\
6 & 2016 & \text{False} \\
7 & 8128 & \text{True} \\
8 & 32640 & \text{False} \\
9 & 130816 & \text{False} \\
10 & 523776 & \text{False} \\
11 & 2096128 & \text{False} \\
12 & 8386560 & \text{False} \\
13 & 33550336 & \text{True} \\
14 & 134209536 & \text{False} \\
15 & 536854528 & \text{False} \\
16 & 2147450880 & \text{False} \\
17 & 8589869056 & \text{True} \\
18 & 34359607296 & \text{False} \\
19 & 137438691328 & \text{True} \\
20 & 549755289600 & \text{False} \\
 & \dots &
\end{pmatrix}$$

Die Ausgabe der Ergebnisse der Rechnungen für Zahlen der Gestalt $n = 2^{k-1}(2^k - 1)$ bietet umfangreiches experimentelles Material, 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

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

In der Tat, multipliziert man den Ausdruck aus, so hat man aus jeder Klammer einen Summanden zu nehmen und zu einem Produkt zusammenzufügen. Alle möglichen Auswahlen ergeben genau die beschriebenen Teiler von n.

Die Teilersumme $\sigma(n)$ ergibt sich also unmittelbar aus der Kenntnis der Primfaktorzerlegung der Zahl n. Ist insbesondere $n = a \cdot b$ eine zusammengesetzte Zahl mit zueinander teilerfremden Faktoren a, b , so gilt offensichtlich $\sigma(n) = \sigma(a) \cdot \sigma(b)$.

Wenden wir das auf die gerade perfekte Zahl $n = 2^{k-1}b$ ($k > 1, b$ ungerade) an, so gilt wegen $\sigma(2^{k-1}) = 1 + 2 + 2^2 \ldots + 2^{k-1} = 2^k - 1$

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

Also ist $2^k - 1$ ein Teiler von $2^k\,b$ und als ungerade Zahl damit auch von b. Es gilt folglich $b = (2^k - 1)c$ und somit $\sigma(b) = 2^k\,c = b + c$. Da b und c Teiler von b sind und $\sigma(b)$ *alle* Teiler von b aufsummiert, hat b nur die Teiler b und $c = 1$, 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 $2^k - 1$ eine Primzahl sein muss.*

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

1.4 Computeralgebrasysteme als Expertensysteme

Neben den bisher betrachteten Rechenfertigkeiten und der Programmierbarkeit, die ein CAS auszeichnen, spielt das

in ihnen gespeicherte mathematische Wissen

für Anwender die entscheidende Rolle. Dieses deckt, wenigstens insofern wir uns auf die dort vermittelten algorithmischen Fertigkeiten beschränken, weit mehr als den Stoff der gymnasialen Oberstufe ab und umfasst die wichtigsten symbolischen Kalküle der Analysis (Differenzieren und Integrieren, Taylorreihen, Grenzwertberechnung, Trigonometrie, Rechnen mit speziellen Funktionen), der Algebra (Matrizen- und Vektorrechnung, Lösen von linearen und polynomialen Gleichungssystemen, Rechnen in Restklassenbereichen), der Kombinatorik (Summenformeln, Lösen von Rekursionsgleichungen, kombinatorische Zahlenfolgen) und vieler anderer Gebiete der Mathematik.

Für viele Kalküle aus mathematischen Teildisziplinen, aber zunehmend auch solche aus verwandten Bereichen anderer Natur- und Ingenieurwissenschaften, ja selbst der Finanzmathematik, gibt es darüber hinaus spezielle Pakete, auf die man bei Bedarf zugreifen kann. Dieses sprunghaft wachsende Expertenwissen ist der eigentliche Kern eines CAS als Expertensystem. Man kann mit gutem Gewissen behaupten, dass heute bereits große Teile der algorithmischen Mathematik in dieser Form verfügbar sind und auch algorithmisches Wissen aus anderen Wissensgebieten zunehmend erschlossen wird. In dieser Richtung spielt das System MATHEMATICA seit Jahren eine Vorreiterrolle.

In dem Zusammenhang ist die im deutschen Sprachraum verbreitete Bezeichnung *Computeralgebra* irreführend, denn es geht durchaus nicht nur um algebraische Algorithmen, sondern auch um solche aus der Analysis und anderen Gebieten der Mathematik. Entscheidend ist allein der *algebraische Charakter* der entsprechenden Manipulationen als endliche Kette von Umformungen symbolischer Ausdrücke. Und genau so geht ja etwa der Kalkül der Differentialrechnung vor: die konkrete Berechnung von Ableitungen elementarer Funktionen erfolgt nicht nach der (auf dem Grenzwertbegriff aufbauenden) Definition, sondern wird durch geschicktes Kombinieren verschiedener *Regeln*, wie der Ketten-, der Produkt- und der Quotientenregel, auf entsprechende Grundableitungen zurückgeführt.

Hier einige Beispielrechnungen mit MAXIMA:

```
diff(x^2*sin(x)*log(x+a),x,2)
```

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

Beim Integrieren kommen darüber hinaus auch prozedurale Verfahren zum Einsatz, wie etwa die Partialbruchzerlegung, die ihrerseits die Faktorzerlegung des Nennerpolynoms und das Lösen von (linearen) Gleichungssystemen verwendet, die aus einem Ansatz mit unbestimmten Koeffizienten gewonnen werden. Auf diese Weise, und das ist ein weiteres wichtiges Moment des Einsatzes von CAS,

spielen algorithmische Fertigkeiten aus sehr verschiedenen Bereichen der Mathematik nahtlos miteinander zusammen.

```
p : (x^3+x^2+x-2)/(x^4+x^2+1)
```

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

```
u : integrate(p,x)
```

$$-\frac{1}{2}\,\log\left(x^2+x+1\right)+\log\left(x^2-x+1\right)-\frac{1}{\sqrt{3}}\,\arctan\left(\frac{2\,x+1}{\sqrt{3}}\right)$$

```
diff(u,x)
```

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

```
ratsimp(%-p)
```

$$0$$

```
integrate(1/(x^4-1),x)
```

$$-\frac{1}{4}\,\log(x+1)-\frac{1}{2}\,\arctan(x)+\frac{1}{4}\,\log(x-1)$$

Wir können die letzte Integrationsaufgabe für verschiedene Exponenten untersuchen und damit schon einmal die Leistungsfähigkeit eines CAS testen. Eine entsprechende Testprozedur hätte für MAXIMA etwa diese Gestalt. Die Ergebnisse entsprechen weitgehend denen, die nach der Theorie zu erwarten sind, sofern die Teilintegrale exakt berechnet werden können.

```
intChallenge(n):=(
  f:1/(1-x^n),
  u:integrate(f,x),
  v:diff(u,x),
  w:ratsimp(v-f),
  [n,f,u,v,w]
);
```

Expertenwissen wird zum Lösen von verschiedenen Problemklassen benötigt, wobei der Nutzer in den seltensten Fällen wissen (und wissen wollen) wird, welche konkreten Algorithmen im jeweiligen Fall genau anzuwenden sind. Deshalb bündeln die CAS die algorithmischen Fähigkeiten zum Auflösen gewisser Sachverhalte in einem oder mehreren

`solve`-Operatoren,

die den Typ eines Problems erkennen und geeignete Lösungsalgorithmen aufrufen.

So lassen sich mit dem `solve`-Operator von MAXIMA Gleichungen und Gleichungssysteme und auch einfache goniometrische Bestimmungsgleichungen lösen.

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

$$\left[x=-\frac{1+\sqrt{3}\;\mathrm{i}}{2},x=\frac{\sqrt{3}\;\mathrm{i}-1}{2}\right]$$

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

$$\left[\left[x=-\frac{\sqrt{5}-1}{2},y=\frac{1+\sqrt{5}}{2}\right],\left[x=\frac{1+\sqrt{5}}{2},y=-\frac{\sqrt{5}-1}{2}\right],[x=-2,y=-2],[x=1,y=1]\right]$$

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

```
solve: using arc-trig functions to get a solution. Some solutions will be lost.
```

$$\left[x = \frac{\pi}{6}\right]$$

Allein das letzte Ergebnis ist unbefriedigend, da die Gleichung natürlich weitere Lösungen hat, wie ein Plot `plot2d(sin(x)-1/2,[x,-10,10])` schnell zeigt.

Der `solve`-Operator von MuPAD bündelt Wissen über Algorithmen zum Auffinden der Nullstellen univariater Polynome, von linearen und polynomialen Gleichungssystemen. Selbst Differentialgleichungen können damit gelöst werden. Als Ergebnis werden Mengen statt Listen zurückgegeben.

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

$$\left\{-\frac{1}{2}\,\mathrm{i}\sqrt{3} - \frac{1}{2}, \frac{1}{2}\,\mathrm{i}\sqrt{3} - \frac{1}{2}\right\}$$

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

$$\left\{[x = 1, y = 1]\,,\ [x = 2, y = -2]\,,\ \left[x = -\frac{\mathrm{i}}{2}\sqrt{3} - \frac{3}{2}, y = \frac{1}{2} - \frac{3\,\mathrm{i}}{2}\sqrt{3}\right],\ \left[x = \frac{\mathrm{i}}{2}\sqrt{3} - \frac{3}{2}, y = \frac{3\,\mathrm{i}}{2}\sqrt{3} + \frac{1}{2}\right]\right\}$$

Selbst für kompliziertere Gleichungen, die trigonometrische Funktionen enthalten, werden Lösungen in mathematisch korrekter Form angegeben.

```
s:=solve(sin(x)=1/2,x);
```

$$\left\{\frac{\pi}{6} + 2\,k\,\pi \mid k \in \mathbb{Z}\right\} \cup \left\{\frac{5\,\pi}{6} + 2\,k\,\pi \mid k \in \mathbb{Z}\right\}$$

Eine Spezialität von MuPAD ist die vollständige und mathematisch weitgehend genaue Angabe der Lösung als Lösungsmenge, die sich mit entsprechenden Mengenoperationen weiter verarbeiten lässt. Sucht man etwa alle Lösungen obiger Aufgabe im Intervall $-10 \le x \le 10$, so kann man diese als Mengendurchschnitt bestimmen und dazu dann Näherungswerte ausgeben lassen. Beachten Sie die unterschiedliche Reihenfolge der Elemente in der Printausgabe. Aber es handelt sich ja auch um Mengen.

```
u1:=s intersect Dom::Interval(-10,10);
```

$$\left\{\frac{\pi}{6}, \frac{5\,\pi}{6}, -\frac{7\,\pi}{6}, -\frac{11\,\pi}{6}, \frac{13\,\pi}{6}, \frac{17\,\pi}{6}, -\frac{19\,\pi}{6}\right\}$$

```
float(u1);
```

$$\{-9.948376, -5.759586, -3.665191, 0.5235987, 2.617993, 6.806784, 8.901179\}$$

MAPLEs Lösung derselben Aufgabe fällt unbefriedigender aus. Wenn wir uns erinnern, dass die Lösungsmenge trigonometrischer Gleichungen periodisch ist, also mit x auch $x + 2\pi$ die Gleichung erfüllt, so haben wir aber immerhin schon die Hälfte der tatsächlichen Lösungsmenge gefunden.

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

$$\frac{1}{6}\,\pi$$

Warum kommt MAPLE nicht selbst auf diese Idee? Auch hier hilft erst ein Blick in die (Online-)Dokumentation weiter; setzt man eine spezielle Systemvariable richtig, so erhalten wir das erwartete Ergebnis, allerdings in einer recht verklausulierten Form: `_B1` und `_Z1` sind Hilfsvariablen mit den Wertebereichen $\texttt{\_B1} \in \{0, 1\}$ (B wie boolean) und $\texttt{\_Z1} \in \mathbb{Z}$.

```
_EnvAllSolutions:=true;

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

$$\frac{1}{6}\,\pi + \frac{2}{3}\,\pi\,\texttt{\_B1\~{}} + 2\,\pi\,\texttt{\_Z1\~{}}$$

MATHEMATICA bis zur Version 8 (wie auch MAXIMA noch in aktuellen Versionen) antwortet ähnlich, aber bis zur Version 5 gab es keine Möglichkeit, die Lösungsschar in parametrisierter Form auszugeben. Im Handbuch der Version 3 (S. 794) wurde dies wie folgt begründet:

```
Solve[Sin[x]==1/2,x]
```

```
Solve::ifun: Inverse functions
are being used by Solve, so some
solutions may not be found.
```

$$\left\{\left\{x \to \frac{\pi}{6}\right\}\right\}$$

> *Obwohl sich alle Lösungen dieser speziellen Gleichung einfach parametrisieren lassen, führen die meisten derartigen Gleichungen auf schwierig darzustellende Lösungsmengen. Betrachtet man Systeme trigonometrischer Gleichungen, so sind für eine Parameterdarstellung diophantische Gleichungssysteme zu lösen, was im allgemeinen Fall als algorithmisch nicht lösbar bekannt ist.*

Wir stoßen mit unserer einfachen Frage also unvermutet an prinzipielle Grenzen der Mathematik, die im betrachteten Beispiel allerdings noch nicht erreicht sind.

Seit Version 5 kann das Kommando `Reduce` verwendet werden, um einen zur Eingabe äquivalenten logischen Ausdruck ähnlicher Qualität wie die Antworten von MUPAD zu generieren.

```
Reduce[Sin[x] == 1/2, x]
```

$$c_1 \in \mathbb{Z}\ \&\&\ \left(x = 2\,\pi\,c_1 + \frac{\pi}{6}\ ||\ x = 2\,\pi\,c_1 + \frac{5\,\pi}{6}\right)$$

Die MATHEMATICA-Entwickler stützen sich dabei auf eine andere Philosophie als die MUPAD-Entwickler. Während letztere konsequent die Mengennotation $L = \{f(t) \,|\, t \in G \wedge H(t)\}$ verwenden, welche die Lösungsmenge durch eine parameterabhängige Vorschrift f konstruieren, wobei für die Parameter $t \in G$ zusätzliche Nebenbedingungen $H(t)$ aufgestellt sein können, stützen sich die ersteren auf die gezielte Umformung dieser Eigenschaften selbst.

```
Reduce[Sin[x] == 1/2 && -8 < x < 8, x]
```

$$x = -\frac{11\,\pi}{6}\ ||\ x = -\frac{7\,\pi}{6}\ ||\ x = \frac{\pi}{6}\ ||\ x = \frac{5\,\pi}{6}\ ||\ x = \frac{13\,\pi}{6}$$

Seit Version 9 verwendet `Solve` die neu eingeführte Konstruktion `ConditionalExpression` zur Darstellung von Lösungsmengen

```
Solve[Sin[x] == 1/2, x]
```

$$\left\{\left\{x \to \texttt{ConditionalExpression}\left[\frac{\pi}{6} + 2\,\pi\,C[1], C[1] \in \mathbb{N}\right]\right\},\right.$$
$$\left.\left\{x \to \texttt{ConditionalExpression}\left[\frac{5\,\pi}{6} + 2\,\pi\,C[1], C[1] \in \mathbb{N}\right]\right\}\ \right\}$$

```
Solve[Sin[x] == 1/2 && -8 < x < 8, x]
```

$$\left\{\left\{x \to -\frac{11\,\pi}{6}\right\}, \left\{x \to -\frac{7\,\pi}{6}\right\}, \left\{x \to \frac{\pi}{6}\right\}, \left\{x \to \frac{5\,\pi}{6}\right\}, \left\{x \to \frac{13\,\pi}{6}\right\}\right\}$$

Oft haben wir nicht nur mit der Frage der Vollständigkeit der angegebenen Lösungsmenge zu kämpfen, sondern auch mit unterschiedlichen Darstellungen ein und desselben Ergebnisses, die sich aus unterschiedlichen Lösungswegen ergeben. Betrachten wir etwa die Aufgabe

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

Eine mögliche Umformung, die uns die vollständige Lösungsmenge liefert, wäre

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

womit die Gleichung zu $\cos\left(x - \frac{\pi}{4}\right) = \frac{1}{2\sqrt{2}}$ umgeformt wurde.
Als Ergebnis erhalten wir (wegen $\cos(x) = u \ \Rightarrow\ x = \pm\arccos(u) + 2\,k\,\pi$)

$$L = \left\{\frac{\pi}{4} + \arccos\left(\frac{1}{2\sqrt{2}}\right) + 2\,k\,\pi,\ \frac{\pi}{4} - \arccos\left(\frac{1}{2\sqrt{2}}\right) + 2\,k\,\pi \;\middle|\; k \in \mathbb{Z}\right\}.$$

MAXIMA streicht an dieser Stelle bereits komplett die Segel.

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

$$\left[\sin(x) = -\frac{2\,\cos(x) - 1}{2}\right]$$

MUPAD in der Version 8 antwortet

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

$$\left\{2\,k\,\pi - \ln\left(\frac{1}{4} - \frac{1}{2}\sqrt{-\frac{7}{2}}\,\mathrm{i} + \frac{\mathrm{i}}{4}\right)\ \mathrm{i} \,|\, k \in \mathbb{Z}\right\} \bigcup \left\{2\,k\,\pi - \ln\left(\frac{1}{2}\sqrt{-\frac{7}{2}}\,\mathrm{i} + \frac{1}{4} + \frac{\mathrm{i}}{4}\right)\ \mathrm{i} \,|\, k \in \mathbb{Z}\right\},$$

während MAPLE folgende Antwort liefert:

```
_EnvAllSolutions:=true:
s:=[solve(sin(x)+cos(x)=1/2,x)];
```

$$\left[\arctan\left(\frac{\frac{1}{4} - \frac{1}{4}\sqrt{7}}{\frac{1}{4} + \frac{1}{4}\sqrt{7}}\right) + 2\,\pi\,\_Z3, \arctan\left(\frac{\frac{1}{4} + \frac{1}{4}\sqrt{7}}{\frac{1}{4} - \frac{1}{4}\sqrt{7}}\right) + \pi + 2\,\pi\,\_Z3\right]$$

Hier wird eine zentrale Fragestellung deutlich, welche in der praktischen Arbeit mit einem CAS ständig auftritt:

Wie weit sind syntaktisch verschiedene Ausdrücke sematisch äquivalent, stellen also ein und denselben mathematischen Ausdruck dar?

In obigem Beispiel wurden die Lösungen von den verschiedenen CAS in sehr unterschiedlichen Formen präsentiert, die sich durch einfache (im Sinne von „offensichtliche“) Umformungen weder ineinander noch in das von uns erwartete Ergebnis überführen lassen.

Versuchen wir wenigstens die näherungsweise Auswertung einiger Elemente beider Lösungsmengen:

```
/* MuPAD */
float(s intersect Dom::Interval(-10,10));
```

$$\{-6.707216347, -4.288357941, -0.4240310395, 1.994827366, 5.859154268, 8.278012674\}$$

```
# Maple
seq(op(subs(_Z3=i,evalf(s))), i=-1..1);
```

$$-6.707216348, -4.288357941, -0.4240310395, 1.994827367, 5.859154268, 8.278012675$$

```
(* Mathematica 9 *)
Solve[Sin[x] + Cos[x] == 1/2 && -10 < x < 10 , x] // N
```

$$\left\{ \begin{array}{l} \{x \to -0.424031\}, \{x \to -6.70722\}, \{x \to 5.85915\}, \{x \to 1.99483\}, \\ \{x \to -4.28836\}, \{x \to 8.27801\} \end{array} \right\}$$

Die Ergebnisse gleichen sich, was es plausibel macht, dass es sich in der Tat um verschiedene syntaktische Formen desselben mathematischen Inhalts handelt. Natürlich sind solche näherungsweisen Auswertungen kein *Beweis* der sematischen Äquivalenz und verlassen außerdem den Bereich der exakten Rechnungen.

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.

1.5 Erweiterbarkeit von Computeralgebrasystemen

Ein weiterer Aspekt, der für den Einsatz eines CAS als Expertensystem wichtig ist, besteht in der

Möglichkeit der Erweiterung seiner mathematischen Fähigkeiten.

Die zur Verfügung stehende Programmiersprache kann nicht nur zur Ablaufsteuerung des Interpreters verwendet werden, sondern auch (in unterschiedlichem Umfang), um eigene Funktionen und ganze Pakete zu definieren.

Betrachten wir etwa die Vereinfachungen, die sich bei der Untersuchung des rationalen Ausdrucks 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.

Definition 2 *Sei $X = (x_1, \ldots, x_n)$ eine endliche Menge von Variablen.*
Ein Polynom $f = f(x_1, \ldots, x_n)$ bezeichnet man als symmetrisch*, wenn für alle Permutationen $\pi \in S_n$ die Polynome f und $f^\pi = f(x_{\pi(1)}, \ldots, x_{\pi(n)})$ übereinstimmen.*

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

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

und

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

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 MAXIMA lauten

```
e(d,vars):=block([u],
  if length(vars)<d then return(0)
  else if d=0 then return(1)
  else u:rest(vars), expand(e(d,u)+first(vars)*e(d-1,u)));

h(d,vars):=block([],
  if d=0 then return(1)
  else if length(vars)=1 then first(vars)^d
  else expand(h(d,rest(vars))+first(vars)*h(d-1,vars)));

p(d,vars):= apply("+",map(lambda([x],x^d),vars));
```

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.
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.

```
vars:[x,y,z];
h(2,vars);
```

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

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

$$x\,y\,z$$

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

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

1.6 Was ist Computeralgebra ?

Was ist nun Computeralgebra? Wir hatten gesehen, dass sie eine spezielle Art von Symbolverarbeitung zum Gegenstand hat, in der, im Gegensatz zur Textverarbeitung, die Symbole mit Inhalten, also einer *Semantik*, verbunden sind. Auch geht es bei der Verarbeitung um Manipulationen eben dieser Inhalte und nicht primär der Symbole selbst.

Von der Natur der Inhalte und der Form der Manipulationen her können wir den Gegenstand der Computeralgebra also in erster Näherung als

symbolisch-algebraische Manipulationen mathematischer Inhalte

bezeichnen. Diese „Natur der Inhalte“ ist dem Computer natürlich nicht direkt zugänglich, sondern bedarf zur adäquaten Anwendung des an Menschen und menschliches Wissen gebundenen *Verständnisses* dieser Inhalte. In einem solch breiteren Verständnis entpuppt sich die Computeralgebra als Teil der Informatik, wenn man diese nicht als „Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Digitalrechnern“ [4], sondern als „technologische Seite der Denkens“ (B. Buchberger) versteht.

Stellen wir die Art der internen Verarbeitung in den Mittelpunkt und gehen eher von der syntaktischen Form aus, in der uns diese Inhalte entgegentreten, so lässt sich der Gegenstand grob als

Rechnen mit Symbolen, die mathematische Objekte repräsentieren

umreißen. 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.

Das Adjektiv **symbolisch** bedeutet, dass das Ziel die Suche nach einer geschlossenen oder approximativen Formel im Sinne des deduktiven Mathematikverständnisses ist.

Das Adjektiv **algebraisch** bedeutet, dass eine *exakte* mathematische Ableitung aus den Ausgangsgrößen durchgeführt wird, anstatt näherungsweise Fließkommaarithmetik einzusetzen. Es impliziert keine Einschränkung auf spezielle Teilgebiete der Mathematik, sondern eine der verwendeten Methoden. Diese sind als mathematische Schlussweise weit verbreitet, denn auch Anwendungen aus der Analysis, welche – wie z.B. Grenzwert- oder Integralbegriff – per definitionem Näherungsprozesse untersuchen, verwenden in ihrem eigenen Kalkül solche algebraischen Umformungen. Dies wird am Unterschied zwischen der Ableitungsdefinition und dem Vorgehen bei der praktischen Bestimmung einer solchen Ableitung deutlich, vgl. auch [11]. Beispiele für solche

algebraisch-symbolischen Umformungen sind Termumformungen, Polynomfaktorisierung, Reihenentwicklung von Funktionen, analytische Lösungen von Differentialgleichungen, exakte Lösungen polynomialer Gleichungssysteme oder die Vereinfachung mathematischer Formeln.

1.7 Computeralgebra und Computermathematik

Computeralgebraische Werkzeuge beherrschen heute schon einen großen Teil der algorithmisch zugänglichen mathematischen und zunehmend auch naturwissenschaftlichen und ingenieurtechnischen Kalküle und bieten fach- und softwarekundigen Anwendern Zugang zu entsprechendem Know-how auf Black-Box-Basis. Implementierungen fortgeschrittener Kalküle aus den Einzelwissenschaften sind ihrerseits nicht denkbar ohne Zugang zu effizienten Implementierungen der zentralen mathematischen Kalküle aus Algebra und Analysis.

Historisch stand und stehen dabei *Termumformungen*, also das Erkennen semantischer Gleichwertigkeit syntaktisch unterschiedlicher Ausdrücke, sowie das effiziente Rechnen mit polynomialen und rationalen Ausdrücken am Ausgangspunkt. Die hierbei verwendeten Methoden unterscheiden sich oftmals sehr von der durch „mathematische Intuition“ gelenkten und stärker heuristisch geprägten Schlussweise des Menschen und greifen die Entwicklungslinien der konstruktiven Mathematik aus den 1920er Jahren wieder auf, vgl. R.Loos in [11].

Neben der numerisch-algorithmischen Sicht auf den Computer als „number cruncher“ spielten in den 1970er Jahren zunächst nichtnumerische Applikationen wie z. B. Anwendungen der diskreten Mathematik (Kombinatorik, Graphentheorie) oder der diskreten Optimierung eine zentrale Rolle, die *endliche* Strukturen untersuchen, welche sich *exakt* im Computer reproduzieren lassen. Den spektakulärsten Durchbruch markiert der computergestützte Beweis des Vier-Farben-Satzes durch Appel und Haken im Jahre 1976, der eine kontroverse Diskussion darüber auslöste, wie weit solche Computerbeweise überhaupt als Beweise im mathematisch-deduktiven Sinne anerkannt werden können. Dies ist heute weitgehend unbestritten, wenn die verwendeten Programme selbst einer Korrektheitsprüfung standhalten.

Im Gegensatz zur diskreten Mathematik hat Computeralgebra mathematische Konstruktionen zum Gegenstand, die zwar syntaktisch endlich (und damit *exakt* im Computer darstellbar) sind, aber semantisch unendliche Strukturen repräsentieren können. Diese *beschreibungsendlichen Strukturen* kommen mathematischen Arbeitstechniken näher als Anwendungen und implementierte Kalküle der numerischen oder diskreten Mathematik.

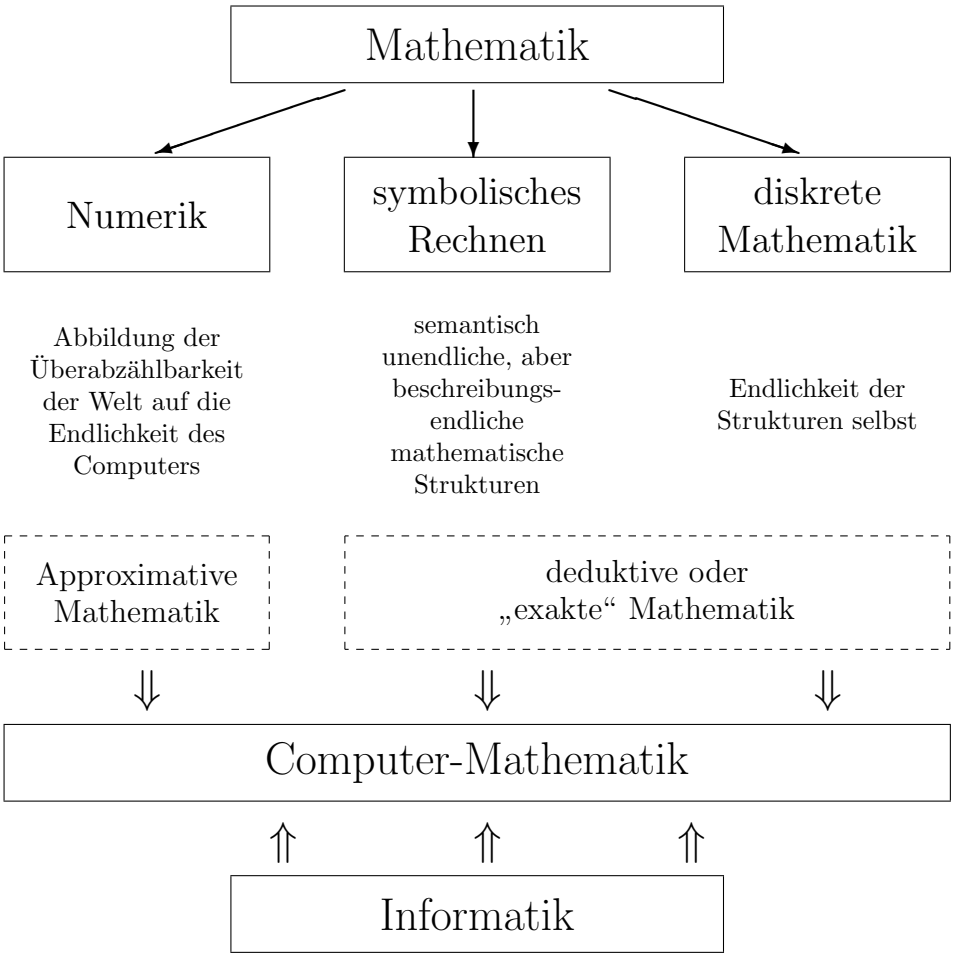
In diesem Sinne beschreibt J.Grabmeier in [6], auf R.Loos zurückgehend,

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

Das Beiwort „algebraisch“ bezieht sich dabei, wie oben erläutert, auf die eingesetzten Methoden. Buchberger verwendet in [1, S. 799] die genaueren Adjektive „exakt“ und „abstrakt“:

> *Symbolisches Rechnen ist der Teil der algorithmischen Mathematik, der sich mit dem exakten algorithmischen Lösen von Problemen in abstrakten mathematischen Strukturen befasst.*

Im Weiteren unterstreicht Buchberger die Bedeutung der Algebraisierung und Algorithmisierung mathematischer Fragestellungen (der „Trivialisierung von Problemstellungen“), um sie einer computeralgebraischen Behandlung im engeren Sinne zugänglich zu machen, und schlägt diesen Aufwand dem symbolischen Rechnen zu. Allerdings werden so die Grenzen zu anderen mathematischen Teilgebieten verwischt, die sich zusammen mit der Computeralgebra im engeren Sinne arbeitsteilig an der Entwicklung und Implementierung der jeweiligen Kalküle beteiligen. Ein solches Verständnis blendet zugleich den technikorientierten Aspekt der Computeralgebra als Computerwissenschaft weitgehend aus.



Die Genese der Computermathematik

Schließlich reicht die Bedeutung der Verfügbarkeit von Computeralgebra-Systemen weit über den Bereich der algorithmischen Mathematik hinaus. Die Vielzahl mathematischer Verfahren, die in einem modernen CAS unter einer *einheitlichen* Oberfläche verfügbar sind, machen dieses zu einem **metamathematischen Werkzeug** für Anwender, ähnlich den Numerikbibliotheken, die im Wissenschaftlichen Rechnen schon lange eine zentrale Rolle spielen. Die zusätzlichen Visualisierungs- und Präsentationsmöglichkeiten, die weltweite Vernetzung, der Zugang zu ständig aktualisierten Datenbeständen und vieles mehr machen die führenden CAS heute zu dem, was MATHEMATICA als Leitspruch erkoren hat: „A fully integrated environment for technical computing.“ Die damit verbundenen Anforderungen an das informations-technische Management bilden einen eigenständigen Teil der Herausforderungen, vor denen die Computeralgebra als interdisziplinäres Gebiet steht.

In einer sich damit immer mehr etablierenden **Computermathematik** [6] als Symbiose dieser Entwicklungen stehen computergestützte numerische, diskrete und symbolische Methoden gleichberechtigt nebeneinander, sind Grundlage und erfahren Anreicherung durch Visualisierungswerkzeuge, die in praktischen Applikationen sich gegenseitig befruchtend ineinander greifen sowie sich mit „denktechnologischen“ Fragen der Informatik verzahnen.

Wir wollen deshalb den Gegenstand des symbolischen Rechnens stärker als Symbiose zwischen Mathematik und Computer verstehen und das Wort *Computeralgebra* in diesem Sinne verwenden. Mit Blick auf die zunehmende Kompliziertheit der entstehenden Werkzeuge sind dafür obige Definitionen noch zu erweitern um den Aspekt der

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

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

Im Computeralgebra-Handbuch [7], an dem weltweit über 200 bekannte Fachleute aus den verschiedensten Bereichen der Computeralgebra mitgearbeitet haben, wird das eigene Fachgebiet etwas ausführlicher wie folgt definiert:

> *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 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 findet die Computeralgebra zunehmend ihren eigenen Platz und nimmt wichtige Entwicklungsimpulse aus beiden Gebieten auf. So mag es nicht verwundern, dass die großen Durchbrüche der letzten Jahre sowohl in der Mathematik als auch in der Informatik die von der Computeralgebra produzierten Werkzeuge wesentlich beeinflusst haben und umgekehrt.

1.8 Computeralgebrasysteme (CAS) – Ein Überblick

Die Anfänge

Dass ein Computer auch zu symbolischem Rechnen fähig ist, war lange vor dem Bau des ersten „echten“ (von-Neumann-)Rechners bekannt. 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 diese Fähigkeiten einer solchen Maschine im Blick. Seine Assistentin, Freundin und Mäzenin, Lady Ada 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.

Die theoretischen Wurzeln der Computeralgebra als einer Disziplin, die algorithmische und abstrakte Algebra im weitesten Sinne mit Methoden und Ansätzen der Computerwissenschaft verbindet, liegen einerseits in der algorithmen-orientierten Mathematik des ausgehenden 19. und beginnenden 20. Jahrhunderts und andererseits in den algorithmischen Methoden der Logik, wie

sie in der ersten Hälfte der 20. Jahrhunderts entwickelt wurden. Die praktischen Anstöße zur Entwicklung computergestützter symbolischer Rechen-Systeme kamen vor allem aus der Physik, der Mathematik und den Ingenieurwissenschaften, wo Modellierungen immer umfangreiche auch symbolische Rechnungen erforderten, die nicht mehr per Hand zu bewältigen waren.

Die ersten Anfänge der Entwicklung von Programmen der Computeralgebra reichen bis in die frühen 50er Jahre und damit die Anfänge des Computerzeitalters zurück. [21] nennt in diesem Zusammenhang Arbeiten aus dem Jahre 1953 von H. G. Kahrimanian [9] und J. Nolan [12] zum analytischen Differenzieren. Kahrimanian schrieb in diesem Rahmen auch ein Assemblerprogramm für die Univac I, das damit als Urvater der CAS gelten kann.

Ende der 50er und Anfang der 60er Jahre wurden am MIT 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, deren Überwindung für die Computeralgebra wesentlich ist (Programme sind auch symbolische Daten), bereits aufgehoben ist.

> CAS der ersten Generation setzen den Fokus auf die Schaffung der sprachlichen Grundlagen und die Sammlung von Implementierungen symbolischer Algorithmen verschiedener Kalküle der Mathematik und angrenzender Naturwissenschaften, vor allem der Physik.

Die Wurzeln dieser ersten allgemeinen Systeme liegen in Versuchen verschiedener Fachwissenschaften, die im jeweiligen Kalkül anfallenden umfangreichen symbolischen Rechnungen einem Computer als Hifsmittel zu übertragen. Schwardmann [17] 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. [21] enthält einen detaillierteren Überblick über die Entwicklungen und Systeme jener Zeit. Hulzen nennt insbesondere das System Mathlab-68 von C. Engelman[2], das in den Jahren 1968–71 eine ganze Reihe symbolischer Verfahren zum Differenzieren, zur Polynomfaktorisierung, unbestimmten Integration, Laplace-Transformationen und zum Lösen von linearen Differentialgleichungen implementierte. Sein Design hatte großen Einfluss auf die damals entstehenden ersten Systeme allgemeiner Ausrichtung.

Diese Systeme, die nicht (nur) für spezielle Anforderungen eines Faches konzipiert wurden, basieren auf LISP und sind seit Mitte der 60er Jahre im Einsatz. REDUCE wurde von A. Hearn seit 1966 aus einem System für Berechnungen in der Hochenergiephysik (Feynman-Diagramme, Wirkungsquerschnitte) entwickelt und verwendete eine kompakte distributive Darstellung von Polynomen in allgemeinen Kernen als Listen.

MACSYMA entstand im Zusammenhang mit Untersuchungen zur künstlichen Intelligenz im Rahmen des DARPA-Programms und setzte die Entwicklungen am MIT von Engelman, Martin und Moses fort. Mit diesem System wurden die Erfahrungen von Mathlab-68 aufgenommen, eine ganze Reihe algorithmischer Unzulänglichkeiten bereinigt und Verbesserungen implementiert. Es verwendete verschiedene interne Darstellungen, um unterschiedliche Anforderungen adäquat zu bedienen. Im Mai 1972 wurde das System im ARPA-Netzwerk auch online verfügbar gemacht. Mit der Infrastruktur der MACSYMA-Nutzerkonferenzen spielte es eine zentrale Rolle in der Entwicklung und Nutzung symbolischer Computermethoden in Wissenschaft und Ingenieurwesen jener Zeit.

Auf Grabmeier [6] geht für diese Systeme die Bezeichnung *Allzwecksysteme der ersten Generation* zurück, der auch darauf hinweist, dass die damaligen programmiertechnischen Restriktionen (Lochstreifen, Stapelbetrieb) für symbolische Rechnungen, die aus noch darzulegenden Gründen meist stärker dialogorientiert sind, denkbar ungeeignete Bedingungen boten.

[2] Engelman's Vision bereits damals: „The construction of a prototype system which could function as a mathematical aid to a scientist in his daily work. We would like the program to be so accessible that the scientist would think of it as a tireless slave to perform his routine mechanical computations."

CAS der zweiten Generation

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 und ein allgemeines Datenmodell für symbolische Ausdrücke unterstützt, jedoch keine Datentypen (im Sinne anderer höherer Programmiersprachen) kennt, sondern es prinzipiell erlaubt, alle im Kernbereich, der *ersten Ebene*, implementierten symbolischen Algorithmen mit allen möglichen Daten zu kombinieren.

Weitergehende Konzepte der Informatik wie insbesondere Typisierung werden nur rudimentär 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

Mathematica entwickelte sich aus dem von C. A. Cole und S. Wolfram Ende der 70er Jahre entworfenen System SMP [3], das wiederum aus der Notwendigkeit geboren wurde, komplizierte algebraische Manipulationen in bestimmten Bereichen der theoretischen Physik effektiv und zuverlässig auszuführen. Im Jahr 1988 wurde schließlich die Version 1.0 von Mathematica auf den Markt gebracht. Es war das erste System, dessen Kern unmittelbar in der sich zu dieser Zeit im Compilerbereich durchsetzenden Sprache C geschrieben wurde und zugleich das erste moderne Desktop-CAS. Im Handbuch schreibt Steven Wolfram dazu: „Seit den 1960er Jahren gab es viele verschiedene Pakete für numerische, algebraische, graphische und andere Aufgaben. Das visionäre Konzept von Mathematica war es, ein System zu schaffen, in welchem all diese Aspekte des technischen Rechnens auf kohärente und einheitliche Weise verfügbar sind.“

Dieser Anspruch, alle wichtigen Bereiche des technischen Rechnens durch eigene Entwicklungen auf hohem Niveau abzudecken, prägt den Weg von Mathematica bis heute. Steven Wolfram bezeichnet das System im Handbuch als „the world's only fully integrated environment for technical computing“.

Jedes der großen CA-Systeme steht heute vor der Frage, wie sich die Mittel für die weitere Entwicklung allokieren lassen. Steven Wolfram setzte dabei frühzeitig auf eine eigene Firma jenseits einer engeren universitären Einbindung, um den erforderlichen cash flow unabhängig von der Konjunktur aktueller Förderprogramme zu sichern. Mathematica spielte damit eine Vorreiterrolle in der konsequenten Vermarktung von Software aus dem symbolischen Bereich, was bis heute in einer restriktiven Lizenzpolitik der speziell für die Weiterentwicklung und den Vertrieb gegründeten Firma Wolfram Research, Inc. zum Ausdruck kommt. Der Erfolg dieses Ansatzes zeigte sich besonders mit den Versionen 3.0 (Sommer 1996) und 4.0 (Frühjahr 1999), die Mathematica in die vorderste Front der „Großen“ gebracht haben. Wolfram Research hat um sein Flaggschiff herum inzwischen eine ganze Infrastruktur mit Webportalen, Nutzerschulungen, Entwicklerkonferenzen sowie Büchern, Zeitschriften und sogar einem eigenen Verlag Wolfram Media aufgebaut, die Mathematica über das eigentliche Softwareprodukt hinaus attraktiv machen. Eingeschlossen in dieses Engagement sind Plattformen wie das *Wolfram Library Archive* (`http://library.wolfram.com`), über welches verschiedenste von Nutzern entwickelte und bereitgestellte Mathematica-Pakete freizügig zugänglich sind, oder das Engagement für die

Online-Enzyklopädien *Wolfram MathWorld* (`http://mathworld.wolfram.com`) und *Eric Weisstein's World of Science* (`http://scienceworld.wolfram.com`).

Trotz dieses Engagements im Geiste des Open-Source-Gedankens, der ein wichtiger Aspekt der Sicherung einer freizügig zugänglichen wissenschaftlichen Infrastruktur ist, werden die Aktivitäten von Steven Wolfram, ähnlich derer von Bill Gates, von der weltweiten Gemeinschaft der Computeralgebraiker teilweise mit großen Vorbehalten verfolgt.

Mit der 2007 herausgegebenen Version 6 wurden wesentliche Teile des Systems überarbeitet, insbesondere ein neues Grafikformat mit stärker interaktiven Möglichkeiten eingeführt und erste Schritte hin zu web- und gridfähigen MATHEMATICA-Versionen gegangen. Auch das Hilfesystem wurde vollkommen überarbeitet und um eine integrierte Online-Komponente ergänzt, über die auch auf aktuelle Datenbestände aus dem Wirtschafts- und Finanzbereich zugegriffen werden kann.

Mit späteren Versionen wurde das konsistente Management dieser weltweit verteilten Ressourcen weiter perfektioniert. Neben Daten und Hilfesystem auf dem eigenen Computer werden zentralisierte Datenbestände gepflegt und laufend aktualisiert, auf die über integrierte Schnittstellen unmittelbar zugegriffen werden kann, so dass sich die Grenzen zwischen lokal gehaltenen Informationen und solchen aus anderen (für den jeweiligen Nutzer zugänglichen) Quellen zunehmend verwischen.

Mit *Wolfram Alpha* (`http://www.wolframalpha.com`) ist Wolfram Research Vorreiter in einer speziellen neuen Sparte von *Software as a Service*, mit der Suchmöglichkeiten wie Google, enzyklopädisches Wissen wie MathWorld oder ScienceWorld und abrufbares algorithmisches Wissen eines CAS vernetzt werden.

Maple

Ähnliche Überlegungen des „Downsizing“ der bis Ende der 1970er Jahre 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 beschränkten 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 MACSYMA und REDUCE 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 der weltweite Vertrieb einer solchen Software, der Support einer Nutzergemeinde sowie die Portierung auf immer neue Rechnergenerationen stellen, die Möglichkeiten einer akademischen Anbindung sprengen und unter heutigen Bedingungen nur über eine Software-Firma stabil zu gewährleisten 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 universitäre Gruppe um K. Geddes und G. Labahn an der University of Waterloo, Ontario (Kanada), über ein ausgebautes akademisches Hinterland verfügt, das ein arbeitsteiliges Vorgehen in der softwaretechnischen und algorithmischen Weiterentwicklung des Systems ermöglicht. MAPLEs Internetportal ist unter `http://www.maplesoft.com` zu erreichen, so dass in Fachkreisen der Name „MapleSoft“ oft auch mit der Firma assoziiert wurde. Im Jahr 2002 gab deshalb Waterloo Maple das als den nunmehr offiziellen Firmennamen (its primary business name) bekannt.

Im Gegensatz zu MATHEMATICA verfolgt MAPLE eine stärker kooperative Politik auch mit anderen Softwarefirmen, um einerseits fremdes Know how für MAPLE verfügbar zu machen (etwa die Numerik-Bibliotheken von NAG, der Numerical Algorithms Group `http://www.nag.co.uk`, mit der Maple im Sommer 1998 eine strategische Allianz geschlossen hat).

In den 90er Jahren wurden MAPLE und MATHEMATICA, die bis dahin nur in mehr oder weniger

experimentellen Fassungen 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 Grafikmoduln. Sie besitzen damit heute in der Regel eine ausgereifte Benutzeroberfläche, sind gut dokumentiert und auf den verschiedensten Plattformen verfügbar. 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.

Derive und CAS in der Schule

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 muSIMP-79 wurde darauf eine überzeugende Antwort gefunden [16, 19]. 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 folgenden Jahren durch die rasante Erweiterung der Hardware-Ressourcen von Arbeitsplatzrechnern dieser Anwendungsbereich auch von den anderen CAS zunehmend erobert wurde, konzentrierte sich die Firma auf das Taschenrechner-Geschäft und entwickelte in Zusammenarbeit mit HP und TI zwei interessante Kleinstrechner mit symbolischen Fähigkeiten, den HP-486X (1991) und den TI-92 (1995).

Zugleich war DERIVE viele Jahre ein Produkt, das mit großem Erfolg im schulischen Bereich eingesetzt wurde. In Österreich hatte man sich frühzeitig für eine landesweite Schullizenz des Systems entschieden und DERIVE flächendeckend im Mathematikunterricht zum Einsatz gebracht. Weitergehende Informationen finden sich im „Austrian Center for Didactics of Computer Algebra“[3]. In Deutschland wird Computeralgebra in Schulen heute vor allem über CAS-fähige Taschenrechner der Firmen Texas Instruments und Casio eingesetzt.

Allerdings gibt es sehr konträre Diskussionen über die Vor- und Nachteile eines solchen technologiegestützten Unterrichts, welche durch den Druck auf die zeitlichen und gestalterischen Freiräume der Schulen im Schlepptau leerer öffentlicher Kassen noch eine ganz spezielle Note erhalten. Entsprechende didaktische Konzepte gehen von einem T^-/T^+-Ansatz aus, in welchem Bereiche festgelegt sind, die ohne bzw. mit Technologie zu behandeln sind. Zugleich werden spezifische Fertigkeiten benannt, welche sich Schüler auch jenseits des unmittelbaren Computereinsatzes für „technologiebasiertes Denken“ neu aneignen müssen. Schließlich wird deutlich benannt, dass CAS-Einsatz auch einen anderen Mathematik-Unterricht erfordert, in welchem projekthafte und explorative Elemente gegenüber der heute üblichen starken Betonung vor allem algorithmischer Fertigkeiten einen deutlich größeren Stellenwert einnehmen werden – eine Herausforderung an Schüler *und* Lehrer.

In Sachsen wurde mit den 2004 eingeführten neuen Lehrplänen auch der Einsatz von CAS und DGS (dynamischer Geometrie-Software) im Gymnasium ab Klasse 8 verbindlich geregelt und umgesetzt.

Um auf diesem Markt (dessen wirtschaftliche Potenzen die des gesamten akademischen Bereichs um Größenordnungen übersteigen) erfolgreicher agieren zu können, wurde DERIVE im August 1999 von Texas Instruments aufgekauft und bildete die Basis für die Software auf den verschiedenen Handhelds von TI mit CAS-Fähigkeiten. Wie weit solche Produkte mit Laptops oder Smartphones, welche die volle Leistungsfähigkeit „großer“ Systeme anbieten, konkurrieren können, wird

[3] `http://www.acdca.ac.at`

die (nahe) Zukunft erweisen. Im Sommer 2006 hat TI die Weiterentwicklung von DERIVE als eigenständigem CAS eingestellt und konzentriert seither seine Kräfte auf neue Taschenrechnergenerationen.

Entwicklungen der 90er Jahre – MuPAD und Magma

Für jedes der bisher betrachteten großen CAS lässt sich der Weg bis zu einem kleinen System spezieller Ausrichtung zur effizienten Ausführung umfangreicher symbolischer Rechnungen in einem naturwissenschaftlichen Spezialgebiet zurückverfolgen. Solche

kleinen Systeme sehr spezieller Kompetenz,

welche einzelne Kalküle in der Physik wie etwa der Hochenergiephysik (SCHOONSHIP, FORM), Himmelsmechanik (CAMAL), der allgemeinen Relativitätstheorie (SHEEP, STENSOR) oder in der Mathematik wie etwa der Gruppentheorie (GAP, CAYLEY), der Zahlentheorie (PARI, KANT, SIMATH) oder der algebraischen Geometrie (Macaulay, CoCoA, GB, Singular) implementieren, gibt es auch heute viele.

Diese Systeme sind wichtige Werkzeuge für den Wissenschaftsbetrieb und stellen – ähnlich der Fachliteratur – die algorithmische und implementatorische Basis für die im jeweiligen Fachgebiet verfügbare Software dar. Wie auch sonst in der Wissenschaft üblich werden diese Werkzeuge arbeitsteilig gemeinsam entwickelt und stehen in der Regel – wenigstens innerhalb der jeweiligen Community – weitgehend freizügig zur Verfügung. Sie sind allerdings, im Sinne unserer Klassifikation, eher den CAS der ersten Generation zuzurechnen, auch wenn die verfügbaren interaktiven Möglichkeiten heute deutlich andere sind als in den 60er Jahren.

Die Grenze zu einem System der zweiten Generation wird in der Regel dort überschritten, wo die Algorithmen des eigenen Fachgebiets fremde algorithmische Kompetenz benötigen. So müssen etwa Systeme zum Lösen polynomialer Gleichungssysteme auch Polynome faktorisieren können, was deutlich jenseits der reinen Polynomarithmetik liegt, die allein ausreicht, um etwa den Gröbner-Algorithmus zu implementieren. Ähnliche Anforderungen noch komplexerer Natur entstehen beim Lösen von Differentialgleichungen.

An der Stelle ergibt sich die Frage, ob es lohnt, eigene Implementierungen dieser Algorithmen zu entwickeln, oder es doch besser ist, sich einem der großen bereits existierenden CAS-Projekte anzuschließen und dessen Implementierung der benötigten Algorithmen zu nutzen. Die Antwort fällt nicht automatisch zugunsten der zweiten Variante aus, denn diese hat zwei Nachteile:

1. Der bisher geschriebene Code für das spezielle Fachgebiet ist, wenn überhaupt, nur nach umfangreichen Anpassungen in der neuen Umgebung nutzbar. Neuimplementierungen im neuen Target-CAS lassen sich meist nicht vermeiden.

2. Derartige Neuimplementierungen lassen sich im Kontext eines CAS allgemeiner Ausrichtung oft nicht ausreichend optimieren.

Andererseits erfordern CAS allgemeiner Ausrichtung einen hohen Entwicklungsaufwand (man rechnet mit mehreren hundert Mannjahren), so dass es – wenigstens noch bis in die jüngste Zeit – keine leistungsfähigen neuen CAS gibt, die wirklich „von der Pike auf neu als CAS“ entwickelt worden sind. Neuentwicklungen allgemeiner Ausrichtung sind nur dort möglich, wo einerseits ein Fundament besteht und andererseits die nötige Manpower für die rasche Ausweitung dieses Fundaments organisiert werden kann.

Das galt auch für 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 folgenden Jahren unter aktiver Beteiligung einer großen Zahl von Studenten, Diplomanden und Doktoranden intensiv vorangetrieben worden ist. Der fachliche Hintergrund im Bereich der Differentialgleichungen ließ es zweckmäßig erscheinen, MuPAD von Anfang an als System allgemeiner

Ausrichtung zu konzipieren. Sein grundlegendes Design orientierte sich an MAPLE, jedoch bereichert um einige moderne Software-Konzepte (Parallelverarbeitung, Objektorientierung), die bis dahin im Bereich des symbolischen Rechnens aus Gründen, die im nächsten Kapitel noch genauer dargelegt werden, kaum Verbreitung gefunden hatten. Mit den speziellen Designmöglichkeiten, die sich aus solchen Konzepten ergeben, gehört MUPAD bereits zu den CAS der dritten Generation.

Im Laufe der 90er Jahre entwickelte sich MUPAD, nicht zuletzt dank der freizügigen Zugangsmöglichkeiten, gerade für Studenten zu einer interessanten Alternative zu den stärker kommerziell orientierten „großen M“. Auch hier stellte 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 1996 übernahm deshalb die eigens dafür gegründete Firma *SciFace* einen Teil dieser Aufgaben insbesondere im software-technischen Bereich. Der Schwerpunkt lag auf der Entwicklung des Grafik-Teils sowie einer besseren Notebook-Oberfläche.

Damit verbunden war die kommerzielle Vermarktung von MUPAD, die zu jener Zeit eine sehr kontroverse Debatte in der deutschen Gemeinde der Computeralgebraiker auslöste. Schließlich waren die bisherigen Entwicklungen zu einem großen Teil mit öffentlichen Geldern finanziert worden. Allerdings ließen die mit solcher Forschungsförderung einher gehenden Refinanzierungs-Zwänge der Paderborner Gruppe keine andere Wahl, wenn sie nicht entscheidendes (immer an konkrete Personen gebundenes) Know how verlieren wollte. Mit einer nach wie vor kostenfrei verfügbaren Lightversion von MUPAD wurde versucht, den Forderungen aus dem akademischen Bereich Rechnung zu tragen. Für eine nachhaltige Etablierung eines solchen Ansatzes wäre es allerdings erforderlich gewesen, dass sich die europäische akademische Öffentlichkeit stärker an der weiteren Entwicklung von MUPAD beteiligt und nicht nur die kostenlose Offerte dankend in Anspruch nimmt. In der folgenden Zeit stellte sich heraus, dass eine solche Erwartung keine Basis hatte.

Mit der Emeritierung von Prof. Fuchssteiner Ende 2005 wurde auch die universitäre Gruppe abgewickelt, so dass nun die Last der weiteren Entwicklung des CAS vollständig auf den Schultern der kleinen Firma *SciFace* lag. Damit wurden Fragen der Sicherung eines ausreichenden cash flow essenziell für das Überleben des Systems und für die Fortschreibung der bisher aufgewendeten Forschungs- und Entwicklungsanstrengungen insgesamt. Im März 2006 schloss MUPAD Pro 4.0 eine Entwicklung ab, in der die bis dahin für die Windows-Version von Microsoft lizensierte Plattform durch QT-basierte Eigenentwicklungen der Notebook- und Grafik-Technologien ersetzt wurde. Damit standen nun einerseits für alle unterstützten Betriebssysteme (Windows, Linux, Mac OS X) funktional weitgehend identische Oberflächen zur Verfügung und andererseits vereinfachte sich die weitere Entwicklung für die verschiedenen Plattformen. Dabei wurde das Grafiksystem vollkommen neu implementiert, womit MUPAD in dieser Kategorie zeitweise die Führungsrolle erreicht hatte[4]. Eine kostenlose Lightversion wurde nicht mehr angeboten.

Trotz aller Anstrengungen blieb die finanzielle Situation von Sciface angespannt. Im Herbst 2008 wurde Sciface schließlich von MathWorks[5] aufgekauft und MUPAD als *Symbolic Math Toolbox* in deren stärker auf numerische Rechnungen spezialisierte Flaggschiff MATLAB integriert[6]. Mathworks wechselte damit zugleich seine Strategie. Während Mathworks vorher auf eine Kooperation mit MAPLE und damit eine Outsourcing-Strategie setzte, wurde durch den Einstieg bei Sciface – ähnlich wie TI bei DERIVE – symbolische Kompetenz in das eigene Portfolio aufgenommen und so Kompetenz im Bereich des symbolischen Rechnens im eigenen Haus aufgebaut. Damit endete zugleich die eigenständige Entwicklung von MUPAD.

MUPAD blieb lange ein weitgehend singuläres Paket im MATLAB-Universum, da es schwierig ist, ein symbolisches System in ein System zu integrieren, welches in seinem innersten Kern bereits auf numerische Rechnungen und Verfahren getrimmt ist. 2017 wird mit dem neuen Konzept der *Live Scripts* ein modernes Notebook-Konzept in MATLAB eingeführt. Im Zuge dieses Redesigns erfolgt auch eine engere Integration der symblischen Fähigkeiten in den MATLAB-Kern.

MAGMA ist ein zweites System, dessen Entwicklergruppe nicht in Amerikas residiert und welches

[4] Grafiken ähnlicher Qualität kamen erst mit MATHEMATICA 6 im Sommer 2007 auf den Markt

[5] `http://www.mathworks.com`

[6] `http://www.mathworks.com/discovery/mupad.html`

in den 90er Jahren an Bedeutung gewann. Es hat seine Wurzeln in der Zahlentheorie und abstrakten Algebra, die bis zum System CAYLEY von John Cannon und die 70er Jahre zurückreichen, und wird von einer Gruppe an der School of Mathematics and Statistics der University of Sydney entwickelt. Es integriert ebenfalls moderne Aspekte der Informatik wie higher order typing. Auch die MAGMA-Gruppe kann sich nicht vollständig aus öffentlichen Mitteln refinanzieren und hat ein Lizenzmodell entwickelt, mit welchem die wissenschaftlichen Einrichtungen, die das System nutzen, an dessen Refinanzierung beteiligt werden. Die Rückläufe werden vollständig darauf verwendet, um – ähnlich Wolfram Research für MATHEMATICA – Entwickler mit vielversprechenden algorithmischen Ideen für eine begrenzte Zeit nach Australien einladen, damit sie ihre Ideen in MAGMA implementieren. Entwickler von MAGMA sowie Mitarbeiter und Studenten von Einrichtungen, die MAGMA lizensiert haben, können MAGMA unter besonderen Lizenz-Bedingungen freizügig nutzen. Die finanzielle Beteiligung wird also davon abhängig gemacht, in welchem Verhältnis jeweils auch das institutionelle Geben und Nehmen stehen.

Computeralgebra – ein schwieriges Marktsegment für Software

Generell war im Laufe der 90er Jahre zu verzeichnen, dass neben der algorithmischen Leistungsfähigkeit eine ausgewogene Lizenzpolitik, mit der die richtige Balance zwischen Refinanzierungserfordernissen einerseits und freizügigen Zugangsmöglichkeiten andererseits gefunden werden kann, für das weitere Schicksal der einzelnen Systeme zunehmend an Bedeutung gewann.

So gelang es weder MACSYMA noch REDUCE, mit den „großen M“ in Bezug auf Oberfläche sowie Vertriebs- und Vermarktungsaufwand Schritt zu halten. Trotz exzellenter algorithmischer Fähigkeiten und einer langjährig gewachsenen Nutzergemeinde ging deshalb die Bedeutung beider Systeme im Laufe der 90er Jahre deutlich zurück.

Besonders interessant ist in diesem Zusammenhang das Schicksal von MACSYMA. Der Grundstein für das System wurde, wie bereits ausgeführt, in den 60er Jahren am MIT im Rahmen eines DARPA-Projekts gelegt. Es entstand ein wirklich gutes System auf LISP-Basis, das in den 70er Jahren weite internationale Verbreitung in akademischen Kreisen fand und eng mit der Geschichte von Unix und dem ArpaNet verbunden war. Um 1980 herum war MACSYMA das weltweit beste symbolisch-numerisch-grafische Softwaresystem und das Flaggschiff der CA-Gemeinde.

Im Jahre 1982 lizensierte das MIT MACSYMA an seine Spin-off-Company Symbolics Inc., womit die kommerzielle Seite des Lebens dieses Systems begann. Wegen der restriktiven amerikanischen Gesetzgebung konnten aber (glücklicherweise) nicht alle Rechte an die Firma übertragen werden, so dass neben der kommerziellen Variante auch nichtkommerzielle Versionen unter teilweise leicht abgeänderten Namen (Vaxima, Maxima) kursierten und von interessierten Wissenschaftlern weiterentwickelt wurden.

Ende der 80er Jahre geriet Symbolics Inc. in zunehmende kommerzielle Schwierigkeiten und MACSYMA ging in den Jahren 1988–92 zunächst gemeinsam mit Symbolics unter. Was dies für die Nutzer eines solchen Systems insbesondere im akademischen Bereich bedeutet, muss nicht im Detail ausgemalt werden. Vielfältige Programme und Lehrmaterialien, die auf der Basis dieses Systems erstellt wurden, werden mit dem Wechsel auf neue Hardware-Plattformen, auf denen MACSYMA nicht mehr zur Verfügung steht, unbrauchbar und damit zunehmend wertlos.

Der Druck der Nutzergemeinde und vielfältige Versprechen auf Unterstützung bewogen Richard Petti im Jahr 1992, die Überreste von MACSYMA aufzukaufen und mit der neu gegründeten Firma Macsyma Inc. wieder auf den Markt zu bringen. In zwei größeren Benchmark-Tests für CAS schnitt MACSYMA sehr erfolgreich ab und im CA-Handbuch [7, S. 283 ff.] wird es noch gelobt. Allerdings war zum Erscheinungstermin des Buches (Anfang 2003) die neue Firma ebenfalls pleite und unter der dort noch zitierten Webadresse `http://www.macsyma.com` ein Online-Shop zweifelhafter Qualität zu finden. Details zu dieser Geschichte sind in *The Macsyma Saga*[7] von Richard Petti zu finden.

[7] `http://maxima-project.org/wiki/index.php?title=The_Macsyma_Saga`

In all den Jahren wurde jedoch auch eine freien Version von MACSYMA unter dem Titel MAXIMA von William Schelter weiterentwickelt, so dass die noch verbliebenen MACSYMA-Anhänger nicht ganz mit leeren Händen dastanden. Mit dem Tod von Bill Schelter im Jahre 2001 stand die kleine MACSYMA-Nutzergemeinde erneut vor einer großen Herausforderung, die sie diesmal ganz im Geiste der GNU-Traditionen löste: Maxima ist das erste große CAS, das unter der GPL als Open-Source-Projekt weiter vorangetrieben wurde und wird, siehe `http://maxima.sourceforge.net`. Seit September 2004 steht eine Version für Windows, Linux und inzwischen auch Mac-OS zur Verfügung.

Einen ähnlichen Weg gingen die Entwickler von REDUCE nach einer mehrjährigen Phase der Inaktivität Ende 2008 und haben das ganze Projekt unter der BSD-Lizenz zur Verfügung gestellt.

Es existieren einige weitere Open-Source-Projekte zur Computeralgebra, jedoch blieb deren Leistungsfähigkeit bisher ebenso beschränkt wie die der meisten firmenbasierten Versuche, mit Neuimplementierungen in das Marktsegment einzusteigen. Offensichtlich sind die aufzubringenden Entwicklungsleistungen im algorithmischen Bereich für ein einigermaßen interessantes CAS allgemeiner Ausrichtung so hoch, dass sie sowohl die Fähigkeiten zur Kräftebündelung heutiger Open Source Projekte als auch die Risikobereitschaft selbst großer kommerzieller Firmen übersteigen. Erfolgversprechende Ansätze sind deshalb nur denkbar

- auf der Basis eines der großen Systeme, dessen Quellen unter die GPL gestellt werden („Netscape-Modell“),
- wenn eine große Software-Firma mit langem Atem entsprechende Vorlaufforschung in ihren eigenen Labs finanziert oder
- auf der Basis eines dichten weltweiten Netzwerks von Computeralgebraikern, welche in der Lage sind, die bisher implementierten Bausteine zusammenzutragen und zu integrieren.

Für alle drei Ansätze gibt es Beispiele. Den **ersten Ansatz** hatten wir mit MAXIMA bereits belegt.

Für den **zweiten Ansatz** möge IBM als Beispiel dienen, die sich als eine der großen Softwarefirmen seit den Anfangstagen der Computeralgebra mit eigenen Aktivitäten an den wichtigsten Entwicklungen beteiligt hat. Das betrifft insbesondere nach einigen Sprachentwicklungen in den 60er Jahren das System SCRATCHPAD, mit dem im *IBM Watson Research Center at Yorktown, NY,* seit den 70er Jahren diese Untersuchungen fortgesetzt wurden. SCRATCHPAD verwendete als damals einziges System im Design ein strenges Typkonzept. 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 Exzellenz (Hulzen widmet in [21] dem Konzept breiten Raum) keine weite Verbreitung. Auch mit der ersten offiziellen Version von AXIOM im Jahre 1991 änderte sich daran wenig. Das mag IBM 1992 bewogen haben, im Zuge einer generellen „Flurbereinigung“ des Firmenprofils die Rechte an der aus markenrechtlichen Gründen in AXIOM umbenannten Software der Firma NAG Ltd. zu übertragen, eine Non-Profit-Firma, welche sich bereits auf dem Gebiet wissenschaftlicher Software ausgewiesen hatte. Um eine Schnittstelle zu ihrem Hauptprodukt, der NAG Numerik-Bibliothek, erweitert, wurde AXIOM in den folgenden Jahren auf eine Vielzahl von Plattformen portiert und auch der zugehörige Standalone-Compiler ALDOR weiterentwickelt. Im Sommer 1998 jedoch straffte NAG seine Produktlinien und begann, sich im Computeralgebrabereich strategisch auf MAPLE zu orientieren. AXIOM und ALDOR werden von NAG seit 2001 nicht mehr unterstützt und vertrieben. Damit endete vorerst die 30-jährige Geschichte eines weiteren wichtigen Computeralgebra-Projekts.

Allerdings haben IBM und NAG den Code für den Open-Source-Bereich freigegeben und die Gemeinde der Nutzer von AXIOM das Schicksal des Systems selbst in die Hand genommen. Nach einer Phase intensiver Aufbereitung des Codes für eine solche Distributionsform steht AXIOM seit August 2003 als Open-Source-Projekt frei zur Verfügung, siehe `http://www.axiom-developer.org`. 2012 gab es es mit FriCAS[8] und Open-Axiom[9] zwei Forks, da es zu allen Zeiten schwierig

[8] `http://fricas.sourceforge.net`
[9] `http://www.open-axiom.org`

System	aktuelle Version	Webseite	Preis Einzellizenz	Preis Stud.-version
Axiom	1.3.2	`fricas.sourceforge.net/`	Open Source, BSD	
GAP	4.8.8	`www.gap-system.org`	Open Source, GPL v.2	
Magma	2.23-5	`magma.maths.usyd.edu.au`	1020 €	–
Maple	2017	`www.maplesoft.com`	1250 €	100 €
Mathematica	11	`www.wolfram.com`	1455 €	145 €
Maxima	5.41.0	`maxima.sourceforge.net`	Open Source, GPL	
Matlab/MuPAD	2017b/8.0	`www.mathworks.com`	700 €[11]	55 €
Reduce	Sept. 2016	`www.reduce-algebra.com`	Open Source, BSD like	
Sage	8.0	`www.sagemath.org`	Open Source, GPL	

Wichtige Computeralgebrasysteme allgemeiner Ausrichtung im Überblick
(Stand Oktober 2017)

war, im Dschungel existierender LISP-Dialekte eine konsistente Entwicklungsumgebung aus Hard- und Software für Axiom zu formulieren und damit das CAS auf dem eigenen Rechner zum Laufen zu bringen. Solche Forks schwächen die Basis der verfügbaren Entwicklungsressourcen weiter. 2017 wird nur noch FriCAS weiterentwickelt. Der strenge Typkonzepte unterstützende Compiler war bereits vorher unter dem Namen Aldor[10] unter die Fittiche der Nutzergemeinde genommen worden.

Der **dritte Ansatz** – Integration vorhandener kleiner Systeme aus verschiedenen Spezialgebieten – hat mit dem Sage-Projekt `http://www.sagemath.org` im Rahmen der europäischen Forschungskooperation in den letzten Jahren eine neue Qualität erreicht. Nach Vorarbeiten zur Vereinheitlichung entsprechender Sprachkonzepte (MathML, OpenMath) für Strukturierung und Austausch mathematischer Informationen und Experimenten zur Kopplung der Ein- und Ausgabekanäle an Open Source Textverarbeitungssysteme wie Emacs oder LaTeX (TeXmacs, MathML-Schnittstellen) wird mit der Sage-Distribution nun auch eine Sammlung spezialisierter Open Source CAS aus dem akademischen Bereich gepflegt, die über eine Python basierte Intermediärsoftware zu einem Gesamtsystem zusammengeschaltet werden. Der Zugriff auf das Gesamtsystem erfolgt über die Kommandozeile oder über eine (spartanische) Notebook-Oberfläche in einem dafür aufgesetzten Webserver. Über diese Oberfläche hat man Zugriff auf das darunter liegende System und kann komplexe Rechnungen organisieren, an denen mehrere der gebündelten CAS beteiligt sind. Die Bezeichnung *Open Source Mathematics Software* zeigt den weitergehenden Anspruch in Richtung einer Open Source Computermathematik, der mit diesem Projekt verfolgt wird.

Computeralgebrasysteme der dritten Generation

CAS der dritten Generation sind solche Systeme, die auf der Datenschicht aufsetzend weitere Konzepte der Informatik verwenden, mit denen Aspekte der inneren Struktur der Daten ausgedrückt werden können.

Dafür sind vor allem strenge Typkonzepte des higher order typing (Axiom, Magma) sowie dezentrale Methodenverwaltung nach OO-Prinzipien (MuPAD, Sage) eingesetzt worden. Die Besonderheiten des symbolischen Rechnens führen dazu, dass die schwierigen theoretischen Fragen, welche mit jedem dieser Konzepte verbunden sind, in sehr umfassender Weise beantwortet werden müssen. Solche Fragen und Antworten werden in diesem Kurs allerdings nur eine randständige Rolle spielen, da wir uns auf die Darstellung der Design- und Wirkprinzipien von CAS der zweiten Generation konzentrieren werden.

[10] `http:www.aldor.org`, letzte Aktivitäten 2013.

Eine Vorreiterrolle beim Einsatz von strengen Typ-Konzepten spielte seit Ende der 70er Jahre das IBM-Laboratorium in Yorktown Heights mit der Entwicklung von SCRATCHPAD/AXIOM. Dabei wurden vielfältige Erfahrungen gesammelt, wie CAS aufzubauen sind, wenn ins Zentrum des Designs moderne programmiertechnische Ansätze der Typisierung, Modularisierung und Datenkapselung gesetzt und diese konsequent in einem symbolisch orientierten Kontext realisiert werden. Computeralgebrasysteme bieten hierfür denkbar gute Voraussetzungen, denn ein solches Typsystem ist ja seinerseits symbolischer Natur und hat andererseits einen stark algebraisierten Hintergrund. Sie sollten also prinzipiell gut mit den vorhandenen sprachlichen Mitteln eines CAS der zweiten Generation modelliert werden können. Andere Versuche, Typ-Informationen mit den Mitteln eines CAS der zweiten Generation zu modellieren, wurden mit dem Domain-Konzept in MUPAD vorgelegt. Auch SAGE setzt auf getypte Datenobjekte.

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 ist aus der funktionalen Programmierung gut bekannt, wo ebenfalls nicht nur Typnamen wie in (frühen Versionen von) C++ oder Java, sondern (im Fall des higher order typing) Typausdrücke auf dem Hintergrund einer ganzen Typsprache auszuwerten sind, um an die prototypischen Eigenschaften von Objekten heranzukommen. Ein solches Herangehen auf einem symbolischen Hintergrund wurde von ALDOR, dem aus AXIOM abgespaltenen Sprachkonzept, 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 entsprechende Versuche in den „großen Sprachfamilien“ C, Pascal oder Java noch nicht zu denken war.

[11] Matlab verwendet ein paketorientiertes Lizenzmodell. Hier sind die Kosten für das Grundmodul sowie die Symbolische Toolbox angegeben.

Kapitel 2

Aufbau und Arbeitsweise eines CAS 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.

2.1 CAS als komplexe Softwareprojekte

CAS – Interpreter versus Compiler

Programmiersysteme werden zunächst grob in Interpreter und Compiler unterschieden. CAS haben wir bisher ausschließlich über eine interaktive Oberfläche, also als Interpreter, kennengelernt. Es erhebt sich damit die Frage, ob es gewichtige Gründe gibt, symbolische Systeme gerade so auszulegen.

Interpreter und Compiler sind Programmiersysteme, die dazu dienen, die in einer Hochsprache fixierte Implementierung eines Programms in ein Maschinenprogramm zu übertragen und auf dem Rechner auszuführen.

Beide Arten durchlaufen dabei die Phasen *lexikalische Analyse, syntaktische Analyse* und *semantische Analyse* des Programms. Ein **Interpreter** bringt den aktuellen Befehl nach dieser Prüfung *sofort* zur Ausführung.

Ein **Compiler** führt in einer ersten Phase, der **Übersetzungszeit**, die Analyse des vollständigen Programms aus, übersetzt dieses in eine maschinennahe Sprache und speichert es ab. Er durchläuft dabei die weiteren Phasen *Codegenerierung* und evtl. *Codeoptimierung*. In einer zweiten Phase, zur **Laufzeit**, wird das übersetzte Programm von einem *Lader* zur Abarbeitung in den Hauptspeicher geladen.

Ein Compiler betreibt also einen größeren Aufwand bei der Analyse des Programms, der sich in der Abarbeitungsphase rentieren soll. Das ist nur sinnvoll, wenn dasselbe Programm mit mehreren Datensätzen ausgeführt wird. Das zentrale Paradigma des Compilers ist also das des *Programmflusses*, längs dessen Daten geschleust werden.

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.

Ein **Interpreter** dagegen zeichnet sich durch eine höhere Flexibilität aus, 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. Ein Interpreter ist also dort von Vorteil, wo man verschiedene Daten auf unterschiedliche Weise kombinieren und modifizieren möchte. Das zentrale Paradigma des Interpreters ist also das der *Datenlandschaft*, die durch Anwendung verschiedener Konstruktionselemente erstellt und modifiziert wird.

Als entscheidender Nachteil schlagen längere Rechenzeiten für einzelne Operationen 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.

Eine solche Flexibilität ist eine wichtige Anforderung an ein CAS, wenn es als *Hilfsmittel für geistige Arbeit* eingesetzt werden soll. Dies ist folglich auch der Grund, weshalb alle großen CAS wenigstens in ihrer obersten Ebene Interpreter sind.

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.

Es ist deshalb nur folgerichtig, diese Teile eines CAS in vorübersetzter (und optimierter) Form bereitzustellen. Allerdings trifft dies nicht nur auf Systemteile selbst zu. Auch der Nutzer sollte die Möglichkeit haben, selbstentwickelte Programmteile, die mit mehreren Datensätzen abgearbeitet werden sollen, zu übersetzen, um in den Genuss der genannten Vorteile zu kommen.

Entsprechend dieser Anforderungsspezifikation sind große CAS allgemeiner Ausrichtung, wie übrigens heute alle größeren Interpretersysteme,

top-level interpretiert, low-level compiliert.

Das Drei-Ebenen-Modell

Bei der Übersetzung von Programmteilen gibt es drei Ebenen, die unterschiedliche Anforderungen an die Qualität der Übersetzung stellen:

- Während der Systementwicklung erstellte Übersetzungen, die in Form von Bibliotheken grundlegender Algorithmen 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 (müssen).

Beispiele für die **erste Ebene** sind die Implementierungen der wichtigsten mathematischen Algorithmen, die die Leistungskraft eines CAS bestimmen. Hier wird man besonderen Wert auf den Entwurf geeigneter Datenstrukturen sowie die Effizienz der verwendeten Algorithmen legen, wofür neben entsprechenden programmiertechnischen Kenntnissen auch profunde Kenntnisse aus dem jeweiligen mathematischen Teilgebiet erforderlich sind.

Beispiele für die **zweite Ebene** sind Anwenderentwicklungen für spezielle Zwecke, die auf den vom System bereitgestellten Algorithmen aufsetzen und so das CAS zum Kern einer (mehr oder weniger umfangreichen) speziellen Applikation machen. Solche Applikationen reichen von kleinen Programmen, mit denen man verschiedene Vermutungen an entsprechendem Datenmaterial testen kann, bis hin zu umfangreichen Sammlungen von Funktionen, die Algorithmen aus einem bestimmten Gebiet von Mathematik, Naturwissenschaft oder Technik zusammenstellen.

Beispiele für die **dritte Ebene** sind schließlich zur Laufzeit entworfene Funktionen, die mit umfangreichem Datenmaterial ausgewertet werden müssen, wie es etwa beim Erzeugen einer Grafikausgabe anfällt.

Natürlich sind die Grenzen zwischen den verschiedenen Ebenen fließend. So muss im Systemdesign der ersten Ebene beachtet werden, dass nicht nur eine Flexibilität hin zu komplexeren Algorithmen zu gewährleisten ist, sondern auch eine Flexibilität nach unten, damit das jeweilige CAS möglichst einfach an unterschiedliche Hardware und Betriebssysteme angepasst werden kann. Hierfür ist es sinnvoll, das Prinzip der *virtuellen Maschine* anzuwenden, d. h. Implementierungen komplexerer Algorithmen in einer maschinenunabhängigen Zwischensprache zu erstellen, die dann von leistungsfähigen Werkzeugen plattformabhängig übersetzt und optimiert werden.

> Die einzelnen CAS sind deshalb so aufgebaut, dass in einem Systemkern (unterschiedlichen Umfangs) Sprachelemente und elementare Datenstrukturen eines leistungsfähigen Laufzeitsystems implementiert werden, in dem dann seinerseits komplexere Algorithmen codiert sind.

Auch zwischen der ersten und zweiten Ebene ist eine strenge Abgrenzung nicht möglich, da die Implementierung guter konstruktiver mathematischer Verfahren eine gute Kenntnis des zugehörigen theoretischen Hintergrunds und damit oft eine akademische Einbindung dieser Entwicklungen wünschenswert macht oder gar erfordert. Außerdem werden in der Regel von Nutzern entwickelte besonders leistungsfähige spezielle Applikationen neuen Versionen des jeweiligen CAS hinzugefügt, um sie so einem weiteren Nutzerkreis zugänglich zu machen. In beiden Fällen ist es denkbar und auch schon eingetreten, dass auf diese Weise entstandene Programmstücke aus Performance-Gründen Auswirkungen auf das Design tieferliegender Systemteile haben.

CAS als komplexe Softwareprojekte – die kooperative Dimension

Die im letzten Abschnitt besprochene 3-Ebenen-Struktur eines CAS hat eine Entsprechung in der Unterteilung der Personengruppen, welche mit einem solchen CAS zu tun haben, in Systementwickler, (mehrheitlich im akademischen Bereich verankerte) „Power User“ und „normale Nutzer“. Während die erste und dritte Gruppe als Entwickler und Nutzer auch in klassischen Softwareprojekten zu finden sind, spielt die zweite Gruppe sowohl für die Entwicklungsdynamik eines CAS als auch für dessen Akzeptanzbreite eine zentrale Rolle.

Mehr noch, die personellen und inhaltlichen Wurzeln aller großen CAS liegen in dieser zweiten Gruppe, denn sie wurden bis Mitte der 80er Jahre von überschaubaren Personengruppen meist an einzelnen wissenschaftlichen Einrichtungen entwickelt. Für jedes CAS ist es wichtig, solche Verbindungen zu erhalten und weiter zu entwickeln, da neue Entwicklungsanstöße vorwiegend aus

diesem akademischen Umfeld kommen. Nur so kann neuer Sachverstand in die CAS-Entwicklung einfließen und mit dem bereits vorhandenen, „in Bytes gegossenen“ Sachverstand integriert werden. Das Management von CAS-Projekten muss diesem prozesshaften Charakter gerecht werden. Allerdings ist das mit Blick auf den schwer zu überschauenden Kreis von Nutzern und Entwicklern eine deutlich größere Herausforderung als bei klassischen Softwareprodukten.

> Die fruchtbringende Einbeziehung einer großen Gruppe vom eigentlichen Kernentwickler-Team unabhängiger „Power User“ in die weitere Entwicklung eines CAS ist nur möglich, wenn große Teile des Systemdesigns selbst offen liegen und auch die technischen Mittel, die Nutzern und Systementwicklern zur Übersetzung und Optimierung von Quellcode zur Verfügung stehen, in ihrer Leistungsfähigkeit nicht zu stark voneinander differieren. Aus einer solchen Interaktion ergibt sich als weitere Forderung an Design *und* Produktphilosophie, dass Computeralgebrasysteme in den entscheidenen Bereichen **offene Systeme** sein müssen.

Die „Power User“ sind die Quelle einer Vielzahl von Aktivitäten zur programmtechnischen Fixierung mathematischen Know-Hows, aus dem heraus die *mathematischen* Fähigkeiten der großen CAS entstanden sind. An diesen Aktivitäten sind Arbeitsgruppen beteiligt, 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. Natürlich ergeben sich für ein konsistentes Management der Anstrengungen eines solch heterogenen Personenkreises im Umfeld des jeweiligen CAS

> hohe Anforderungen auch im organisatorischen Bereich, die weit über Fragen des reinen Software-Managements hinausgehen.

Die großen CAS haben eher den Charakter von Entwicklungsumgebungen bzw. -werkzeugen, die zu verschiedenen, von den Entwicklern und Softwarefirmen nicht vorhersehbaren Zwecken vor allem im wissenschaftlichen und technischen Bereich eingesetzt werden. Dabei entstand und entsteht eine Vielzahl von Materialien unterschiedlicher Qualität und Ausrichtung, welche von den meisten Autoren – wie in Wissenschaftskreisen üblich – frei zitier- und nachnutzbar zur Verfügung gestellt werden.

Diese zu sammeln und zu systematisieren war schon immer ein Anliegen der weltweiten Gemeinde der Anhänger der verschiedenen CAS. In den letzten Jahren wurden diese frei verfügbaren Sammlungen zunehmend in Portalstrukturen wie etwa `http://www.maplesoft.com` (Maple) oder `http://demonstrations.wolfram.com` (Mathematica) integriert, über welche Nutzer relevante Materialien bereitstellen oder suchen und sich herunterladen können. Auch andere CAS verfügen heute über solche Portale unterschiedlicher Qualität.

2.2 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 Maxima zum Beispiel

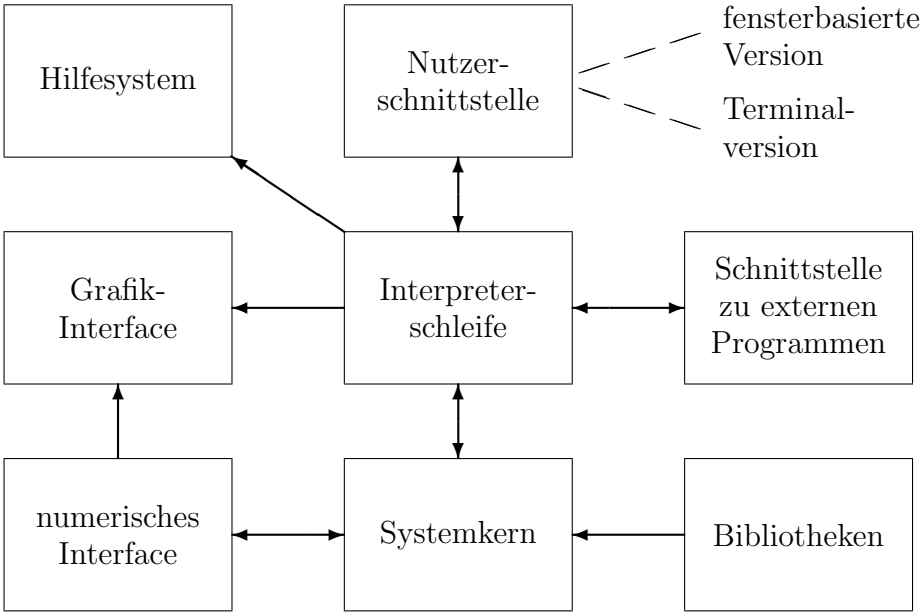


Bild 1: Prinzipieller Aufbau eines Computeralgebra-Systems

- Eingaben wie `?? plot` zur Aktivierung des Hilfesystems oder
- Eingaben wie `plot2d(sin(x),[x,-5,5])` für Grafikausgaben.

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 *Grafik-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.

Ehe wir uns Aufbau und Arbeitsweise des Systemkerns zuwenden, in dem die symbolischen Fähigkeiten des jeweiligen Systems konzentriert sind, wollen wir einen kurzen Blick auf die anderen Komponenten werfen.

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 Grafikmodus erfolgt.

Grafikbasierte Ausgabesysteme sind heute meist in der Lage, *integrierte Dokumente* zu produzieren, die Texte, Ergebnisse von Rechnungen und Grafiken verbinden und in gängigen Formaten (HTML, LaTeX) exportieren können. Eine Vorreiterrolle spielten auf diesem Gebiet die Systeme

MATHEMATICA und MAPLE mit dem Konzept des *Arbeitsblatts*. Hier trafen sich Entwicklungen aus dem Bereich des wissenschaftlichen Publizierens, der Gestaltung interaktiver Oberflächen und Methoden des symbolischen Rechnens, womit sich zugleich vollkommen neue Horizonte in Richtung der (elektronischen) Publikation interaktiver mathematischer Texte eröffnen. Das MATHEMATICA-Frontend ist dabei auch in der MATHEMATICA-Sprache geschrieben, was symbolische Manipulationen auf ganzen Dokumenten mit sprachlichen Mitteln erlaubt, mit denen die Rechnungen selbst organisiert werden. Andere Systeme (MAPLE, MUPAD) verwenden standardisierte Dokument-Architekturen oder verfügen in diesem Bereich nur über rudimentäre Möglichkeiten.

Auch **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 verzeichnen. Dabei wird zunehmend, wie im letzten Kapitel bereits für das Grafik-Interface beschrieben, nicht nur auf entsprechende Ansätze, sondern auch auf bereits fertige Software-Komponenten zurückgegriffen. Hilfesysteme sind meist hierarchisch oder/und nach Schlagworten sortiert, so dass man relativ genaue Kenntnisse benötigt, wie konkrete Kommandos oder Funktionen heißen bzw. an welcher Stelle in der Hierarchie man relevante Informationen findet. Auch hier hat MATHEMATICA eine Vorreiterstellung inne, denn mit Version 7 steht ein lokal-globales Hilfesystem zur Verfügung, das neben der bisherigen lokalen Hilfekomponente auch auf global verteilte, über Webschnittstellen zu erreichende Bestandteile wie etwa das *Wolfram Demonstration Project* `http://demonstrations.wolfram.com` und dessen natürliche Fortsetzung in *Wolfram Alpha* `http://www.wolframalpha.com` setzt.

Obwohl es auch große und leistungsfähige Programmpakete zur Numerik gibt, treiben die CAS die Entwicklung ihres **numerischen Interface** in zwei Richtungen voran. Zum einen ist zu bedenken, dass ein CAS nur dann zu einem nützlichen Instrument, einem computermathematischen Werkzeug in der Hand eines Wissenschaftlers oder Ingenieurs wird, wenn es die volle „compute power“ bereitstellt, die von dieser Klientel im Alltag benötigt wird. Dazu gehört neben symbolischen Rechenfertigkeiten und Visualisierungstools auch die Möglichkeit, ohne weitergehenden Aufwand symbolische Ergebnisse numerisch auszuwerten. Deshalb hat jedes der großen CAS eigene Routinen für numerische Berechnungen etwa von Nullstellen oder bestimmten Integralen „für den Hausgebrauch“. Dabei wird stark von den speziellen Möglichkeiten adaptiver Präzision einer bigfloat-Arithmetik Gebrauch gemacht, die auf der Basis der vorhandenen Langzahlarithmetik leicht implementiert werden kann. Diese Fähigkeiten, die etwa beim Bestimmen nahe beieinander liegender Nullstellen von Polynomen oder beim Berechnen numerischer Näherungswerte hoher Präzision eine Rolle spielen, sind stärker in den Systemkern integriert.

Andererseits ist eine wichtige, wenn nicht gar die wichtigste[1] Anwendung von Computeralgebra die Aufbereitung symbolischer Daten zu deren nachfolgenden numerischen Weiterverarbeitung. Deshalb werden neben Codegeneratoren auch zunehmend **explizite Schnittstellen und Protokolle** entworfen, über welche die Programme mit externem Sachverstand kommunizieren können. Über solche Schnittstellen kann insbesondere mit vorhandenen Numerikbibliotheken kommuniziert werden. Allerdings kann eine solche Schnittstelle umfangreichere Funktionen erfüllen, etwa webbasierte Client-Kommunikation organisieren oder kooperative Prozesse mehrerer CAS oder mehrerer Prozesse eines CAS koordinieren. Auch auf diesem Gebiet nimmt MATHEMATICA mit dem MathLink-Protokoll seit vielen Jahren eine führende Stellung ein.

Anforderungen an das Systemkerndesign

Betrachten wir nun die Anforderungen näher, die beim Design des Systemkerns zu berücksichtigen sind, in dem die uns in diesem Kurs interessierenden symbolischen Rechnungen letztendlich ausgeführt werden. Diese Anforderungen kann man grob folgenden sechs Komplexen zuordnen:

[1] Nach [15] werden moderne CAS zu 90 % zur Generierung effizienten Codes eingesetzt.

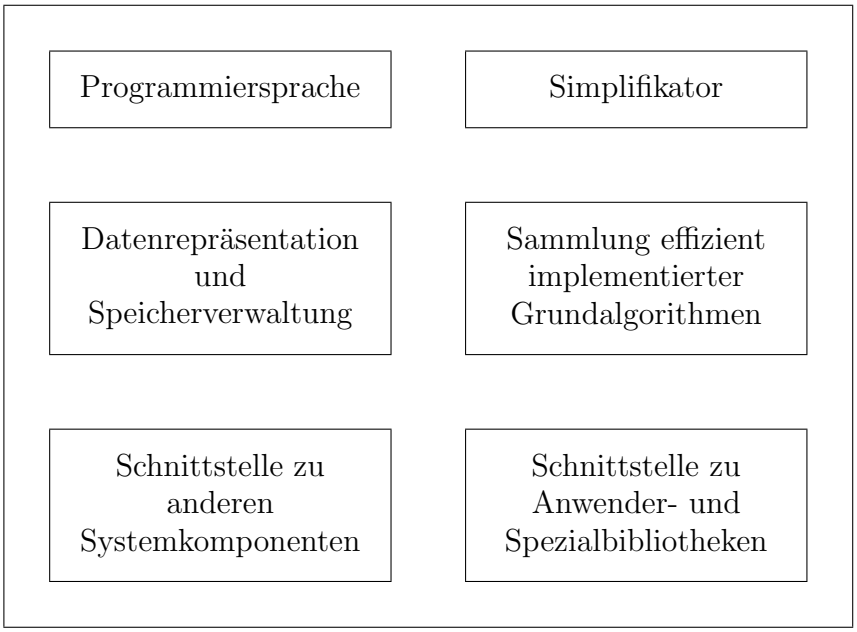


Bild 2: Komponenten des Systemkern-Designs

- Es ist ein Konzept für eine *Datenrepräsentation* zu entwickeln, das es erlaubt, heterogen strukturierte Daten, wie sie typischerweise im symbolischen Rechnen auftreten, mit der notwendigen Flexibilität, aber doch nach einheitlichen Gesichtspunkten zu verwalten und zu verarbeiten. Damit verbunden ist die Frage nach einer entsprechend leistungsfähigen *Speicherverwaltung*.

  Dabei ist zu berücksichtigen, dass symbolische Ausdrücke sehr unterschiedlicher und im Voraus nicht bekannter Größe auftreten, also als *dynamische Datentypen* mit einer ebensolchen *dynamischen* Speicherverwaltung anzulegen sind.

- Es wird eine *Programmiersprache* benötigt, mit der man den Ablauf der Rechnungen steuern kann. Bekanntlich reicht ein kleines Instrumentarium an Befehlskonstrukten aus, um die gängigen Programmablaufkonstrukte (Schleifen, Verzweigungen, Anweisungsverbünde) zu formulieren. Weiterhin sollte die Sprache Methoden des strukturierten Programmierens (Prozeduren und Funktionen, Modularisierung) unterstützen, vermehrt um Instrumente, die sich aus der Spezifik des symbolischen Rechnens ergeben.

- Es 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 effizient im Kern implementierten *Polynomarithmetik* besteht, die in der Lage ist, rationale Ausdrücke umzuformen, und einem (vom Nutzer erweiterbaren) *Simplifikationssystem*, das die Navigation in der transitiven Hülle der dem System bekannten elementaren Umformungsregeln gestattet.

- Es werden *effiziente Implementierungen grundlegender Algorithmen* (Langzahl- und Polynomarithmetik, Rechnen in modularen Bereichen, zahlentheoretische Algorithmen, Faktorisierung von Polynomen, Bigfloat-Arithmetik, Numerikroutinen, Differenzieren, Integrieren, Rechnen mit Reihen und Summen, Grenzwerte, Lösen von Gleichungssystemen, spezielle

Funktionen ...) benötigt, die in verschiedenen Kontexten des symbolischen Rechnens immer wieder auftreten.

Diese in Form von black-box-Wissen vorhandene Kompetenz ist die Kernkompetenz des Systems. Die Implementierung dieser Algorithmen baut wesentlich auf anderen Teilen des Designkonzepts auf, die damit darüber entscheiden, wie effektiv Implementierungen überhaupt sein können. Gewöhnlich sind Teile dieser Sammlung von Funktionen aus Gründen der Performance nicht in der Programmiersprache des jeweiligen Systems, sondern maschinennäher ausgeführt.

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

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

  Spezialbibliotheken sind Sammlungen von Implementierungen algorithmischer Verfahren aus mathematischen Teildisziplinen, die in der Sprache des jeweiligen Systems geschrieben sind und speziellere Kalküle (Tensorrechnung, Gruppentheorie) zur Verfügung stellen. Derartige Spezialbibliotheken werden oftmals von der jeweiligen mathematischen Community als Gemeineigentum entwickelt und gepflegt und von den CAS nur gesammelt und weitergegeben.

  Anwenderbibliotheken sind Sammlungen von Anwendungen mathematischer Methoden in anderen Wissenschaften, die auf den symbolischen Möglichkeiten des jeweiligen CAS aufsetzen. Solche Anwendungsbibliotheken, insbesondere im ingenieur-technischen und business-ökonomischen Bereich, werden oft kommerziell vertrieben.

- Schließlich ist ein Konzept für das *Zusammenwirken des Systemkern mit den anderen Systemkomponenten* zu entwickeln.

2.3 Klassische und symbolische Programmiersysteme

Das klassische Konzept einer imperativen Programmiersprache geht vom Konzept des *Programms* aus als einer „schrittweisen Transformation einer Menge von Eingabedaten in eine Menge von Ausgabedaten nach einem vorgegebenen Algorithmus“ ([4]).

Diese Transformation erfolgt durch *Abarbeiten einzelner Programmschritte,* in denen die Daten entsprechend den angegebenen Instruktionen verändert werden. Den Zustand der Gesamtheit der durch das Programm manipulierten Daten bezeichnet man als den *Programmstatus.*

Die Programmschritte werden in *Anweisungen und Deklarationen* unterteilt, wobei Anweisungen den Programmstatus ändern, Deklarationen dagegen nicht, sondern lediglich Bedeutungen von Bezeichnern festlegen.

Die Definition der Semantik klassischer Programmiersprachen wie etwa C++ in [20] erfolgt in mehreren Schritten:

1. **lexikalische Konventionen**; Definition der einzelnen lexikalischen Einheiten der Sprache (Token, Bezeichner, Schlüsselworte, Literale, Konstanten),

2. **Ausdrücke** als Folge von Operatoren und Operationen, die eine Berechnung im Sinne des funktionalen Programmierens spezifizieren; Definition des lexikalischen Aufbaus, von Syntax, Auswertungsreihenfolge und Bedeutung,

3. **Anweisungen** als Folge von Programmschritten, mit denen der Programmstatus, einem Kontrollfluss folgend, geändert wird; Ausdrucks-Anweisungen (Zuweisungen und Funktionsaufrufe), Verbundanweisungen, Selektions-Anweisungen, Iterations-Anweisungen, Sprung-Anweisungen, Deklarations-Anweisungen,

4. **Deklarationen** und Deklaratoren; sie legen die Interpretation des jeweiligen Bezeichners fest.

Anweisungen (Ausdrucks-Anweisungen) können *Zuweisungen* oder *Prozeduraufrufe* sein. Durch erstere wird direkt der Wert einer Variablen geändert, durch zweitere erfolgt die Änderung des Programmstatus als Seiteneffekt. Die Abfolge der Abarbeitung der einzelnen Programmschritte wird durch *Steuerstrukturen* festgelegt, die klassisch den Anweisungen zugeordnet werden. Dies ist aber nicht zwingend erforderlich. Insbesondere kann zwischen Auswertung der Bestandteile einer Steueranweisung und der Ausführung dieser Anweisung selbst unterschieden werden. Diese Unterscheidung wird von veschiedenen CAS (insbesondere MATHEMATICA) vorgenommen, da eine Steueranweisung mit symbolischen Bestandteilen oft nur unvollständig ausgewertet und deshalb nicht unmittelbar ausgeführt werden kann. Andererseits ist die Anweisung als Rechenvorschrift selbst wieder symbolischer Natur und kann deshalb in symbolischer Form für spätere Auswertungen vorgehalten werden.

Bei **Deklarationen** unterscheidet man gewöhnlich zwischen Deklarationen von *Datentypen*, *Variablen*, *Funktionen* und *Prozeduren*. Jede solche Deklaration verbindet mit einem *Bezeichner* eine gewisse Bedeutung. Derselbe Bezeichner kann in einem Programm in verschiedenen Bedeutungen verwendet werden, was durch die Begriffe *Sichtbarkeit* und *Gültigkeitsbereich* beschrieben wird.

Allerdings sind diese Bedeutungen in einer klassischen Programmiersprache nur zur Übersetzungszeit von Belang, da sie nur unterschiedliche Modi der Zuordnung von Speicherbereich und Adressen zu den jeweiligen Bezeichnern fixieren. Für diese Zwecke wird eine Tabelle, die **Symboltabelle** angelegt, in der die jeweils gültigen Kombinationen von Bezeichner und Bedeutung gegenübergestellt sind.

Bei der Übertragung in ein (typfreies) Maschinenprogramm durch den Compiler steuern die Informationen in der Symboltabelle die Weise, nach welcher dieses Ergebnisprogramm erstellt wird. Im Ergebnisprogramm selbst sind diese Informationen nicht mehr enthalten.

Dies gilt auch für die Auswertung eines Ausdrucks in einem Interpreter: der Ausdruck wird in ein solches Maschinenprogramm übersetzt und dieses dann gestartet, um den Rückgabewert zu berechnen.

In beiden Fällen wird der Ausdruck zunächst geparst, in einem Parsebaum aufgespannt und dann mit Hilfe der in der Symboltabelle vorhandenen Referenzen vollständig ausgewertet. Der Parsebaum existiert damit nur in der Analysephase.

Das ist bei symbolischen Ausdrücken anders: Dort kann nur teilweise ausgewertet werden, denn der resultierende Ausdruck kann symbolische Bestandteile enthalten, denen aktuell kein Wert zugeordnet ist, die aber in Zukunft in der Rolle eines Wertcontainers verwendet werden könnten. Der Parsebaum kann in diesem Fall nicht vollständig abgebaut werden.

> **klassisch:** In der Phase der Syntaxanalyse wird ein Parsebaum des zu analysierenden Ausdrucks auf- und vollständig wieder abgebaut. Im Ergebnis entsteht ein ablauffähiger Programmteil in einer maschinennahen Sprache.
> **symbolisch**: Als Form der Darstellung symbolischer Inhalte bleibt ein Parsebaum auch nach der Syntaxanalyse bestehen, da nicht alle Bezeichner als Referenzen aufgelöst werden können.
> In der **Symboltabelle** eines CAS werden zu den verschiedenen Bezeichnern nicht nur *programmrelevante Eigenschaften* gespeichert, sondern auch darüber hinaus gehende semantische Informationen über die mathematischen Eigenschaften des jeweiligen Bezeichners. Diese Informationen werden dynamisch aktualisiert.
>
> **Die Prozesse in einem CAS zur *Laufzeit* haben damit viele Gemeinsamkeiten mit den Analyseprozessen in einem Compiler zur *Übersetzungszeit*.**

In CAS, die auf einem Hostsystem mit eigener Symboltabelle aufsetzen (SAGE auf Python, MUPAD Live Scripts auf Matlab), müssen alle symbolischen Variablen vor ihrer ersetn Verwendung

als symbolische Variablen deklariert und damit in diese Symboltabelle eingetragen werden.

2.4 Ausdrücke

In klassischen Programmiersprachen spielen Ausdrücke eine zentrale Rolle. Der Wert, welcher einem Bezeichner zugeordnet wird, ergibt sich aus einem *Ausdruck* oder *Funktionsaufruf*. Auch in Funktionsaufrufen selbst spielen (zulässige) Ausdrücke als Aufrufparameter eine wichtige Rolle, denn man kann sie statt Variablen an all den Stellen verwenden, an denen ein *call by value* erfolgt.

Ähnliches gilt für CAS. So beginnt Teil 2 des MATHEMATICA-Handbuch [26] unter der Überschrift „Principles of Mathematica“ mit dem Abschnitt „Everything is an expression“ und den Worten

> *Mathematica* handles many different kinds of things: mathematical formulas, lists and graphics, to name a few. Although they often look very different, *Mathematica* represents all of these things in one uniform way. They are all *expressions*.

Zulässige Ausdrücke werden rekursiv als Zeichenketten definiert, welche nach bestimmten Regeln aus Konstanten, Variablen- und Funktionsbezeichnern sowie verschiedenen *Operationszeichen* zusammengesetzt sind. 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*. Neben Infix-Operatoren spielen auch Postfix- (etwa $x!$), Roundfix- (etwa $|x|$) oder Mixfix-Notationen (etwa $f[2]$) eine Rolle.

Um den Wert von Ausdrücke mit Operatorsymbolen korrekt zu berechnen, müssen gewisse Vorrangregeln eingehalten werden, nach denen zunächst Teilausdrücke (etwa Produkte) zusammengefasst werden. Zur konkreten Analyse solcher Ausdrücke wird vom Compiler ein Baum aufgebaut, dessen Ebenen der grammatischen Hierarchie entsprechen. So besteht (in der Notation von [20]) ein *additive-expression* aus einer Summe von *multiplicative-expression*s, jeder *multiplicative-expression* aus einem Produkt von *pm-expression*s usw. Die Blätter dieses Baumes entsprechen den *atomaren Ausdrücken*, den **Konstanten und Variablenbezeichnern**. Im klassischem Auswerteschema *call by value* ist der Unterschied zwischen Konstanten und Variablen unerheblich, da jeweils ausschließlich der Wert in die Berechnung des entsprechenden Funktionswerts eingeht.

Der Wert des Gesamtausdrucks ergibt sich durch rekursive Auswertung der Teilbäume und Ausführung der entsprechend von innen nach außen geschachtelten Funktionsaufrufen. So berechnet sich etwa der Ausdruck $a + b * c$ über einen Baum der Tiefe 2 als `+(a,*(b,c))`.

> Für einen klassischen Compiler (und Interpreter) können Ausdrücke – nach der Auflösung einiger Diversitäten – als rekursiv geschachtelte Folge von Funktionsaufrufen verstanden werden. Die Argumente eines Funktionsaufrufs können Konstanten, Variablenbezeichner oder (zusammengesetzte) Ausdrücke sein.

Derselbe Mechanismus findet auch in symbolischen Rechnungen Anwendung. Allerdings können auch „Formeln“ als Ergebnisse stehenbleiben, da die Funktionsaufrufe mangels entsprechender Funktionsdefinitionen (Funktionssymbole) oder entsprechender Werte für einzelne Variablenbezeichner (Variablensymbole) nicht immer aufgelöst werden können. Solche Konzepte spielen eine zentrale Rolle bei der Darstellung symbolischer Ausdrücke.

Wie bereits ausgeführt zeichnen sich CAS der zweiten Generation durch das Fehlen eines ausgebauten Typsystems aus, mit dem in klassischen Programmiersprachen Informationen über den mit einem Bezeichner verbundenen „Inhalt“ vorstrukturiert werden können. Typinformationen, die in einzelnen Systemen der zweiten Generation gelegentlich abgefragt werden können, beziehen sich ausschließlich auf syntaktische Informationen über die Struktur der jeweiligen Ausdrücke.

Die Schwierigkeiten, welche sich aus der Einführung eines strengen Typkonzepts im Sinne von Schnittstellendeklarationen und abstrakten Datentypen ergeben, können hier nicht besprochen werden.

Zur internen Darstellung von Ausdrücken in CAS

Heißt es in MATHEMATICA also „everything is an expression“, so bedeutet dies zugleich, dass alles, was in diesem CAS an Ausdrücken verwendet wird, intern nach denselben Prinzipien aufgebaut ist.

In diesem Abschnitt wollen wir studieren, welche Datenstrukturen die einzelnen CAS verwenden, um Ausdrücke (also in unserem Verständnis geschachtelte Funktionsaufrufe) darzustellen. Es sollte sich – wie dies obiger MATHEMATICA-Merksatz nahelegt – um ein uniformes Datenstrukturkonzept handeln, denn anderenfalls hätte man für Polynome, Matrizen, Operatoren, Integrale, Reihen usw. jeweils eigene Datenstrukturen zu erfinden, was das Speichermanagement wesentlich erschweren würde. Die klassische Lösung des Ableitungsbaums mit dem Funktionsnamen in der Wurzel und den Argumenten als Nachfolger hat noch immer den Nachteil geringer Homogenität, da Knoten mit unterschiedlicher Anzahl von Nachfolgern unterschiedliche Speicherplatzanforderungen stellen.

Die Argumente von Funktionen mit unterschiedlicher Arität lassen sich allerdings in Form von Argumentlisten darstellen, wobei der typische Knoten einer solchen Liste aus zwei Referenzen besteht – dem Verweis auf das jeweilige Argument (car) und dem Verweis auf den Rest der Liste (cdr). Die Bezeichnungen `car` und `cdr` sowie dieses Konzept der homogenen Darstellung geschachtelter Listen gehen auf das Sprachkonzept von LISP zurück.

Sehen wir uns an, wie Ausdrücke in MAPLE, MATHEMATICA, MAXIMA, MUPAD und REDUCE intern dargestellt werden. Die folgenden Prozeduren gestatten es jeweils, diese innere Struktur[2] sichtbar zu machen.

MATHEMATICA:

Die Funktion `FullForm` gibt die interne Darstellung eines Ausdruck preis.

MAPLE:

```
level1:=u -> [op(0..nops(u),u)];

structure := proc(u)
  if type(u,atomic) then u else map(structure,level1(u)) fi
end;
```

`level1` extrahiert die oberste Ebene, `structure` stellt die gesamte rekursive Struktur der Ausdrücke[3] dar.

MAXIMA:

```
level1(u):=makelist(part(u,i),i,0,length(u));
structure(u):= if atom(u) then u else map(structure,level1(u));
```

[2] Es bleibt dabei dahingestellt, ob dies wirklich die innere Struktur ist oder nur eine von den Systementwicklern nach außen propagierte Sicht. In den „neueren“ Systemen MAPLE, MATHEMATICA, MUPAD kann diese Frage von außen überhaupt nicht beantwortet werden, da der Systemkern in einer anderen Programmiersprache geschrieben ist. In den „alten“ Systemen ist teilweise ein Blick „unter die Haube“ möglich. In REDUCE etwa mit dem Kommando `lisp prettyprint prop 'u`, wobei u ein Variablen- oder Funktionsbezeichner ist, in MAXIMA mit dem Kommando `:lisp $u`, wobei der Bezeichner u zu $\$u$ umzuschreiben ist. In MAXIMA kann man außerdem mit `to_lisp()` in den Lisp-Mode umschalten und dort mit `(symbol-plist '$u)` weiteren Einblick in die mit verschiedenen Bezeichnern assoziierten Informationen erhalten.

[3] Nicht berücksichtigt ist der Fall, dass ein op-Slot nicht nur einen Ausdruck, sondern eine ganze Ausdruckssequenz (expression sequence) enthält.

REDUCE:

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

Wir betrachten folgende Beispiele:

$$(x+y)^5$$

MATHEMATICA:
```
Power[Plus[x, y], 5]
```

MAPLE und MAXIMA:
```
[^,[+,x,y],5]
```

REDUCE:
```
(plus (expt x 5) (times 5 (expt x 4) y) (times 10
(expt x 3) (expt y 2)) (times 10 (expt x 2) (expt y
3)) (times 5 x (expt y 4)) (expt y 5))
```

REDUCE überführt den Ausdruck sofort in eine expandierte Form. Dies können wir mit `expand` auch bei den anderen beiden Systemen erreichen. Die innere Struktur der expandierten Ausdrücke hat jeweils die folgende Gestalt:

MATHEMATICA:
```
Plus[Power[x, 5], Times[5, Power[x, 4], y], Times[10,
Power[x, 3], Power[y, 2]], Times[10, Power[x, 2],
Power[y, 3]], Times[5, x, Power[y, 4]], Power[y, 5]]
```

MAPLE und MAXIMA:
```
[+, [^, x, 5], [*, 5, [^, x, 4], y], [*, 10, [^, x,
3], [^, y, 2]], [*, 10, [^, x, 2], [^, y, 3]], [*, 5,
x, [^, y, 4]], [^, y, 5]]
```

Ähnliche Gemeinsamkeiten findet man auch bei der Struktur anderer Ausdrücke. Eine Zusammenstellung verschiedener Beispiele finden Sie in der Tabelle. Die interne Darstellung in den Systemen AXIOM, MUPAD[4] und SAGE ist nicht so einfach zu ergründen.

> Wir sehen, dass in allen betrachteten CAS die interne Darstellung der Argumente symbolischer Funktionsausdrücke in Listenform erfolgt, wobei die Argumente selbst wieder Funktionsausdrücke sein können, also der ganze Parsebaum in geschachtelter Listenstruktur abgespeichert wird.
> Der zentrale Datentyp für die interne Darstellung von Ausdrücken in CAS der 2. Generation ist also die geschachtelte Liste.

Bemerkenswert ist, dass sowohl in MAPLE als auch in REDUCE der Funktionsname keine Sonderrolle spielt, sondern als „nulltes“ Listenelement, als *Kopf*, gleichrangig mit den Argumenten in der Liste steht. Das gilt intern auch für MATHEMATICA, wo man auf die einzelnen Argumente eines Ausdruchs s mit `Part[s,i]` und auf das Kopfsymbol mit `Part[s,0]` zugreifen kann. Eine solche Darstellung erlaubt es, als Funktionsnamen nicht nur Bezeichner, sondern auch symbolische Ausdrücke zu verwenden. Ausdrücke statt Funktionsnamen entstehen im symbolischen Rechnen auf natürliche Weise: Betrachten wir etwa $f'(x)$ als Wert der Funktion f' an der Stelle x. Dabei ist f' ein symbolischer Ausdruck, der aus dem Funktionssymbol f durch Anwenden des Postfixoperators $'$ entsteht. Besonders deutlich wird dieser Zusammenhang in MUPAD: `f'(x)` wird sofort als `D(f)(x)` dargestellt, wobei `D(f)` die Ableitung der Funktion f darstellt, die ihrerseits an der Stelle x ausgewertet wird. $D : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$ ist also eine Funktion, die Funktionen in Funktionen abbildet. Solche Objekte treten in der Mathematik (und auch im funktionalen Programmieren) häufig auf. MATHEMATICA geht an der Stelle sogar noch weiter: `f'[x]//FullForm`

[4] In älteren Versionen von MUPAD (mit eigener Notebook-Oberfläche) konnte zur Analyse des Ausdrucks u das Kommando `prog::exprlist(u)` verwendet werden, in neueren Versionen (Live Editor) steht noch das Kommando `children(u)` zur Verfügung, das aber den Kopfterm nicht mit einbezieht.

Matrix in den verschiedenen Systemen definieren:

MATHEMATICA	`M={{1,2},{3,4}}`
MAPLE	`M:=matrix(2,2,[[1,2],[3,4]])`
MAXIMA	`M:matrix([1,2],[3,4])`
REDUCE	`M:=mat((1,2),(3,4))`

$1/2$	MATHEMATICA:	`Rational[1, 2]`
	MAPLE:	`[fraction, 1, 2]`
	MAXIMA:	`[//, 1, 2]`
	REDUCE:	`(quotient 1 2)`
$1/x$	MATHEMATICA:	`Power[x, -1]`
	MAPLE:	`[^, x, -1]`
	MAXIMA:	`[//, 1, x]`
	REDUCE:	`(quotient 1 x)`
$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$	MATHEMATICA:	`List[List[1, 2], List[3, 4]]`
	MAPLE:	`[array, 1 .. 2, 1 .. 2, [(1, 1) = 1, (2, 1) = 3, (2, 2) = 4, (1, 2) = 2]]`
	MAXIMA:	`[MATRIX, ['[', 1, 2], ['[', 3, 4]]`
	REDUCE:	`(mat (1 2) (3 4))`
$\sin(x)^3 + \cos(x)^3$	MATHEMATICA:	`Plus[Power[Cos[x], 3], Power[Sin[x], 3]]`
	MAPLE:	`[+, [^, [sin, x], 3], [^, [cos, x], 3]]`
	MAXIMA:	`[+, [^, [SIN, x], 3], [^, [COS, x], 3]]`
	REDUCE:	`(plus (expt (cos x) 3) (expt (sin x) 3))`
Liste [a,b,c]	MATHEMATICA:	`List[a, b, c]`
	MAPLE:	`[list, a, b, c]`
	MAXIMA:	`['[', a, b, c]`
	REDUCE:	`(list a b c)`

Tabelle 2: Zur Struktur einzelner Ausdrücke in verschiedenen CAS

wird intern als `Derivative[1][f][x]` dargestellt. Hier ist also `Derivative[1]` mit obiger Funktion D identisch, während `Derivative` sogar die Signatur $\mathbb{N} \to ((\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R}))$ hat.

Eine 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 Urmutter aller modernen CAS. CAS verwenden das erste Listenelement darüber hinaus auch zur Kennzeichnung einfacher Datenstrukturen (Listen, Mengen, Matrizen) sowie zur Speicherung von Typangaben atomarer Daten, also für ein **syntaktisches Typsystem.**

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 Grundkursen „Algorithmen und Datenstrukturen" (etwa [13] oder [25]). Die dort eingeführte *destruktiv veränderbare Listenstruktur* mit den zentralen Operationen `insert` und `delete` ist für unsere Zwecke ungeeignet, 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 `prepend`, der eine neue Liste $u = [e, l]$ aus einem Element e und einer Restliste l erstellt, so dass `e=first prepend(e,l)` und `l=rest prepend(e,l)` gilt. Dieses für die Sprache LISP typische Vorgehen[5] erlaubt es, auf aufwändiges Duplizieren von Teilausdrücken zugunsten des Duplizierens von Pointern zu verzichten, indem beim Kopieren einer Liste die Referenzen übernommen, die Pointerpaare 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.

Schließlich sei noch bemerkt, dass als Argumentsequenzen in Funktionsdefinitionen komma-separierte Folgen auftreten. Die meisten CAS kennen neben dem Datentyp *Liste* deshalb auch den Datentyp *Sequenz*, was – grob gesprochen – als „das Innere" einer Liste betrachtet werden kann. Für MAPLE und MUPAD ist eine solche durch die Funktion `op` extrahierbare Sequenz sogar der zentrale Datentyp, während andere CAS wie etwa MATHEMATICA Sequenzen zwar kennen, aber in ihrer reinen Form so gut wie nicht verwenden, sondern aus Konsistenzgründen Sequenzen immer mit einem Kopfterm daherkommen. Die meisten Listenoperationen lassen sich in diesen CAS aber auch auf Ausdrücke anwenden, deren Kopfterm nicht `List` ist.

Mit `Apply` kann in MAXIMA (zweite Zeile) oder MATHEMATICA (dritte Zeile) überdies der Kopfterm einer Sequenz ebenso leicht ausgetauscht werden wie durch `op` in MAPLE (erste Zeile).

```
u:=[1,2,3]: '+'(op(u));
u:[1,2,3]: apply("+",u);
u={1,2,3}; Apply[Plus,u]
```

6

2.5 Das Variablenkonzept des symbolischen Rechnens

Wir hatten bereits gesehen, dass die Analyse symbolischer Ausdrücke *zur Laufzeit*, die ja im Mittelpunkt des Interpreters eines CAS steht, weniger Ähnlichkeiten zum Laufzeitverhalten einer

[5] Dort heißen die drei Funktionen `car`, `cdr` und `cons`.

klassischen Programmiersprache hat als vielmehr zu den Techniken, die Compiler oder Interpreter derselben *zur Übersetzungszeit* verwenden.

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 der Bezeichner x hier im *Symbolmodus* für sich selbst steht oder im *Wertmodus* Container eines anderen Werts ist. Mehr noch kann sich die Bedeutung je nach Kontext unterscheiden. So wird etwa in einem Aufruf

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

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

Sehen wir uns diese Besonderheit des Variablenkonzepts zunächst in einigen REDUCE-Beispielen an. Durch die Zuweisung $x := 2$ wurde der Bezeichner x aus dem *Symbolmodus* x in den *Wertmodus* überführt. Der unter dem Bezeichner x gespeicherte *Wert* geht in die nachfolgende Auswertung von p ein. Die Zuweisung an x beeinflusst also als Seiteneffekt das (zukünftige) Auswerteverhalten von p.

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

$$9\,x^3 - 37\,x^2 + 47\,x - 19$$

```
x:=2;
p;
```

$$-1$$

Versuchen wir nun diese Umwandlung rückgängig zu machen. Der erste Versuch führt nicht zum Erfolg.
x ist noch immer im Wertmodus, nun aber Container eines symbolischen Werts.

```
x:=free;
p;
```

$$9\,\mathrm{free}^3 - 37\,\mathrm{free}^2 + 47\,\mathrm{free} - 19$$

Auch diese „Verzweiflungstat“ hilft nicht weiter, denn bei einer Zuweisung wird die rechte Seite ausgewertet und deren *Wert* der linken Seite zugewiesen.

```
x:=x;
```

$$\mathrm{free}$$

Um x als Symbol zu verwenden, müssen wir diese Auswertung verhindern, was in REDUCE durch das Kommando `let` erreicht werden kann. Leider geht nun gar nichts mehr.

```
let x=x;
p;

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

Der Grund ist leicht gefunden: Da der unter p gespeicherte Ausdruck den Bezeichner x enthält, wird untersucht, ob x im Wertmodus ist (ist es und hat x, sich selbst, als Wert), dieser Wert eingesetzt – was den aktuell für p gefundenen Ausdruck nicht verändert – und dieselbe Frage für denselben Ausdruck noch einmal gestellt. Wir kommen damit in eine Endlosschleife der ständigen Auswertung von x.

Natürlich könnte man für diesen Fall entweder die offensichtlich zu unendlicher Rekursion führende Zuweisung `let x=x` unterbinden oder einfach mit dem Auswerten aufhören, wenn sich nichts mehr ändert.

Um den hier beabsichtigten Effekt zu erreichen, benötigen wir also eine spezielle Anweisung, welche x aus dem Wertmodus in den Symbolmodus zurücksetzt, in dem x kein Wert zugewiesen ist.

```
clear x;
p;
```

$$9\,x^3 - 37\,x^2 + 47\,x - 19$$

Ähnliche Funktionen stehen in allen CAS zur Verfügung. Sie sind in der folgenden Tabelle aufgelistet.

AXIOM	`)clear value x`
MAXIMA	`kill(x)`
MAPLE	`x:='x'`
MATHEMATICA	`Clear[x]`
MUPAD	`delete x`
REDUCE	`clear x`
SAGE	`x='x'`

Tabelle 3: Bezeichner in den Symbolmodus zurücksetzen

MAPLE suggeriert mit seiner Syntax, dass hier eine Selbstwert-Zuweisung erfolgt, aber intern ist es anders implementiert (und funktioniert auch anders als die Zuweisung `x:='y'`).

MAXIMA dagegen folgt einem anderen Auswerteverhalten – es wird p nur eine Ebene tief ausgewertet und der Wert $x = 2$ nicht rekursiv eingesetzt. Dafür muss zusätzlich ein Schalter `infeval` gesetzt oder die zusätzliche Evaluation durch `eval` getriggert werden. Das Auswerteverhalten von MAXIMA ist noch deutlich komplexer, was hier nicht im Detail diskutiert werden kann.

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

$$9\,x^3 - 37\,x^2 + 47\,x - 19$$

```
x:2; p;
```

$$9\,x^3 - 37\,x^2 + 47\,x - 19$$

```
p,infeval; /* oder p,eval; */
```

$$-1$$

Damit lassen sich unendliche Evaluationsschleifen durch rekursive Wertzuweisungen bereits im Ansatz vermeiden. Meist sind solche rekursiven Evaluationen auch nicht intendiert.

Bezeichner werden, wie bereits beschrieben, in einer *Symboltabelle* erfasst, um sie an allen Stellen, an denen sie im Quellcode auftreten, in einheitlicher Weise zu verarbeiten. Für jeden Bezeichner existiert **genau** ein Eintrag in der Tabelle, unter dem auch weitere Informationen vermerkt sind.

Da in einem CAS jeder in irgendeinem Ausdruck auftretender Bezeichner – auch wenn er zunächst im Symbolmodus ist – später in den Wertmodus übergehen kann, muss dieser *Mechanismus der eindeutigen Referenz* in einem solchen Kontext auf **alle** Bezeichner von ihrem allerersten Auftreten in einem Ausdruck an angewendet werden. Dementsprechend wird bei der Analyse der einzelnen Eingaben jeder String, der einen Bezeichner darstellen könnte, daraufhin geprüft, ob er bereits in die Symboltabelle eingetragen ist. In diesem Fall wird eine Referenz an die entsprechende Stelle der Symboltabelle eingetragen. Andernfalls ist die Symboltabelle um einen neuen Eintrag zu erweitern.

In einigen CAS kann man sich diese Symboltabelle ganz oder teilweise anzeigen lassen. In MATHEMATICA sind die Bezeichner nach Kontexten geordnet, wobei die Kontexte `Global‘` (alle nutzerdefinierten Bezeichner) und `System‘` (alle vom System eingeführten Bezeichner) die wichtigsten sind. Führt man in einer neu begonnenen Sitzung die Zuweisung $u = (x + y)^2$ aus, so sind danach im Kontext `Global‘` drei Symbole definiert, wovon u im Wertmodus, x und y dagegen (noch) im Symbolmodus sind.

```
u=(x+y)^2
Names["Global‘*"]

     u x y

?u

     Global‘u
     u = (x + y)^2

?x

     Global‘x
```

MAPLE verfügt sogar über zwei solche Funktionen. Die Funktion `unames` listet alle Bezeichner im Symbolmodus (unassigned names) auf, `anames` alle Bezeichner im Wertmodus.

Zusammenfassung

In CAS treten Bezeichner in zwei verschiedenen **Modi**, im Symbol- oder im Wertmodus auf. Ein Bezeichner ist so lange im Symbolmodus, bis ihm ein Wert zugewiesen wird. Durch eine Wertzuweisung geht der Bezeichner in den Wertmodus über.
Für einen Bezeichner im Wertmodus ist zwischen dem Bezeichner als Wertcontainer und dem Bezeichner als Symbol zu unterscheiden.
Durch eine spezielle Deklaration kann ein Bezeichner in den Symbolmodus zurückversetzt werden. Dabei gehen alle Wertzuweisungen an diesen Bezeichner verloren.
Bezeichner werden in einer Symboltabelle zusammengefasst, um ihre eindeutige Referenzierbarkeit zu sichern.

Dieses einheitliche Management der Bezeichner 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. Dies ist auch deshalb erforderlich, weil in einen Ausdruck wie `D(f)`, die Ableitung der einstelligen Funktion f, Funktionssymbole auf dieselbe Weise eingehen wie Variablensymbole in den Ausdruck `sin(x)`.

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 Bezeichner zu verbinden.

Eine solche einheitliche Behandlung interner und externer Bezeichner 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.

2.6 Auswerten von Ausdrücken

Die Tatsache, dass der Wert von Bezeichnern symbolischer Natur sein kann, in den Bezeichner eingehen, die ihrerseits einen Wert besitzen können, führt zu einer weiteren Besonderheit von CAS.

Betrachten wir diese Zuweisungen in MATHEMATICA und sehen uns an, was als Wert des Symbols a berechnet wird.

```
a=b+1; b=c+3; c=3; a
```

$$7$$

Es ist also die gesamte Evaluationskette durchlaufen worden: In den Wert $b+1$ von a geht das Symbol b ein, das seinerseits als Wert den Ausdruck $c+3$ hat, in den das Symbol c eingeht, welches den Wert 3 besitzt.

Denkbar wäre auch eine eingeschränkte Evaluationstiefe, etwa nur Tiefe 1 wie in MAXIMA. In MATHEMATICA kann man das Evaluationsgeschehen mit dem Kommando `Trace` verfolgen.

```
Trace[a]
```

$$\{a, 1+b, \{b, 3+c, \{c, 3\}, 3+3, 6\}, 1+6, 7\}$$

Ändern wir den Wert eines Symbols in dieser Evaluationskette, so kann sich auch der Wert von a bei erneuter Evaluation ändern.

```
b=2*d; a
```

$$2\,d$$

```
Trace[a]
```

$$\{a, 1+b, \{b, 2\,d\}, 1+2\,d\}$$

> Wertzuweisungen an eine Variable können als Seiteneffekt Einfluss auf das Auswerteverhalten anderer Variablen haben.

Bei dieser Art der rekursiven Auswertung kann es eintreten, dass ein zu evaluierendes Symbol nach endlich vielen Evaluationsschritten selbst wieder in dem entstehenden Ausdruck auftritt und damit der Evaluationsprozess nicht terminiert. Die CAS, welche diesen Ansatz verwenden, haben deshalb verschiedene Zähler, mit denen verfolgt wird, wieviele rekursive Auswerterunden schon ausgeführt wurden.

MATHEMATICA verwendet dazu die Systemvariablen `$RecursionLimit` und `$IterationLimit`. Die rekursive Auswertung von Symbolen wird nach Erreichen der vorgegebenen Tiefe abgebrochen und der nicht weiter ausgewertete symbolische Ausdruck in eine `Hold`-Anweisung eingeschlossen, um ihn auch zukünftig vor (automatischer) Auswertung zu schützen.

Innerhalb von Funktionsdefinitionen wird meist nur eine Ebene tief ausgewertet und damit rekursive Auswertung standardmäßig unterbunden. Das ist eine plausible Setzung, da lokale Variablen in Funktionen generell nur als Wertcontainer verwendet werden (sollten) und nicht als Symbole. Diese Standardsetzung vermeidet zugleich nicht terminierende Auswerteprozesse in Funktionskörpern.

Dasselbe Prinzip – Auswertung nur eine Ebene tief – wird von MAXIMA, AXIOM und SAGE als globales Auswertungsschema verwendet, wobei es in MAXIMA zusätzlich eine Funktion `eval` gibt, mit welcher eine mehrfache Auswertung ausgeführt werden kann. Das Auswertungsverhalten dieser drei CAS unterscheidet sich damit von dem der anderen CAS[6].

Zuweisung	MAXIMA	MATHEMATICA
`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 4: Unterschiedliches Auswertungsverhalten von MAXIMA und MATHEMATICA

[6]Das Auswertungsverhalten der anderen CAS kann in MAXIMA für den Ausdruck `expr` durch den Zusatz `expr,infeval;` erreicht werden.

Werden auf der rechten Seite einer Zuweisung Bezeichner im Wertmodus verwendet, so ist zu unterscheiden, ob das Symbol oder dessen Wert gemeint ist.

In diesem MAPLE-Beispiel wurde a der Wert 3 des Bezeichners u zugewiesen, b dagegen das Symbol u, dessen aktueller Wert 3 ist. An dieser Stelle ist bei einer erneuten Auswertung der Unterschied noch nicht zu erkennen.

```
u:=3: a:=u: b:='u': [a, b]
```

$$[3, 3]$$

Nach dieser Änderung jedoch erkennen wir, dass sich Änderungen des Werts von u auf b auswirken, auf a dagegen nicht.

```
u:=5: [a, b];
```

$$[3, 5]$$

Geht ein Bezeichner im Wertmodus in einen Ausdruck ein, so ist also zu unterscheiden, ob der Wert oder das Symbol gemeint ist. Im ersten Fall wird der Bezeichner *ausgewertet*, im zweiten Fall wird der Bezeichner *nicht ausgewertet*.

MATHEMATICA kennt hierfür sogar zwei Zuweisungsoperatoren, `=` für eine Zuweisung *mit* Auswertung der rechten Seite und `:=` für eine Zuweisung *ohne* Auswertung der rechten Seite. Das gerade betrachtete Beispiel lässt sich damit in MATHEMATICA auf diese Weise anschreiben.

```
u=3; a=u; b:=u; {a, b}
```

$$\{3, 3\}$$

```
u=5; {a, b}
```

$$\{3, 5\}$$

Oft spricht man in diesem Zusammenhang von *früher Auswertung* und *später Auswertung*. Diese Terminologie ist allerdings irreführend, denn beide Ergebnisse werden natürlich bei einem späteren Aufruf, wenn sie als Teil in einen auszuwertenden Ausdruck eingehen, auch selbst ausgewertet. Korrekt müsste man also von *früher Auswertung* und *früher Nichtauswertung* sprechen.

Generell gibt es zwei verschiedene Mechanismen, einen Bezeichner im Wertmodus vor der Auswertung zu bewahren. Dies geschieht entweder wie in MAPLE, MAXIMA oder MUPAD (und LISP) durch eine spezielle *Hold*-Funktion oder wie in AXIOM oder REDUCE durch zwei verschiedene Zuweisungsoperatoren, wovon einer die in die rechte Seite eingehenden Bezeichner auswertet, der andere nicht. MATHEMATICA verfügt sogar über beide Mechanismen. Diese Besonderheiten sind in einer klassischen Programmiersprache, in der sich Namensraum und Wertebereich nicht überlappen, unbekannt.

> Die mit einem solchen Verhalten verbundene Konfusion lässt sich vermeiden, wenn Bezeichner im Wertmodus konsequent *nur* als Wertcontainer verwendet werden. Dazu muss von Anfang an geplant werden, welche Bezeichner im Symbolmodus und welche als Wertcontainer verwendet werden sollen. Bezeichnern, die nicht explizit als Wertcontainer vorgesehen sind, darf dann im gesamten Gültigkeitsbereich (global) kein Wert zugewiesen werden.

Muss einem solchen Bezeichner im Symbolmodus in einem *lokalen Kontext* ein Wert zugewiesen werden, so kann dies unter Verwendung des **Substitutionsoperators** erfolgen. Diese Wertzuweisung ist nur in dem entsprechenden Ausdruck wirksam, ein Moduswechsel des Bezeichners erfolgt nicht. Es ist zu beachten, dass in einigen CAS dabei keine vollständige Evaluation oder gar Simplifikation des Ergebnisses ausgeführt wird.

System	Substitutions-operator	Zuweisung mit Auswertung	Zuweisung ohne Auswertung	Hold-Operator
AXIOM	subst(f(x),x=a)	x:=a	x==a	–
MAXIMA	ev(f(x),x=a)	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	–
SAGE	f(x).subs(x=a)	x=a	–	'x'

Tabelle 5: Substitutions- und Zuweisungsoperatoren der verschiedenen CAS

2.7 Der Funktionsbegriff im symbolischen Rechnen

Funktionen bezeichnen im mathematischen Sprachgebrauch *Abbildungen* $f : X \to Y$ von einem Definitionsbereich X in einen Wertevorrat Y. In diesem Verständnis steht der ganzheitliche Aspekt stärker im Mittelpunkt, der es erlaubt, über Klassen von Funktionen und deren Eigenschaften zu sprechen sowie abgeleitete Objekte wie Stammfunktionen und Ableitungen zu betrachten.

In klassischen Programmiersprachen steht dagegen der konstruktive Aspekt der Funktionsbegriffs im Vordergrund, d.h. der *Algorithmus*, nach welchem die Abbildungsvorschrift f jeweils realisiert werden kann. Eine Funktion ist in diesem Sinne eine (beschreibungs-)endliche Berechnungsvorschrift, die man auf Elemente $x \in X$ anwenden kann, um entsprechende Elemente $f(x) \in Y$ zu produzieren. Wir unterscheiden dabei Funktionsdefinitionen und Funktionsaufrufe.

Betrachten wir den Unterschied am Beispiel der Wurzelfunktion `sqrt`. Die Mathematik gibt sich durchaus mit dem Ausdruck `sqrt(2)` zufrieden und sieht darin sogar eine exaktere Antwort als in einem dezimalen Näherungswert. Sie hat dafür extra die symbolische Notation $\sqrt{2}$ erfunden. In einer klassischen Programmiersprache wird dagegen beim Aufruf `sqrt(2)` ein Näherungswert für $\sqrt{2}$ berechnet, etwa mit dem Newtonverfahren. Beim Aufruf `sqrt(x)` würde sogar mit einer Fehlermeldung abgebrochen, da dieses Verfahren für symbolische Eingaben nicht funktioniert. Mathematiker würden dagegen, wenigstens für $x \geq 0$, als Antwort `sqrt(x)` $= \sqrt{x}$ gelten lassen als „diejenige positive reelle Zahl, deren Quadrat gleich x ist“.

So ist es auch in CAS. Symbolische Ausdrücke werden intern, wie wir gesehen haben, sofort in Funktionsausdrücke wie `sqrt(2)` umgesetzt, in denen Funktionssymbole wie `sqrt`, d. h. „Funktionen ohne Funktionsdefinition“, ein wichtiger Bestandteil sind. Die Beziehung zwischen Funktionssymbolen mit Funktionsdefinition und Funktionssymbolen ohne Funktionsdefinition ist vollkommen analog zum Wechselverhältnis zwischen Wert- und Symbolmodus von Variablen.

Allerdings hängt der Modus eines Funktionsbezeichners zusätzlich von den jeweiligen Aufrufparametern ab. Es wird grundsätzlich zunächst versucht, den Bezeichner als Funktionsaufruf zu interpretieren. Die Auswertung eines solchen Funktionsaufrufs folgt dem klassischen Schema, wobei als Wert eines Aufrufparameters ein symbolischer Ausdruck im weiter oben definierten Sinne auftritt. Existiert keine Funktionsdefinition, so wird allerdings nicht mit einer Fehlermeldung abgebrochen, sondern aus den Werten der formalen Parameter und dem Funktionssymbol als „Kopf“ ein neuer Funktionsausdruck gebildet.

Normalerweise ist damit eine Funktion nur partiell (im informatischen Sinne) definiert ist, d. h. für spezielle Argumente findet ein Funktionsaufruf statt, für andere dagegen wird ein Funktionsausdruck gebildet wird.

So wird etwa in diesen MAXIMA-Konstrukten das Funktionssymbol `sin` zur Konstruktion von Funktionsausdrücke verwendet, die für Sinus-Funktionswerte stehen, welche nicht weiter ausgewertet werden können.

```
[sin(x), sin(2)];
```

$$[\sin(x), \sin(2)]$$

Im Gegensatz dazu ist dies ein klassischer Funktionsaufruf, der eine Funktionsdefinition von `sin` zur näherungsweisen Berechnung für einen reellwertigen Parameter verwendet.

```
sin(2.55);
```

$$0.5576837174$$

Es kann auch sein, dass eine erneute Auswertung eines Funktionsausdrucks zu einem späteren Zeitpunkt einen Funktionsaufruf absetzt, wenn dem Funktionssymbol inzwischen eine Funktionsdefinition (für diese Argumente) zugeordnet worden ist.

Funktionstransformationen und Transformationsfunktionen

Eine besondere Art von Funktionen sind die MAPLE-Funktionen `expand, collect, combine, normal`, die symbolische Ausdrücke *transformieren*, d. h. Funktionsaufrufe darstellen, deren Wirkung auf einen Umbau der als Parameter übergebenen Funktionsausdrücke – eine Funktionstransformation – ausgerichtet ist.

> Solche *Funktionstransformationen* ersetzen gewisse Kombinationen von Funktionssymbolen und evtl. speziellen Funktionsargumenten durch andere, semantisch gleichwertige Kombinationen.

Transformationen sind eines der zentralen Designelemente von CAS, da auf diesem Wege syntaktisch verschiedene, aber semantische gleichwertige Ausdrücke produziert werden können. Sie werden bis zu einem gewissen Grad automatisch ausgeführt.

So wird etwa bei der Berechnung von `sin(Pi/4)` die Transformation auf Grund des Zusammentreffens der Symbole bzw. symbolischen Ausdrücke `sin` und `Pi/4` ausgelöst, welches das CAS von sich aus erkannt und automatisch ausgeführt hat.

```
sin(Pi/4);
```

$$\frac{1}{2}\sqrt{2}$$

Auch automatisch ausgeführten Vereinfachungen wie etwa `sqrt(2)^2` zu 2 liegen Transformationen zu Grunde. Da Transformationen in einem breiten und in seiner Gesamtheit widersprüchlichen Spektrum möglich sind, können sie in den meisten Fällen aber nicht automatisch vorgenommen werden. Für einzelne Transformationsaufgaben gibt es deshalb spezielle *Transformationsfunktionen*, die aus dem Gesamtspektrum eine (konsistente) Teilmenge von Transformationen auf einen als Parameter übergebenen Ausdruck *lokal* anwenden.

Betrachten wir die Wirkung eine solchen Transformationsfunktion, hier der MAPLE-Funktion `expand`, näher.

Dieser Aufruf von `expand` verwandelt ein Produkt von zwei Summen in eine Summe nach dem Distributivgesetz.

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

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

Dieser Aufruf von `expand` verwandelt eine Winkelfunktion mit zusammengesetztem Argument in einen zusammengesetzten Ausdruck mit einfachen Winkelfunktionen. Es wurde eines der Additionstheoreme für Winkelfunktionen angewendet.

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

$$\sin(x)\cos(y) + \cos(x)\sin(y)$$

Dieser Aufruf von `expand` schließlich ersetzt eine Summe im Exponenten durch ein Produkt entsprechend den Potenzgesetzen.

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

$$e^a e^{\sin(b)}$$

Charakteristisch für solche Transformationen ist die Möglichkeit, das Aufeinandertreffen von vorgegebenen Funktionssymbolen in einem Ausdruck festzustellen. Dabei wird ein Grundsatz der klassischen Funktionsauswertung verletzt: Es wird nicht nur der *Wert* der Aufrufargumente benötigt, sondern auch Information über deren *Struktur*.

So muss die Transformationsfunktion beim Aufruf `expand(sum`$_1$`*sum`$_2$`)` etwa erkennen, dass ihr Argument ein Produkt zweier Summen ist, die Listen l_1 und l_2 der Summanden beider Faktoren extrahieren und einen Funktionsaufruf `expandproduct(`l_1`,`l_2`)` absetzen, der aus l_1 und l_2 alle paarweisen Produkte bildet und diese danach aufsummiert. Details sind hier im Systemkern verborgen.

Ähnlich müsste `expand(sin(sum))` das Funktionssymbol `sin` des Aufrufarguments erkennen und danach eine Funktion `expandsin`[7] aufrufen.

> Charakteristisch für Transformationsfunktionen ist also der Umstand, dass nicht nur der (semantische) Wert, sondern auch die (syntaktische) Struktur der Aufrufparameter an der Bestimmung des Rückgabewerts beteiligt ist, womit komplexere Teilstrukturen der Aufrufargumente zu analysieren sind.

Transformationen sind manchmal nur über Umwege zu realisieren, da beim Aufruf einer Funktion deren Argumente *ausgewertet* werden, was deren Struktur so verändern kann, dass die syntaktischen Bestandteile, deren Zusammentreffen ausgewertet werden soll, gar nicht mehr vorhanden sind.

Möchte man etwa das Polynom

$$f = x^3 + x^2 - x + 1 \in \mathbb{Z}[x]$$

nicht über den ganzen Zahlen, sondern modulo 2, also im Ring $\mathbb{Z}_2[x]$, faktorisieren, so führen in Maple weder die erste noch die zweite Eingabe zum richtigen Ergebnis.

```
factor(f) mod 2;
```

$$x^3 + x^2 + x + 1$$

```
factor(f mod 2);
```

$$(x+1)\left(x^2+1\right)$$

Das zweite Ergebnis kommt der Wahrheit zwar schon nahe (Ausmultiplizieren ergibt $x^3 + x^2 + x + 1 \equiv f \pmod 2$), aber ist wegen $(x^2+1) \equiv (x+1)^2 \pmod 2$ noch immer falsch.

In beiden Fällen wird bei der Auswertung der Funktionsargumente die für eine Transformation notwendige Kombination der Symbole `factor` und `mod` zerstört, denn `factor` ist ein Funktions*aufruf*, der als Ergebnis ein Produkt, die Faktorzerlegung des Arguments über den ganzen Zahlen, zurückliefert. Im ersten Fall (der intern als `mod(factor(f),2)` umgesetzt ist) wird vor dem Aufruf von `mod` das Polynom (in $\mathbb{Z}[x]$) faktorisiert. Das Ergebnis enthält die Information `factor` nicht mehr, so dass `mod` als Reduktionsfunktion $\mathbb{Z}[x] \to \mathbb{Z}_2[x]$ operiert und die Koeffizienten dieser ganzzahligen Faktoren reduziert. Im zweiten Fall dagegen wird das Polynom f erst modulo 2 reduziert.

[7] In Maple heißt sie `‘expand/sin‘` und kann mit `interface(verboseproc = 2); print(‘expand/sin‘)` studiert werden.

Das Ergebnis enthält die Information `mod` nicht mehr und wird als Polynom aus $\mathbb{Z}[x]$ interpretiert und entsprechend faktorisiert.

Das CAS hat also in beiden Fällen nicht die Faktorisierung über einem modularen Bereich, sondern die über den ganzen Zahlen aufgerufen. Wenn die Faktorisierung als Funktions*aufruf* realisiert würde, so müsste zur Unterscheidung zwischen den Algorithmen zur Faktorisierung von Polynomen über $\mathbb{Z}$ und über Restklassenkörpern auf verschiedene Funktionsnamen, etwa `factor` und `factormod`, zurückgegriffen werden.

Um dies wenigstens in dem Teil, der dem Nutzer sichtbar ist, zu vermeiden, führt MAPLE ein Funktionssymbol `Factor` ein, das sein Argument unverändert zurückgibt.

```
Factor(f) mod 2;
```

$$(x+1)^3$$

Der Aufruf wird als `mod(Factor(f),2)` umgesetzt, beim Auswerten bleibt `Factor(f)` als Funktionsausdruck unverändert und am Zusammenstoßen von `mod` und `Factor` wird erkannt, dass modulares Faktorisieren aufzurufen ist. Dazu wird eine Funktions*transformation* ausgelöst, die aus den Bestandteilen f und 2 den Funktions*aufruf* `` `mod/Factor`(f,2) `` generiert[8].

Der Name `` `mod/Factor` `` dieser MAPLE-Funktion ergibt sich direkt durch Stringkonkatenation, wobei die Backquotes aus dieser Zeichenkette, die mit / ein für normale Bezeichner nicht zulässiges Zeichen enthält, einen Bezeichner machen. Wir sehen, dass es die symbolischen Möglichkeiten eines CAS erlauben, im Prozess des Abarbeitens einer Funktion neue Bezeichner zu generieren. Diese Bezeichner können sowohl für Variablen stehen als auch Funktionsbezeichner sein. In beiden Fällen kann es sich sowohl um neue als auch dem System bereits bekannte Bezeichner handeln. Dieser Mechanismus kann mit einiger Perfektion zur Simulation von Polymorphismus eingesetzt werden.

Funktionssymbole wie `Factor` werden in der MAPLE-Dokumentation als *inerte Funktionen* bezeichnet. In der hier verwendeten Terminologie handelt es sich um Funktionssymbole ohne Funktionsdefinition, so dass alle Funktionsaufrufe zu Funktionsausdrücken mit `Factor` als Kopf auswerten. Solche Hilfskonstruktionen sind für diesen Zweck allerdings nicht zwingend erforderlich.

So lässt sich etwa in MATHEMATICA der modulare Faktorisierungsalgorithmus über eine Option `Modulus` aufrufen.

```
Factor[f,Modulus -> 2]
```

$$(1+x)^3$$

Auch hier wird am Vorhandensein und der Struktur des zweiten Arguments erkannt, dass die modulare Faktorisierungsroutine zu verwenden ist und diese im Zuge der Transformation als Funktionsaufruf aktiviert.

Diese spezielle Kombination von Bezeichner `Modulus` und Wert 2 in einem `Rule`-Konstrukt wird in MATHEMATICA generell für die Angabe von Optionen verwendet. Da im Aufrufkonstrukt Optionsbezeichner *und* Optionswert erhalten bleiben, kann damit eine syntaktische Analyse wie oben beschrieben stattfinden. Voraussetzung ist allerdings, dass Optionsbezeichner ausschließlich im Symbolmodus verwendet werden. Für systeminterne Optionsbezeichner wird dies durch den `Protect`-Mechanismus sichergestellt, der die Zuweisung von Werten verhindert. Dieser Zugang ließe sich leicht auch in MAPLE realisieren.

Ähnlich geht MAXIMA vor. Hier können Optionen als lokale Werte von Kontextvariablen komma-separiert angegeben werden.

```
factor(f),modulus=2;
```

$$(1+x)^3$$

[8] Beachten Sie, dass durch die Backquotes der String `mod/Factor` als Bezeichner interpretiert wird, der Parser ohne diese Backquotes den String aber in drei Token `mod`, `/` und `Factor` zerlegen würde. Security by obscurity.

In SAGE kann derselbe Effekt über den objektorientierten Ansatz erreicht werden. Zunächst wird der Polynomring R und die Variable $x \in R$ konstruiert. Bei der Konstruktion von f werden die Polynomoperationen im Ring R verwendet. `type(x)` und `type(f)` weisen beide Variablen als zu einem speziellen Polynomring-Typ gehörig aus. Auf der Basis dieser Information kann auch die korrekte Faktorisierungsroutine ausgewählt werden.

```
R=PolynomialRing(GF(2),'x')
x=R.gen()
f=x^3+x^2-x+1
factor(f)
```

$$(x+1)^3$$

Noch einfacher kann diese Rechnung in AXIOM angeschrieben werden. Das Polynom f hat den Typ `Polynomial(Integer)`, der durch den Coerce-Operator `::` in ein Polynom über $\mathbb{Z}_2$ verwandelt wird. Aus dem Typ wird nun die korrekte Faktorisierungsroutine inferiert.

```
f:=x^3+x^2-x+1
factor(f::Polynomial PrimeField 2)
```

$$(x+1)^3$$

Auch die Ergebnisse von auf den ersten Blick stärker „algorithmischen“ Funktionen wie etwa `diff`, mit der man in MAPLE Ableitungen berechnen kann, entstehen oft durch Transformationen.

Beim Zusammentreffen von `diff` und `sin` wurde diese Kombination durch `cos` ersetzt.

```
diff(sin(x),x);
```

$$\cos(x)$$

Hier ist gar nichts geschehen, denn das Ergebnis ist nur die zweidimensionale Ausgabeform des Funktionsausdrucks `diff(f(x),x)`, der nicht vereinfacht werden konnte, weil über f hier nichts weiter bekannt ist.

```
diff(f(x),x);
```

$$\frac{d}{dx} f(x)$$

Einzig die Kettenregel gilt für beliebige Funktionsausdrücke, so dass diese Transformation automatisch ausgeführt wird. Auch wenn für die Bezeichner f und g nichts bekannt ist, so ist aus der syntaktischen Struktur zu erkennen, dass es sich um Funktionssymbole handelt.

```
diff(f(g(x)),x);
```

$$\mathrm{D}(f)(g(x)) \frac{d}{dx} g(x)$$

Hinter der zweidimensionalen Ausgabe verbirgt sich der Ausdruck `D(f)(g(x))*diff(g(x),x)`. Dabei tritt mit dem Symbol D sogar eine Funktion auf, die ein Funktions*symbol* als Argument hat, weil die Ableitung der Funktion f an der Stelle $y = g(x)$ einzusetzen ist.

Solche Funktionen von Funktionen entstehen in verschiedenen mathematischen Zusammenhängen auf natürliche Weise, da auch Funktionen Gegenstand mathematischer Kalküle (etwa der Analysis) sind. Sie sind auch im informatischen Kontext etwa im funktionalen Programmieren (LISP) gut bekannt. Die Ableitung von $f(x)$, die in MAPLE nicht nur als `diff(f(x),x)`, sondern auch als `D(f)(x)` dargestellt wird, kann in MATHEMATICA näher an der üblichen mathematischen Notation als `f'[x]` eingegeben werden. Es ist in diesem Zusammenhang wichtig und nicht nur aus mathematischer Sicht korrekt, zwischen der Ableitung der Funktion f und des Ausdrucks $f(x)$ zu unterscheiden; gerade dieser Umstand wird durch die beschriebene Notation berücksichtigt.

Funktionen von Funktionen können auch ein Eigenleben führen.
Um die Antwort im dritten Beispiel zu formulieren, hat MAPLE eine weitere Art von Funktionen eingeführt, von denen nur die Zuordnungsvorschrift bekannt ist, die aber keinen Namen besitzen. Solche Funktion werden auch als namenlose Funktionen (pure function) bezeichnet. Sie ist das Gegenteil eines Funktionssymbols, von dem umgekehrt der Name, aber keine Anwendungsvorschrift bekannt war.

```
D(sin);
```

$$\cos$$

```
D(tan);
```

$$1 + \tan^2$$

```
D(ln);
```

$$\left(a \mapsto a^{-1}\right)$$

Namenlose Funktionen entstehen auf natürliche Weise in Antworten des CAS auf Problemstellungen, in denen Funktionen zu konstruieren sind, wie etwa beim Integrieren und allgemeiner beim Lösen von Differenzialgleichungen. In der Informatik sind sie aus dem *Lambda-Kalkül* gut bekannt.

Die Notation $f = 1 + \tan^2$ ist erklärungsbedürftig. Was bedeutet $f(x) = (1 + \tan^2)(x)$? Offensichtlich wird dabei auf die Eigenschaft von Funktionen in der Menge $F = \mathrm{Fun}(\mathbb{R} \to \mathbb{R})$ Bezug genommen, dass diese addiert und multipliziert werden können, indem $(g_1 + g_2)(x) = g_1(x) + g_2(x)$ und ähnlich Multiplikation und Division punktweise definiert werden. $1(x) = 1$ ist dann die konstante Funktion. Diese Notation ist im Vergleich zum Lambda-Kalkül allerdings weniger leistungsfähig.

MATHEMATICA verwendet deshalb in seinen Ausgaben konsequent die Notation $\ldots[\#]\,\&$ für namenlose Funktionen, und zwar einheitlich auch in den Fällen, wo der Name der Funktion eigentlich bekannt ist. # steht dabei für den bzw. die formalen Parameter und & schließt die Funktionsdefinition ab. Die interne Darstellung kann wieder mit `FullForm` studiert werden.

```
{Sin', Log'}
```

$$\left\{\mathrm{Cos}[\#1]\,\&,\ \frac{1}{\#1}\,\&\right\}$$

Es ist sogar denkbar, dass nicht nur der Name, sondern auch die genaue Zuordnungsvorschrift nicht bekannt ist und nur eine semantische Spezifikation der Funktion als „Black Box“ existiert wie im Ergebnis der folgenden Interpolationsaufgabe in MATHEMATICA:

```
points = {{0,0},{1,1},{2,3},{3,4},{4,3},{5,0}};
ifun = Interpolation[points]
```

$$\mathrm{InterpolatingFunction}[\{\{0, 5\}\}, <>]$$

`ifun` ist eine auf dem Intervall $[0, 5]$ definierte Interpolationsfunktion durch die angegebenen Punkte `points`, welche die in der Dokumentation angegebenen Eigenschaften hat, von der aber diesmal nicht einmal die genaue Funktiondefinition durch entsprechende Optionen angezeigt werden kann.

Solche Antworten entstehen zum Beispiel, wenn das Ergebnis Funktionen enthält, die nur in vorcompilierter Form vorliegen. Auch derartige Funktionen können einem Bezeichner zugewiesen und dann zur grafischen Visualisierung, hier zum Beispiel mit `Plot[ifun[x],{x,0,5}]`, aufgerufen werden. Auch eine Taylorentwicklung `Series[ifun[x],{x,0,3}]` kann berechnet werden.

2.8 Listen und Steuerstrukturen im symbolischen Rechnen

Nachdem wir die Details und Besonderheiten beim Auswerten von Ausdrücken besprochen haben, können wir uns nun der Ablaufsteuerung in komplexeren Programmkonstrukten zuwenden, die wir weiter vorn im Konzept der *Anweisung* bzw. Steuerstrukturen zusammengefasst hatten.

In diesem Bereich weist ein CAS die größte Ähnlichkeit mit klassischen Programmiersprachen 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 imperativer Programmiersprachen orientieren. Auch Unterprogrammtechniken stehen zur Verfügung, wie wir im Abschnitt „Funktionen“ bereits gesehen hatten. Ich verzichte deshalb auf eine detaillierte Darstellung dieser Konzepte und Sprachmittel.

Im letzten Abschnitt komme ich auf eine weitere Besonderheit des symbolischen Rechnens zu sprechen, welche sich aus dem Umstand ergibt, dass in den Testbedingungen, welche den Kontrollfluss steuern, boolesche Ausdrücke vorkommen können, die nicht vollständig ausgewertet werden können, etwa weil sie Bezeichner im Symbolmodus enthalten.

Operationen auf Listen

Zunächst soll aber der qualifizierte Umgang mit Listen genauer besprochen werden. Listen nehmen eine zentrale Stellung im Datentypdesign ein, und jedes CAS stellt deshalb eine Unmenge verschiedener Funktionen zur Listenmanipulation zur Verfügung. In dieser Fülle kann man schnell den Überblick verlieren. Es zeigt sich aber, dass bereits ein kleines Repertoire an Funktionalität ausreicht, um alle erforderlichen Aufgaben zur Listenmanipulation zuverlässig ausführen zu können. Dieses Repertoire wird im Weiteren besprochen.

Zugriff auf Listenelemente

Zum Umgang mit Listen wird zunächst einmal ein **Zugriffsoperator** auf einzelne Listenelemente, evtl. auch über mehrere Ebenen hinweg, benötigt. Die meisten Systeme stellen auch eine iterierte Version zur Verfügung, mit der man tiefer gelegene Teilausdrücke in einer geschachtelten Liste selektieren kann.

	erste Ebene	zweite Ebene
Axiom	`l.i`	`l.i.j`
Maxima	`part(l,i)` oder `l[i]`	`part(l,i,j)` oder `l[i][j]`
Maple	`op(i,l)` oder `l[i]`	`op([i,j],l)` oder `l[i,j]`
Mathematica	`l[[i]]`	`l[[i,j]]`
Reduce	`part(l,i)`	`part(l,i,j)`
Sage	`l[i]`	`l[i][j]`

Tabelle 6: Zugriffsoperatoren auf Listenelemente in verschiedenen CAS

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 l[i]
```

wobei `nops(l)` die Länge der Liste l zurückgibt und mit `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.

Listengenerierung und -transformation

Daneben spielen **Listenmanipulation** eine wichtige Rolle, insbesondere deren Generierung, selektive Generierung und uniforme Transformation. Zur Erzeugung von Listen gibt es in allen CAS

einen **Listengenerator**, der eine Liste aus einzelnen Elementen nach einer Bildungsvorschrift generiert. REDUCE hat dazu die Syntax von `for` so erweitert, dass ein Wert zurückgegeben wird.

AXIOM	`[i^2 for i in 1..5]`
MAXIMA	`makelist(i^2,i,1,5)`
MAPLE	`[seq(i^2,i=1..5)]`
MATHEMATICA	`Table[i^2, {i,1,5}]`
REDUCE	`for i:=1:5 collect i^2`
SAGE	`[i^2 for i in (1..5)]`

Tabelle 7: Liste der ersten 5 Quadratzahlen erzeugen

Bei der **selektiven Listengenerierung** möchte man aus einer bereits vorhandenen Liste alle Elemente mit einer gewissen Eigenschaft auswählen. Die meisten CAS haben dafür einen eigenen Select-Operator, der als Argumente eine boolesche Funktion und eine Liste nimmt und die gewünschte Teilliste zurückgibt. Ein solcher Operator kann sich auch hinter einer Infixnotation verbergen wie im Fall von AXIOM. Einzig REDUCE nutzt für diesen Zweck eine Kombination aus Listengenerator und Listen-Konkatenation `join` sowie die einen Wert zurückgebende `if`-Anweisung.

AXIOM	`[i for i in 1..50 | prime?(i)]`
MAXIMA	`sublist(makelist(i,i,1,50),primep)`
MAPLE	`select(isprime,[seq(i,i=1..50)])`
MATHEMATICA	`Select[Range[1,50],PrimeQ]`
REDUCE	`for i:=1:50 join if primep(i) then {i} else {}`
SAGE	`[i for i in (1..50) if is_prime(i)]`

Tabelle 8: Primzahlen bis 50 in einer Liste aufsammeln

Uniforme Listentransformationen treten immer dann auf, wenn man auf alle Elemente einer Liste ein und dieselbe Funktion anwenden möchte.

Viele CAS distributieren eine Reihe von Funktionen, die nur auf einzelne Listenelemente sinnvoll angewendet werden können, automatisch über Liste. So wird etwa von MAXIMA hier offensichtlich die Transformation

$$\texttt{float} \circ \texttt{list} \to \texttt{list} \circ \texttt{float}$$

angewendet.

```
u:makelist(sin(i),i,1,3);
```

$$[\sin(1), \sin(2), \sin(3)]$$

```
float(u);
```

$$[0.8414709, 0.9092974, 0.14112001]$$

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.
Bezeichnung und Syntax dieser Transformation lauten in allen CAS sinngemäß `map(f,l)` für die Anwendung einer Funktion f auf die Elemente einer Liste l.

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

$$[1, 2, 3]$$

```
f(u);
map(f,u);
```

$$f([1, 2, 3])$$

$$[f(1), f(2), f(3)]$$

In Anwendungen spielen daneben noch verschiedene Varianten des Zusammenbaus einer Ergebnisliste aus mehreren Ausgangslisten eine Rolle. Hier sind insbesondere zu nennen:

- das Aneinanderfügen der Elemente einer Liste von Listen (Join), das die Verschachtelungstiefe um Eins verringert,

- und ein „Reißverschlussverfahren“ (Zip) der parallelen Listentraversion, welches eine Liste von n-Tupeln erstellt, die durch eine gegebene Funktion f aus den Elementen an je gleicher Position in den Listen $l_1, \ldots, l_n$ erzeugt werden. Obwohl eine solche Funktion auch durch Generierungs- und Zugriffsoperatoren als (MAXIMA)

  ```
  makelist(f(l1[i],...,ln[i]),i,1,length(l1))
  ```

  implementiert werden kann, stellen einige CAS aus Geschwindigkeitsgründen eine spezielle Zip-Funktion zur Verfügung (MAPLE) oder erlauben `map` mit mehrstelligen Funktionen (MAXIMA, SAGE).

Die Listentransformationen `map`, `select`, `join` und `zip` bezeichnen wir als **elementare Listentransformationen**.

	Mapping	Join	Zip
AXIOM	`map(f,l)`		
MAXIMA	`map(f,l)`	`apply(append,l)`	`map(f,`l_1`,...,`l_n`)`
MAPLE	`map(f,l)`	`map(op,l)`	`zip(f,`l_1`, `l_2`)`
MATHEMATICA	`Map[f,l]`	`Flatten[l,1]`	`Thread[f[`l_1`,...,`l_n`]]`
REDUCE	`map(f,l)`	`for each x in l join x`	
SAGE	`map(f,l)`		`map(f,`l_1`,...,`l_n`)`

Tabelle 9: Weitere (weniger) elementare Listentransformationen

Ein komplexes Beispiel

Zur Illustration wird nun in einem komplexen Beispiel das Zusammenwirken der verschiedenen Listentransformationen am Beispiel des CAS MAXIMA demonstriert.

Gleichungen über Restklassenringen lassen sich lösen, indem alle möglichen (endlich vielen) Reste nacheinander in die Gleichung eingesetzt werden. Bestimmen wir als Beispiel alle Lösungen der Kongruenz $x^3 + x + 1 \equiv 0 \pmod{31}$.

Zunächst erstellen wir eine Wertetafel der Funktion.

```
werteTafel:makelist([x,mod(x^3+x+1,31)],x,0,30);
```

$$[\ [0,1],[1,3],[2,11],[3,0],[4,7],[5,7],[6,6],[7,10],[8,25],[9,26],[10,19],[11,10],[12,5],$$
$$[13,10],[14,0],[15,12],[16,21],[17,2],[18,23],[19,28],[20,23],[21,14],[22,7],[23,8],$$
$$[24,23],[25,27],[26,26],[27,26],[28,2],[29,22],[30,30]\ ]$$

Wir sehen, dass genau für die Reste $x = 3$ und $x = 14$ der Funktionswert gleich 0 ist. Diese beiden Elemente können mit einem `select`-Kommando ausgewählt werden.

```
u:sublist(werteTafel,
      lambda([v],v[2]=0));
```

$$[\ [3,0],[14,0]\ ]$$

Schließlich extrahieren wir mit `map` die Liste der zugehörigen x-Werte aus der Liste der Paare.

```
map(first,u);
```

$$[3,14]$$

Eine kompakte Lösung der Aufgabe lautet also:

```
sublist(makelist(i,i,0,30),lambda([x],mod(x^3+x+1,31)=0));
```

Diese Lösung für $p = 31$ kann leicht auf andere Primzahlen p verallgemeinert werden. Dazu definieren wir eine Funktion `sol`, mit der wir uns einen Überblick über die Nullstellen des Polynoms $x^3 + x + 1 \in \mathbb{Z}_p[x]$ für verschiedene Primzahlen p verschaffen können:

```
sol(p):=sublist(makelist(i,i,0,p-1),lambda([x],mod(x^3+x+1,p)=0));
```

Nun wird diese Funktion auf die Primzahlen kleiner als 50 angewendet, die vorher in einer Liste `primeList` aufgesammelt werden. Zur besseren Übersicht sind in der Ergebnisliste `solutionList` *Paare* `[p,sol(p)]` enthalten.

```
primeList:sublist(makelist(i,i,1,50),primep);
solutionList:map(lambda([p],[p,sol(p)]),primeList);
```

$$\left[\ [2,[\,]],[3,[1]],[5,[\,]],[7,[\,]],[11,[2]],[13,[7]],[17,[11]],[19,[\,]],[23,[4]],[29,[26]],\right.$$
$$\left.[31,[3,14]],[37,[25]],[41,[\,]],[43,[38]],[47,[25,34,35]]\ \right]$$

Wir erkennen, dass die Gleichung für verschiedene p keine (etwa für $p = 2$), eine (etwa für $p = 3$), zwei (für $p = 31$) oder drei (für $p = 47$) verschiedene Lösungen haben kann. Dem entspricht eine Zerlegung des Polynoms $P(x) = x^3 + x + 1$ über dem Restklassenkörper $\mathbb{Z}_p$ in Primpolynome: Im ersten Fall ist $P(x)$ irreduzibel, im zweiten zerfällt es in einen linearen und einen quadratischen Faktor und in den letzten beiden Fällen in drei Linearfaktoren, wobei im vorletzten Fall eine der Nullstellen eine doppelte Nullstelle ist. Mit `map` und `factor` kann das wie folgt nachgeprüft werden.

```
res:map(lambda([p],[p,ev(factor(x^3+x+1),modulus=p)]),primeList);
```

$$\left[\ \left[2, x^3+x+1\right], \left[3, (x-1)\left(x^2+x-1\right)\right], \left[5, x^3+x+1\right], \left[7, x^3+x+1\right],\right.$$
$$\left[11, (x-2)\left(x^2+2\,x+5\right)\right], \left[13, (x+6)\left(x^2-6\,x-2\right)\right], \left[17, (x+6)\left(x^2-6\,x+3\right)\right],$$
$$\left[19, x^3+x+1\right], \left[23, (x-4)\left(x^2+4\,x-6\right)\right], \left[29, (x+3)\left(x^2-3\,x+10\right)\right],$$
$$\left[31, (x-3)\,(x-14)^2\right], \left[37, (x+12)\left(x^2-12\,x-3\right)\right], \left[41, x^3+x+1\right],$$
$$\left.\left[43, (x+5)\left(x^2-5\,x-17\right)\right], [47, (x+12)\,(x+13)\,(x+22)]\ \right]$$

Substitutionslisten

Mehrere der Konstrukte, die wir in diesem Kapitel kennengelernt haben, spielen in Substitutionslisten zusammen, welche die Mehrzahl der CAS als Ausgabeform des `solve`-Operators verwendet.

Betrachten wir etwa die Ausgabe, die Maple beim Lösen des Gleichungssystems

$$\left[x^2 + y = 2,\ y^2 + x = 2\right]$$

produziert.
Maple verwendet aus Gründen, die wir später noch kennenlernen werden, hier selbst Quadratwurzeln nicht von sich aus.

```
sys:=[x^2+y=2, y^2+x=2];
s:=solve(sys,[x,y]);
```

$$\left[\ [y=-2,\, x=-2],\ [y=1,\, x=1],\right.$$
$$\left[y = \text{RootOf}\left(\_Z^2 - \_Z - 1\right),\right.$$
$$\left.\left.x = 1 - \text{RootOf}\left(\_Z^2 - \_Z - 1\right)\ \right]\right]$$

Wir wenden deshalb noch auf jeden einzelnen Eintrag der Liste `s` die Funktion `allvalues` an, die einen RootOf-Ausdruck, hinter dem sich mehrere Nullstellen verbergen, in die entsprechenden Wurzelausdrücke oder, wenn dies nicht möglich ist, in numerische Näherungslösungen aufspaltet.

```
sol:=map(allvalues,s);
```

$$\left[[y=-2,\, x=-2]\,,\ [y=1,\, x=1]\,,\ \left[y=\frac{1}{2}\sqrt{5}+\frac{1}{2},\, x=\frac{1}{2}-\frac{1}{2}\sqrt{5}\right],\ \left[y=\frac{1}{2}-\frac{1}{2}\sqrt{5},\, x=\frac{1}{2}\sqrt{5}+\frac{1}{2}\right]\right]$$

Beachten Sie, dass `s` als Liste aus 3 Elemente in eine Liste `sol` von 4 Elementen expandiert. Deren mathematisch ansprechende Form (Verwendung des Gleichheitszeichens) ist auch aus programmiertechnischer Sicht günstig. Wir können jeden einzelnen Eintrag der Liste als lokale Variablensubstitution in komplexeren Ausdrücken verwenden. Eine solche Art von Liste wird als *Substitutionsliste* bezeichnet.

Wollen wir etwa durch die Probe die Richtigkeit der Rechnungen prüfen, so können wir nacheinander jeden Listeneintrag aus `sol` in `sys` substituieren und nachfolgend vereinfachen oder gleich

```
subs(sol[1],sys);
```

$$[2=2, 2=2]$$

```
for v in sol do expand(subs(v,sys)) od;
```

berechnen. Allerdings ist letzteres nicht sehr weitsichtig, denn damit werden die Ergebnisse der Rechnungen nur auf dem Bildschirm ausgegeben und nicht für die weitere Verarbeitung gespeichert.

> Sie sollten deshalb Ergebnisse stets als Datenaggregation erzeugen, mit der später weitergerechnet werden kann. Ebenso sollte vermieden werden, auf vorherige Ergebnisse mit dem Operator `last` (% in den meisten CAS) zuzugreifen. Da sich der Wert von `last` dauernd ändert, ist hiermit keine stabile Referenz möglich.

In obiger Situation ist es also sinnvoller, die Ergebnisse der Probe in einer Liste aufzusammeln:

```
probe:=map(v -> expand(subs(v,sys)),sol);
```

$$[[2=2, 2=2], [2=2, 2=2], [2=2, 2=2], [2=2, 2=2]]$$

Die booleschen Ausdrücke $2 = 2$ werden aus noch zu erläuternden Gründen nur sehr zögerlich ausgewertet. Eine Auswertung kann mit `evalb(2=2)` erzwungen werden. Allerdings vertauscht `evalb` nicht mit Listen, so dass eine entsprechende Transformation von `probe` explizit über zwei `map`-Kommandos angeschrieben werden muss:

```
map(u->map(evalb,u)),probe);
```

$$[\,[\texttt{true}, \texttt{true}], [\texttt{true}, \texttt{true}], [\texttt{true}, \texttt{true}], [\texttt{true}, \texttt{true}]\,]$$

Alternativ kann man die Einträge jeder Liste `[2=2,2=2]` und-verknüpfen und so für jede der vier Proben ein einziges `true` oder `false` erzeugen.

```
map(u->convert(u,`and`),probe);
```

$$[\texttt{true}, \texttt{true}, \texttt{true}, \texttt{true}]$$

Aus solchen Substitutionslisten lassen sich auf einfache Weise abgeleitete Ausdrücke zusammenstellen. So liefert das folgende Kommando die Menge aller Lösungspaare in der üblichen Notation:

```
map(u -> subs(u,[x,y]), sol);
```

$$\left[[-2,\,-2]\,,\;[1,\,1]\,,\;\left[\frac{1+\sqrt{5}}{2},\,\frac{1-\sqrt{5}}{2}\right],\;\left[\frac{1-\sqrt{5}}{2},\,\frac{1+\sqrt{5}}{2}\right]\right]$$

Zu jeder der Lösungen kann auch die Summe der Quadrate und die Summe der dritten Potenzen berechnet werden. Hier ist gleich die Berechnung der Potenzen bis zum Exponenten 5 zusammengefasst. Alle diese Rechnungen ergeben ganzzahlige Werte.

```
[seq(map(u -> expand(subs(u,x^i+y^i)), sol), i=1..5)];
```

$$[\,[-4,2,1,1],[8,2,3,3],[-16,2,4,4],[32,2,7,7],[-64,2,11,11]\,]$$

Wir haben hierbei wesentlich davon Gebrauch gemacht, dass bis auf MATHEMATICA die einzelnen CAS nicht zwischen der mathematischen Relation $A = B$ (`A equal B`) und dem Substitutionsoperator $x = A$ (`x replaceby A`) unterscheiden. MAXIMA, MAPLE, MATHEMATICA und REDUCE geben ihre Lösungen für Gleichungssysteme in mehreren Variablen als solche Substitutionslisten zurück. MAXIMA, MATHEMATICA und REDUCE verwenden diese Darstellung auch für Gleichungen in einer Variablen, MAPLE dagegen in diesem Fall nur, wenn Gleichungen und Variablen als einelementige *Mengen oder Listen* angegeben werden.

Obige Rechnungen können wie folgt auch mit MAXIMA ausgeführt werden.

```
sys:[x^2+y=2, y^2+x=2];
/* Lösung bestimmen */
sol:solve(sys,[x,y]);
```

$$\left[\left[x=-\frac{\sqrt{5}-1}{2},\,y=\frac{\sqrt{5}+1}{2}\right],\;\left[x=\frac{\sqrt{5}+1}{2},\,y=-\frac{\sqrt{5}-1}{2}\right],\right.$$
$$\left.[x=-2,\,y=-2]\,,\;[x=1,\,y=1]\right]$$

```
/* Probe ausführen */
probe:map(lambda([u],expand(subst(u,sys))), sol);
```

$$[\,[2=2,2=2],[2=2,2=2],[2=2,2=2],[2=2,2=2]\,]$$

```
/* Boolesche Konjunktion aller Ausdrücke dieser Listem */
map(every, probe);
```

$$[\,\texttt{true},\texttt{true},\texttt{true},\texttt{true}\,]$$

```
/* Berechnung von x^i+y^i für die 4 Lösungen und i=1..20 */
makelist(map(lambda([u], expand(subst(u,x^i+y^i))), sol), i, 1, 20);
```

$$[\,[1,1,-4,2],\;[3,3,8,2],\;[4,4,-16,2],\;\ldots,\;[9349,9349,-1048576,2],\;[15127,15127,2097152,2]\,]$$

Boolesche Ausdrücke und Steuerstrukturen

Auch boolesche Funktionen können in der ganzen Vielfalt von Formen auftreten, in welcher Funktionen im symbolischen Rechnen generell vorkommen. Boolesche Funktionsausdrücke der Logik erster Stufe, d. h. solche Ausdrücke, die nur Operatoren `and`, `or`, `not` usw. sowie Vergleichsoperatoren für arithmetische Ausdrücke enthalten, bezeichnen wir auch kurz als **boolesche Ausdrücke**. Diese können Variablen im Symbolmodus enthalten und in diesem Fall nicht als boolesche Funktions*aufrufe* verwendet werden, solange die freien Variablen nicht mit Werten belegt sind.

Eine solche Auswertung (zu einem Wahrheitswert `true` oder `false`) ist aber erforderlich, wenn ein boolescher Ausdruck als Testbedingung in einer klassischen Steuerstruktur verwendet wird. In klassischen Steuerstrukturen können als Testbedingung also nur boolesche Ausdrücke ohne freie Variablen vorkommen: Derartige Ausdrücke wollen wir als **boolesche Formeln** bezeichnen. Wir werden weiter unten sehen, dass selbst eine derartige Beschränkung der Semantik von Testbedingungen nicht ausreicht, um mit klassischen Steuerstrukturen „klassisch“ umzugehen.

Dennoch verwenden die meisten CAS das Steuerstrukturkonzept klassischer Programmiersprachen, welches mit jeder Auswertung der jeweiligen Steuerstruktur die Auswertung der Testbedingung im Sinne einer booleschen Formel erfordert. Kann die Testbedingung nicht ausgewertet werden, so wird mit einer Fehlermeldung abgebrochen.

Axiom	a is declared as being in Integer but has not been given a value.
Maple	Error, cannot determine if this expression is true or false: $1 < a$
Mathematica	`If[a > 1, 1, 2]`
Maxima	`if a>1 then 1 else 2`
Reduce	$a - 1$ invalid as number
Sage	2

Tabelle 10: Ergebnis der Auswertung der `If`-Anweisung
`if (a>1) then 1 else 2`

Einzig Mathematica und Maxima verfolgen ein anderes, aber naheliegendes Konzept – „everything is an expression“. Auch Steuerstrukturen wie `If[...]` oder `While[...]` werden als Ausdrücke behandelt, für die bei teilweiser Auswertung ein *Funktionsausdruck* mit dem entsprechenden Funktionskopf `If` oder `While` zurückgegeben wird. Ein solcher Ausdruck kann weiter ausgewertet werden, wenn später mehr Informationen vorliegen.

Maxima:

```
u:if x>0 then 1 else 2

          if x>0 then 1 else 2

subst(x=2,u),eval;

                    1

subst(x=-2,u),eval;

                    2
```

„Lazy evaluation“ bezieht sich dabei also nicht nur auf die Möglichkeit,

- dass Variablenbezeichner im Symbolmodus bei späterer Auswertung im Wertmodus auftreten können oder
- dass Funktionsbezeichner in Funktionsausdrücken bei späterer Auswertung Funktionsaufrufe auslösen können, sondern
- dass auch Ablaufstrukturen bei späterer Auswertung neu durchlaufen werden können.

Verfolgen CAS das klassische Steuerstrukturkonzept, so müssen auch boolesche Ausdrücke mit freien Variablen als boolesche *Formeln* interpretiert werden, wenn sie als Testbedingungen in

Steuerstrukturen auftreten. Das geschieht auf die in der Logik übliche Weise – die Ausdrücke werden so interpretiert, als ob die freien Variablen durch All-Quantoren gebunden sind. Unerwartetes Auswerteverhalten von Steuerstrukturen kann meist auf diese Weise schnell erklärt werden.

Ansonsten werden boolesche Ausdrücke sehr vorsichtig behandelt, da die genaue Semantik der entsprechenden Formeln („Prüfe Bedingungen“, „Löse Gleichungen“ usw.) vielfältig sein kann.

So wird Gleichheit in den verschiedenen CAS sowohl bei der Formulierung eines Gleichungssystems als auch dessen Lösung verwendet (hier exemplarisch in MAXIMA).
In beiden Kontexten wird `=` als Operatorsymbol verwendet, aus dem ein syntaktisches Objekt „Gleichung“ als Funktionsausdruck konstruiert worden ist.

```
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]);
```

$$\left[\left[x = -\frac{1}{5},\ y = \frac{4}{5}\right]\right]$$

Die meisten CAS verfahren mit symbolischen Ausdrücken der Form $a = b$ auf ähnliche Weise. Eine boolesche Auswertung wird in einem booleschen Kontext automatisch vorgenommen oder kann in einigen CAS durch spezielle Funktionen (MAPLE: `evalb`, MUPAD und MAXIMA: `is`, SAGE: `bool`) erzwungen werden. MATHEMATICA kennt sogar zwei Vergleichsoperatoren: `==` (`Equal`) und `===` (`SameQ`).
Allerdings entsprechen die Ergebnisse nicht immer den Erwartungen (MAXIMA, ähnlich auch die anderen CAS).

```
1=2;
```

$$1 = 2$$

aber

```
if 1=2 then yes else no;
```

```
no
```

Mit dieser Auswertefunktion kann man auch genauer untersuchen, wie die einzelnen CAS einen booleschen Ausdruck als boolesche Formel interpretieren.

So ist im ersten Fall die Antwort für alle Variablenbelegungen $(x, y) = (t, 3 - t)$ falsch, aber der Ausdruck wurde ja auch nicht als „Löse die Gleichung“, sondern als Formel

$$\forall x\, \forall y\ (x + y = 3)$$

interpretiert.

```
is(x+y=3);
```

```
false
```

Im zweiten Fall sind linke und rechte Seite *syntaktisch* (literal) gleich, so dass die *semantische* Gleichwertigkeit der beiden Seiten der Gleichung offensichtlich ist.

```
is(x+y=x+y);
```

```
true
```

Im dritten Beispiel sind die beiden Seiten der Gleichung *nach Auswertung* syntaktisch gleich, im vierten Beispiel besteht auch nach der Auswertung semantische, nicht aber syntaktische Gleichheit. Anders nur SAGE, das im letzten Beispiel ebenfalls `true` zurückliefert.

```
is(x+x=2*x);
```

```
true
```

```
is(x*(x+1)=x*x+x);
```

```
false
```

Wir sehen also, dass sich die Interpretation eines mit = angeschriebenen booleschen Ausdrucks als boolesche Formel auf die *syntaktische* Gleichheit (nach Auswertung) beschränkt, semantisch gleichwertige Ausdrücke für ein korrektes Verhalten der Steuerstruktur also vorher vom Anwender in syntaktisch gleiche Ausdrücke transformiert werden müssen. Wir sehen im nächsten Kapitel, dass mehr auch nicht zu erwarten ist, da bewiesen werden kann, dass das Problem der Transformation semantisch gleichwertiger Ausdrücke in syntaktisch gleiche Form selbst für überschaubare Klassen von Ausdrücken algorithmisch nicht lösbar ist.

Die meisten CAS verwenden bei der Auswertung boolescher Formeln eine dreiwertige Logik – wie etwa die MAXIMA-Funktion `is`, die einen der drei Werte `true`, `false` oder `unknown` zurückgibt.

Beachten Sie dabei, dass die klassische Formel

$$(A)\ \texttt{or}\ (\texttt{not}\ A)$$

nicht mehr automatisch zu `true` auswerten muss, sondern wie in diesem Fall zu

$$\texttt{unknown or unknown} = \texttt{unknown}$$

auswerten kann.

```
is(a>1);
                    unknown
not(is(a>1));
                    unknown
(is(a>1) or not(is(a>1)));
                    unknown
```

Auswertung Boolescher Formeln über den reellen Zahlen

Wir kommen auf ein weiteres Problem der exakten Auswertung selbst von booleschen Formeln zu sprechen, die allein Vergleiche von reellen Zahlen enthalten.

Einfache Größenvergleiche lassen sich für derartige Ausdrücke oft erfolgreich ausführen.

```
is(1<sqrt(5));
                    true
```

Für eine *exakte* mathematische Ableitung dieser Eigenschaft hätte man korrekterweise wie folgt argumentieren müssen:

$$0 < 1 < 5 \quad \Rightarrow \quad 1 = \sqrt{1} < \sqrt{5}\,,$$

wobei – genau genommen – zusätzlich die Monotonieeigenschaft der Wurzelfunktion eine Rolle spielt. MAXIMA nimmt, wie andere CAS auch, an dieser Stelle schlicht einen numerischen Vergleich beider Seiten vor.

Allerdings steht bei einer solchen Bestimmung des booleschen Werts von Zahlenvergleichen das *prinzipielle Problem*, numerisch sehr nahe beieinander liegende Werte von exakt gleichen, aber syntaktisch verschiedenen Zahlausdrücken zu unterscheiden. Es lassen sich beliebig komplizierte Wurzelausdrücke konstruieren, die ganzen Zahlen nahe kommen, aber von ihnen verschieden sind. Numerisch kann das Auseinanderfallen erst durch sehr genaue Berechnung mit vielen Nachkommastellen bestätigt und exakte Gleichheit sowieso nicht gezeigt werden.

Beispiel: Die Folgenglieder $a_n = \alpha^n + \beta^n$ mit $\alpha = 1 + \sqrt{2}$ und $\beta = 1 - \sqrt{2}$ sind sämtlich ganzzahlig, da sich in der Expansion nach der binomischen Formel die Summanden, welche $\sqrt{2}$ mit ungeraden Exponenten enthalten, gerade wegheben. Wegen $|\beta| < 1$ kommt dabei α^n der ganzen Zahl a_n beliebig nahe, wobei die Annäherung alternierend von unten und von oben erfolgt. Wir wollen prüfen, in welchem Umfang die verschiedenen CAS $\alpha^n - a_n > 0$ korrekt entscheiden.

a_n kann durch Expandieren der beiden Potenzen direkt berechnet und daraus eine `ichallenge` gebaut werden. Hier der entsprechende MAXIMA-Code.

```
a(n):=expand((1+sqrt(2))^n+(1-sqrt(2))^n);
s:1+sqrt(2);

ichallenge(n):=(s^n-a(n)>0);
ichallenge(10);
```

$$\left(\sqrt{2}+1\right)^{10} - 6726 > 0$$

MAXIMA berechnet in den Standardeinstellungen die Ergebnisse bis $n = 20$ korrekt, danach wird der Vorzeichenwechel nicht mehr erkannt. Auch die Änderung der Präzision der numerischen Rechnungen hat nur beschränkten Erfolg.

```
makelist(is(ichallenge(i)),i,11,20);
```

$$[\texttt{true}, \texttt{false}, \texttt{true}, \texttt{false}, \texttt{true}, \texttt{false}, \texttt{true}, \texttt{false}, \texttt{true}, \texttt{false}]$$

```
makelist(is(ichallenge(i)),i,21,30);
```

$$[\texttt{false}, \texttt{false}, \texttt{false}, \texttt{false}, \texttt{false}, \texttt{false}, \texttt{false}, \texttt{false}, \texttt{false}, \texttt{false}]$$

MATHEMATICA berechnet in den Standardeinstellungen die Ergebnisse bis etwa $n = 90$ korrekt, danach wird darauf hingewiesen, dass die intern eingestellte Präzision nicht mehr ausreicht, um das Ergebnis sicher zu berechnen, und das bisherige Ergebnis der Rechnung wird in symbolischer Form zurückgegeben.

```
Table[iChallenge[n], {n, 70, 80}]
```

$$\{\texttt{False}, \texttt{True}, \texttt{False}, \texttt{True}, \texttt{False}, \texttt{True}, \texttt{False}, \texttt{True}, \texttt{False}, \texttt{True}, \texttt{False}\}$$

```
iChallenge[90]
```

```
Greater::meprec: Internal precision limit $MaxExtraPrecision = 50.‘ reached
while evaluating ...
```

$$-2817161083671398345489218335204499 8 + (1 + \texttt{Sqrt}[2])^{90} > 0$$

MAPLE berechnet in den Standardeinstellungen die Ergebnisse bis etwa $n = 200$ korrekt, danach wird `FAIL` zurückgegeben.

```
[seq(is(ichallenge(i)),i=150..160)];
```

$$[\texttt{false}, \texttt{true}, \texttt{false}, \texttt{true}, \texttt{false}, \texttt{true}, \texttt{false}, \texttt{true}, \texttt{false}, \texttt{true}, \texttt{false}]$$

```
is(ichallenge(250));
```

```
FAIL
```

Mit AXIOM kann die Frage nicht studiert werden, da s vom Typ `AlgebraicNumber` ist, aber auf diesem Bereich keine Operation `<` definiert ist. Die Frage $s^n = a_n$ kann aber untersucht werden.
`ichallenge(300)` gibt s^{300} sofort in expandierter Form als $u_n + v_n\sqrt{2}$ zurück, der Coerce-Operator `::` versucht, die Gleichung als Boolesche Formel zu interpretieren. Dazu ist nur $v_n = 0$ zu testen.

```
a(n) == (1+sqrt(2))^n+(1-sqrt(2))^n
s:=1+sqrt(2)
ichallenge(n) == s^n = a(n)
```

```
ichallenge(300)::Boolean
```

```
                                false
```

So hätten die anderen CAS auch vorgehen bzw. durch einen gezielten Nutzerhinweis auf diese Spur gebracht werden können, etwa MAXIMA. Die Ergebnisqualität wird dennoch nur wenig besser.

```
ichallenge(n):=(expand(s^n)-a(n)>0);
makelist(is(ichallenge(i)),i,45,50);
```

```
[false, true, true, true, true, true]
```

REDUCE reagiert ähnlich – `expand` ist nicht erforderlich, da die Ausdrücke in der Standardeinstellung bereits expandiert werden.

```
s:=1+sqrt(2);
algebraic procedure a(n);
  (1+sqrt(2))^n+(1-sqrt(2))^n ;
algebraic procedure ichallenge(n);
  if (s^n-a(n)>0) then true else false;
ichallenge(30);
```

```
     ***** 107578520350*sqrt(2) - 152139002499 invalid as number
```

```
algebraic procedure ichallenge(n);
  if (s^n-a(n)=0) then true else false;
ichallenge(300);
```

```
                                false
```

Die Entscheidung, dass zwei Zahlen exakt gleich sind, kann auf diese Weise *prinzipiell* nicht getroffen werden und wird hier von MAXIMA auch falsch beantwortet, denn es gilt $w = 6$, wie MAPLE bei der Eingabe von w ohne weiteres Zutun feststellt.

```
w:sqrt(11+6*sqrt(2))+sqrt(11-6*sqrt(2));
map(is,[w=6, w>6]);
```

```
                               [false, true]
```

Auch AXIOM gibt an dieser Stelle eine falsche Antwort.

```
(w=6)::Boolean
```

```
                                false
```

Dieses Beispiel gehört zu einer Serie von Wurzelausdrücken, die exakt mit ganzen Zahlen oder einfacheren Wurzelausdrücken übereinstimmen, was aber in keiner Weise offensichtlich ist. So gilt

zum Beispiel

$$\begin{aligned}\sqrt{11+6\sqrt{2}}+\sqrt{11-6\sqrt{2}} &= 6\\ \sqrt{5+2\sqrt{6}}+\sqrt{5-2\sqrt{6}} &= 2\sqrt{3}\\ \sqrt{5+2\sqrt{6}}-\sqrt{5-2\sqrt{6}} &= 2\sqrt{2}\end{aligned}$$

Weitere Serien interessanter Herausforderungen ergeben sich aus trigonometrischen Identitäten wie

$$\cos\left(\frac{\pi}{2\,n+1}\right)+\cos\left(\frac{3\,\pi}{2\,n+1}\right)+\ldots+\cos\left(\frac{(2\,n-1)\,\pi}{2n+1}\right)=\frac{1}{2}\,,$$

die sich mit MuPAD und folgender Funktion testen lassen:

```
cosChallenge:=n->_plus(cos((2*i+1)*PI/(2*n+1))$i=0..n-1);
```

Die MuPAD-Antwort im Fall $n = 3$ ist zwar korrekt, aber ebenfalls allein durch einen numerischen Vergleich gefunden worden.

```
simplify(cosChallenge(3));
```

$$\cos\left(\frac{\pi}{7}\right)-\cos\left(\frac{2\,\pi}{7}\right)+\cos\left(\frac{3\,\pi}{7}\right)$$

```
is(cosChallenge(3)=1/2);
```

```
TRUE
```

Maple (mit `simplify`) und Mathematica (mit `FullSimplify`) finden die entsprechende Vereinfachung für konkrete Werte von n.

Derartige Probleme muss ein CAS bei der Auswertung boolescher Formeln korrekt behandeln können. Verwendet ein CAS eine dreiwertige Logik mit `true`, `false` und `unknown`, so kann mit `if (expr='unknown') then ...` im Falle einer unklaren Antwort eine genauere Untersuchung gestartet werden. Dies setzt allerdings voraus, dass im jeweiligen CAS-Konzept in Situationen, die sich nicht mit den aktuell gültigen Einstellungen entscheiden lassen, auch `unknown` zurückgegeben wird. Wir haben gesehen, dass eine solche „Ehrlichkeit“ bei den Entwicklern der einzelnen CAS sehr unterschiedlich ausgeprägt ist. Die mathematisch korrektesten Antworten liefert auch hier Mathematica.

Die Beispiele zeigen weiter, dass die CAS boolesche Funktionen unterschiedlich interpretieren. Während = (`equal`) sehr vorsichtig ausgewertet wird und in fast allen Kontexten (selbst variablenfreien) ein boolescher Ausdruck zurückgegeben wird, werden andere boolesche Funktionen stärker ausgewertet. Weiter ist zu berücksichtigen, dass boolesche Funktionen in unterschiedlichen Auswertungskontexten unterschiedlich stark ausgewertet werden. Neben der Unterscheidung zwischen der eingeschränkten Auswertung im Rahmen von Substitutionskommandos (Maple, MuPAD) und der „üblichen“ Auswertung als Funktionsargument oder rechte Seite einer Zuweisung ist auch noch zu berücksichtigen, dass boolesche Ausdrücke innerhalb von Steuerablaufkonstrukten meist stärker als „üblich“ ausgewertet werden.

Kapitel 3

Das Simplifizieren von Ausdrücken

Eine wichtige Eigenschaft von CAS ist die Möglichkeit, *zielgerichtet* Ausdrücke in eine semantisch gleichwertige, aber syntaktisch verschiedene Form zu transformieren. Wir hatten im letzten Kapitel gesehen, dass solche *Transformationen* eine zentrale Rolle im symbolischen Rechnen spielen und dass dazu – wieder einmal ähnlich einem Compiler zur Compilezeit – die syntaktische Struktur von Ausdrücken zu analysieren ist.

Zum besseren Verständnis der dabei ablaufenden Prozesse ist zunächst zu berücksichtigen, dass einige zentrale Funktionen wie etwa die Polynomaddition aus Effizienzgründen als Funktionsaufrufe, zudem auf teilweise speziellen Datenstrukturen, implementiert sind und deshalb Vereinfachungen wie $(x+2)+(2\,x+3) \to 3\,x+5$ unabhängig von jeglichen Transformationsmechanismen ausgeführt werden.

Weiterhin gibt es eine Reihe von Vereinfachungen, die automatisch ausgeführt werden.
Jedoch ist nicht immer klar, in welcher Richtung eine mögliche Umformung auszuführen ist.

$$\begin{aligned} \sin(\arcsin(x)) &\to x \\ \sin(\arctan(x)) &\to \tfrac{x}{\sqrt{x^2+1}} \\ \mathrm{abs}(\mathrm{abs}(x)) &\to \mathrm{abs}(x) \end{aligned}$$

3.1 Simplifikationen als zielgerichtete Transformationen

An verschiedenen Stellen einer Rechnung können Transformationen mit unterschiedlichen Intentionen und sogar einander widersprechenden Zielvorgaben erforderlich sein.

Zur Berechnung von $\int \sin(2\,x)\,\cos(3\,x)\,d\,x$ ist es etwa angezeigt, den Ausdruck der Form $\sin(2\,x)\,\cos(3\,x)$ nach dem Additionstheorem

$$\sin(a)\,\cos(b) = \frac{1}{2}\,(\sin(a+b)+\sin(a-b))$$

in die Differenz $\frac{1}{2}\,(\sin(5\,x)-\sin(x))$ zu zerlegen, um dann diese Differenz termweise integrieren zu können.

```
int(sin(2*x)*cos(3*x),x);
```

$$\frac{\cos(x)}{2} - \frac{\cos(5\,x)}{10}$$

Um die Lösung der Gleichung $\sin(2\,x) = \cos(3\,x)$ zu bestimmen, ist es dagegen sinnvoll, nach der Umformung des Ausdrucks in $\sin(2\,x)+\sin(3\,x-\frac{\pi}{2}) = 0$ darauf das umgekehrte Additionstheorem

$$\sin(a)+\sin(b) = 2\,\sin\left(\frac{a+b}{2}\right)\sin\left(\frac{a-b}{2}\right)$$

anzuwenden, um die Differenz in das Produkt

$$2\sin\left(\frac{10\,x-\pi}{4}\right)\cos\left(\frac{2\,x-\pi}{4}\right)=0$$

zu verwandeln. Hieraus lässt sich die Lösungsmenge unmittelbar ablesen als

$$\begin{aligned}L&=\left\{\frac{\pi}{10}+\frac{2}{5}\,k\,\pi\,|\,k\in\mathbb{Z}\right\}\cup\left\{-\frac{\pi}{2}+2\,k\,\pi\,|\,k\in\mathbb{Z}\right\}\\&=\left\{\frac{\pi}{10}+2\,k\,\pi\,|\,k\in\mathbb{Z}\right\}\cup\left\{\frac{5\,\pi}{10}+2\,k\,\pi\,|\,k\in\mathbb{Z}\right\}\cup\left\{\frac{9\,\pi}{10}+2\,k\,\pi\,|\,k\in\mathbb{Z}\right\}\\&\quad\cup\left\{\frac{13\,\pi}{10}+2\,k\,\pi\,|\,k\in\mathbb{Z}\right\}\cup\left\{\frac{17\,\pi}{10}+2\,k\,\pi\,|\,k\in\mathbb{Z}\right\}\cup\left\{-\frac{\pi}{2}+2\,k\,\pi\,|\,k\in\mathbb{Z}\right\}\end{aligned}$$

in guter Übereinstimmung mit dem Ergebnis, welches MuPAD berechnet

```
solve(sin(2*x)=cos(3*x),x);
```

$$\begin{aligned}&\left\{\frac{\pi}{2}+k\,\pi\,|\,k\in\mathbb{Z}\right\}\cup\left\{\frac{\pi}{10}+2\,k\,\pi\,|\,k\in\mathbb{Z}\right\}\cup\left\{\frac{9\,\pi}{10}+2\,k\,\pi\,|\,k\in\mathbb{Z}\right\}\\&\qquad\cup\left\{\frac{13\,\pi}{10}+2\,k\,\pi\,|\,k\in\mathbb{Z}\right\}\cup\left\{\frac{17\,\pi}{10}+2\,k\,\pi\,|\,k\in\mathbb{Z}\right\}\end{aligned}$$

Ein ebenso überzeugendes Ergebnis liefert Wolfram-Alpha auf die Eingabe

```
solve sin(2*x)=cos(3*x)
```

$$\begin{aligned}x&=\pi\left(n-\frac{1}{2}\right)\ \text{and}\ n\in\mathbb{Z}\\x&=\pi\left(2\,n-\frac{7}{10}\right)\ \text{and}\ n\in\mathbb{Z}\\x&=\pi\left(2\,n-\frac{3}{10}\right)\ \text{and}\ n\in\mathbb{Z}\\x&=\pi\left(2\,n+\frac{1}{10}\right)\ \text{and}\ n\in\mathbb{Z}\\x&=\pi\left(2\,n+\frac{9}{10}\right)\ \text{and}\ n\in\mathbb{Z}\,.\end{aligned}$$

Dasselbe Ergebnis liefert MATHEMATICA mit

```
Solve[Sin[2*x]==Cos[3*x],x] // FullSimplify
```

In dem gerade betrachteten Beispiel wurde dasselbe Additionstheorem in jeweils unterschiedlicher Richtung angewendet. 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.

Eine solche *zielgerichtete Transformation* von Ausdrücken in semantisch gleichwertige *mit gewissen vorgegebenen Eigenschaften* wollen wir als **Simplifikation** bezeichnen.

In den meisten CAS gibt es für solche Simplifikationen eine Reihe spezieller Transformationsfunktionen wie `expand`, `collect`, `factor` oder `normal`, welche 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.) *lokal* auf einen Ausdruck anwenden.

Daneben existiert meist eine (oder mehrere) komplexere Funktion `simplify`, welche das Ergebnis verschiedener Transformationsstrategien miteinander vergleicht und an Hand des Ergebnisses entscheidet, welches denn nun das „einfachste“ ist.

Dies kann zu durchaus überraschenden Ergebnissen führen, wie das folgende MATHEMATICA-Beispiel zeigt:

```
u = a^n/((a-b)(a-c)) + b^n/((b-a)(b-c)) + c^n/((c-a)(c-b))

v = Table[u /. n -> i // Simplify, {i, 2, 7}]
```

$$1,\ a+b+c,\ a^2+(b+c)\,a+b^2+c^2+b\,c,$$

$$a^3+(b+c)\,a^2+\left(b^2+b\,c+c^2\right)\,a+b^3+c^3+b\,c^2+b^2c,$$

$$\frac{a^6}{(a-b)(a-c)}+\frac{\frac{b^6}{b-a}+\frac{c^6}{a-c}}{b-c},\ \frac{a^7}{(a-b)(a-c)}+\frac{\frac{b^7}{b-a}+\frac{c^7}{a-c}}{b-c}$$

Im MATHEMATICA-Hilfesystem heißt es dazu:

> There are many situations where you want to write a particular algebraic expression in the simplest possible form. Although it is difficult to know exactly what one means in all cases by the 'simplest form', a worthwhile practical procedure is to look at many different forms of an expression, and pick out the one that involves the smallest number of parts.

Diese *Anzahl von Teilen* lässt sich mit der Funktion `LeafCount` bestimmen. Für die ausgeführte Simplifikation erhält man maximal 50 Terme, während die Expansion als ganzrationale Ausdrücke für $n \geq 6$ längere Terme liefert. Das Beispiel zeigt also, dass in der Tat der „einfachste“ Ausdruck in einer wohlbestimmten Semantik gefunden wurde, auch wenn das Ergebnis möglicherweise nicht mit Ihren Erwartungen übereinstimmt.

```
LeafCount /@ v
```

$$\{1, 4, 18, 39, 50, 50\}$$

```
w = Table[u /. n -> i // Together, {i,2,7}]
LeafCount /@ w
```

$$\{1, 4, 19, 44, 79, 124\}$$

Um Vereinfachungen in verschiedenen wohldefinierten Richtungen zu erreichen, halten die CAS spezielle Transformationsfunktionen für unterschiedliche Simplifikationsaufgaben bereit. Damit kann die Simplifikationsrichtung im Laufe des interaktiven Dialogs leicht geändert werden. Nur ein kleiner Satz „allgemeingültiger“ Vereinfachungen wird automatisch durch das System ausgeführt. 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.

In diesem Kapitel werden wir deshalb untersuchen, welche Möglichkeiten CAS zur Verfügung stellen, um eigene Simplifikationsstrategien zu definieren und anzuwenden. Dabei sind zwei grundlegend verschiedene Herangehensweisen im Einsatz, ein *funktionales* (MAPLE, MUPAD) und ein *regelbasiertes Transformationskonzept* (REDUCE, MAXIMA, MATHEMATICA, AXIOM).

3.2 Das funktionale Transformationskonzept

> Beim funktionalen Konzept werden Transformationen als Funktionsaufrufe realisiert, in welchen eine genaue syntaktische Analyse der (ausgewerteten) Aufrufparameter erfolgt und danach entsprechend verzweigt wird.

Da in die Abarbeitung eines solchen Funktionsaufrufs die *Struktur* der Argumente mit eingeht, werden dazu Funktionen benötigt, welche diese Struktur wenigstens teilweise analysieren. 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 für Variablen**, sondern um eine **syntaktische Typanalyse für Ausdrücke**, die in der Regel nur die Syntax der obersten Ebene des Ausdrucks analysiert (z. B. entsprechende Schlüsselworte an der Stelle 0 der zugehörigen Liste auswertet) oder ein mit dem Bezeichner verbundenes Typschlüsselwort abgreift. Dies erkennen wir etwa an nebenstehendem Beispiel.

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

$$x$$

```
h:=exp(ln(x)+ln(y));
```

$$e^{\ln(x)+\ln(y)}$$

```
simplify(h);
```

$$x\,y$$

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

Betrachten wir als Beispiel die Definition der Exponentialfunktion in MAPLE, was mit `print(exp)` angezeigt werden kann, nachdem `interface(verboseproc=2)` gesetzt wurde:

```
proc(x::algebraic)
local res, i, t, q, n, f, r;
option builtin = HFloat_exp,
‘Copyright (c) 1992 by the University of Waterloo. All rights reserved.‘;
    if nargs <> 1 then error "expecting 1 argument, got %1", nargs
    elif type(x, ’complex(float)’) then return evalf(’exp’(x))
    elif type(x, ’rational’) then res := ’exp’(x)
    elif type(x, ’function’) and op(0, x) = ’ln’ then res := op(1, x)
    elif ...
    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); res := exp((i - t)*Pi*I)
        elif type(6*i, ’integer’) or type(4*i, ’integer’) then
            res := cos(-I*x) + sin(-I*x)*I
        else res := ’exp’(x)
        end if
    elif  ...
    elif type(x, ’function’) and nops(x) = 1 then
        n := op(0, x);
        t := op(1, x);
        if n = ’arcsinh’ then res := t + sqrt(1 + t^2)
        elif n = ’arccosh’ then res := t + sqrt(t + 1)*sqrt(t - 1)
        elif n = ’arctanh’ then res := (t + 1)/sqrt(1 - t^2)
        elif n = ’arccsch’ then res := 1/t + sqrt(1 + 1/t^2)
        elif n = ’arcsech’ then res := 1/t + sqrt(1/t - 1)*sqrt(1/t + 1)
```

```
        elif n = 'arccoth' then res := 1/sqrt((t - 1)/(t + 1))
        else res := 'exp'(x)
        end if
    elif ...
    else res := 'exp'(x)
    end if;
    exp(args) := res
end proc
```

Zunächst

```
if nargs <> 1 then error "expecting 1 argument, got %1", nargs
```

wird die Korrektheit der Anzahl der Aufrufargumente geprüft. Die meisten Zeilen des folgenden Codes beginnen mit `type(x, Art)` und analysieren den Kopfterm des aufgerufenen Arguments. Sehen wir uns die einzelnen Zeilen nacheinander an:

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

untersucht, ob x eine (reelle oder komplexe) float-Zahl ist und ruft in diesem Fall `evalf(exp(x))` auf, was nach einer durch `evalf` ausgelösten Funktionstransformation zu `` `evalf/exp`(x) `` transformiert wird. `` `evalf/fff`(x) `` definiert ein einheitliches Konzept des Aufrufs von Funktionsdefinitionen zur numerischen Auswertung von Funktionen `fff`, das zur Laufzeit erweitert werden kann.

Ehe wir uns genauer anschauen, was im Fall eines Produkts ausgeführt wird, analysieren wir die an mehreren Stellen auftretende Rückgabe `res:=` `` `exp`(x) `` genauer.

In diesem Fall wird ein Funktionsausdruck mit dem Kopf `exp` und dem *ausgewerteten* Argument x gebildet.
Am letzten Beispiel sehen wir noch einmal, dass das Argument vor dem Aufruf von `exp` wirklich ausgewertet wurde.

```
exp(2/3);
```

$$\exp\left(\frac{2}{3}\right)$$

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

$$\exp(9)$$

Die Zeile

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

untersucht, ob das Argument x ein Funktionsausdruck mit dem Kopf `ln` ist, und gibt in dem Fall das erste Element von $x = \ln(z)$, also z, zurück: $\exp(\ln(z)) = z$.

Weiter

```
elif type(x, '`*`') and type(-I*x/Pi, 'rational') then
```

wird geprüft, ob x ein Produkt ist, also mit dem Kopf `*` beginnt, und zusätzlich die Gestalt $x = y{\cdot}\pi\,\mathrm{i}$ mit $y \in \mathbb{Q}$ hat, da dann $\mathrm{e}^{y\,\pi\,\mathrm{i}}$ zu $\cos(y\,\pi) + \mathrm{i}\,\sin(y\,\pi)$ vereinfacht werden kann. Mit den Zeilen

```
y := -I*x/PI;
if 1 < y or y <= -1 then
      t := trunc(i); t := t + irem(t, 2); res := exp((y - t)*Pi*I)
```

wird untersucht, ob $-1 \le y < 1$ gilt, und anderenfalls in einem neuen Aufruf das Argument y durch $y - t$ ersetzt, wobei t eine ganze gerade Zahl mit $-1 \le y - t < 1$ ist. Dies entspricht der Anwendung der Identität $\mathrm{e}^{2\pi\,\mathrm{i}} = 1$.

```
exp(24/7*Pi*I);
```

$$\mathrm{e}^{-4/7\ \mathrm{i}\,\pi}$$

```
    elif type(6*i, 'integer') or type(4*i, 'integer') then
          res := cos(-I*x) + sin(-I*x)*I
    else res := 'exp'(x)
```

Die Verwandlung in $\cos(y\,\pi) + \mathrm{i}\,\sin(y\,\pi)$ wird nur dann vorgenommen, wenn $4\,y$ oder $6\,y$ eine ganze Zahl ist. Zusammen mit den MAPLE bekannten expliziten Werten der Winkelfunktionen an diesen Stellen ergibt sich das hier gezeigte Verhalten. Anderenfalls wird ein Funktionsausdruck mit dem Kopf `exp` zurückgegeben.

```
exp(23/6*Pi*I);
```

$$\frac{1}{2}\sqrt{3} - \frac{1}{2}\,\mathrm{i}$$

Die restlichen Zeilen realisieren die Funktionstransformationen $\exp(\mathrm{arcsinh}(t)) = t + \sqrt{1+t^2}$ usw., die sich aus den entsprechenden Funktionsdefinitionen ergibt.

```
exp(arcsinh(t));
```

$$t + \sqrt{1+t^2}$$

Das Beispiel zeigt, dass `exp` eine *Transformationsfunktion* ist und die symbolischen Fähigkeiten eines CAS eingesetzt werden können, um Polymorphie zu simulieren. Neue Funktionsbezeichner können während des Aufrufs durch Stringoperationen erzeugt werden und auf vorhandene Funktionsdefinition verweisen, wie wir beim Zusammensetzen von `evalf(exp(x))` zu `` `evalf/exp`(x) `` gesehen hatten.

Dieses Prinzip findet bei inerten MAPLE-Funktionen Anwendung. Schauen wir uns dazu noch einmal das Beispiel der Funktion `Factor` an, welche beim modularen Faktorisieren von Polynomen zu verwenden ist.

```
Factor(f) mod 2;
```

$$(x+1)^3$$

`Factor(f)` wird unverändert zurückgegeben und `mod` erkennt an der Struktur `mod(Factor(f),p)`, dass modular zu faktorisieren ist. Schauen wir uns mit `print(`mod/Factor`)` den Quellcode der speziellen modularen Faktorisierungsroutine `` `mod/Factor`(f,p) `` an, die hier aufgerufen wird.

```
proc(a)
local K, f, p;
option
`Copyright (c) 1990 by the University of Waterloo. All rights reserved.`;
    if nargs = 3 then K := args[2]; p := args[3]
    else K := NULL; p := args[2]
    end if;
    f := Factors(a, K) mod p;
    f[1]*convert(map(proc(x) x[1]^x[2] end proc, f[2]), `*`) mod p
end proc
```

Der Funktionsrumpf enthält die Kombination `Factors(a, K) mod p`, die nach denselben Regeln in `` `mod/Factors`(f,p) `` umgesetzt wird und im Wesentlichen eine Liste von Paaren aud Primfaktor und Exponent zurückgibt, die in der letzten Zeile mit `convert` in ein Produkt verwandelt wird.

Die Nachteile des funktionalen Transformationskonzepts fallen sofort ins Auge:

1. Man hat mit jedem Funktionsaufruf eine Menge verschiedener Typinformationen zu ermitteln, womit jeder Funktionsaufruf relativ aufwändige 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 redefinieren.

3.3 Das regelbasierte Transformationskonzept

Beim regelbasierten Zugang wird die im funktionalen Zugang notwendige Code-Redundanz vermieden, indem der Transformationsvorgang als `Apply(Expression, Rules)` aus einem allgemeinen Programmteil und einem speziellen Datenteil aufgebaut wird.
Der Datenteil `Rules` enthält die jeweils konkret anzuwendenden Ersetzungsregeln, also Informationen darüber, welche Kombinationen von Funktionssymbolen wie zu ersetzen sind.
Der Programmteil `Apply`, der *Simplifikator*, stellt die erforderlichen Routinen zur Mustererkennung und Unifikation bereit.

Der Simplifikator `Apply` ist also eine zweistellige Funktion, welche einen symbolischen Ausdruck A und einen Satz von *Transformationsregeln* übergeben bekommt und diese Regeln so lange auf A und die entstehenden Folgeausdrücke anwendet, bis keine Ersetzungen mehr möglich sind. Im Gegensatz zum funktionalen Zugang sind hier 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.

Damit enthält die Programmiersprache eines CAS neben funktionalen und imperativen auch Elemente einer logischen Programmiersprache. Wir werden uns in einem späteren Abschnitt genauer mit der Funktionsweise eines solchen Regelsystems vertraut machen. An dieser Stelle wollen wir uns 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. 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)}
```

Die ersten 7 Regeln ersetzen spezielle Kombinationen von fest vorgegebenen Symbolen durch andere. Solche Regeln werden auch als *spezielle Regeln* bezeichnet, denn in ihnen sind alle Bezeichner nur in ihrer literalen Bedeutung präsent.

In den weiteren Regeln kommen Bezeichner auch als *formale Parameter* als Platzhalter für beliebige Teilausdrücke vor. So vereinfacht REDUCE etwa `sin(atan(A))` zu $\frac{A}{\sqrt{1+A^2}}$, egal wie der Teilausdruck A beschaffen ist. Der Bezeichner `pi` steht dagegen für das Symbol π in seiner literalen Bedeutung. Wir haben also auch hier zwischen Bezeichnern in ihrer literalen Bedeutung (als Symbolvariable) (neben `e` sind das in obigen Regeln die verschiedenen Funktionssymbole) und Bezeichnern als Wertcontainer (hier: als formale Parameter) zu unterscheiden. Regeln mit formalen Parametern werden auch als *allgemeine Regeln* bezeichnet.

Manche der angegebenen Regeln sind noch konditional untersetzt, d. h. werden nur dann angewendet, wenn die Belegung der formalen mit aktuellen Parametern noch Zusatzvoraussetzungen erfüllt. Diese Effekte sind von regelorientierten Programmiersprachen wie etwa Prolog aber gut bekannt.

Konzeptionelle Anforderungen

1. Transformationen von Ausdrücken können über Regelanwendungen realisiert werden.

   Dazu muss das CAS eine *Mustererkennung* (pattern matcher) zur Lokalisierung entsprechender Anwendungsmöglichkeiten sowie der Zuordnung von Belegungen für die formalen Parameter bereitstellen.

2. Wie bei Funktionen ist zwischen *Regeldefinition* und *Regelanwendung* zu unterscheiden.

3. Wie bei Funktionen können in Regeldefinitionen formale Parameter auftreten. Bei Bezeichnern in einer Regeldefinition ist zu unterscheiden, ob der Bezeichner literal als Symbol für sich selbst steht oder als formaler Parameter eine Platzhalterfunktion hat.

   In den Systemen werden Bezeichner, die als Platzhalter verwendet werden, besonders gekennzeichnet. Dies kann am einfachsten geschehen, indem diese Bezeichner in einer separaten Liste $(u_1, \ldots, u_n)$ zusammengefasst werden.

4. Im Gegensatz zu Funktionen kann die Anwendung einer passenden Regel konditional sein, d. h. vom Wert einer vorab zu berechnenden booleschen Wächterbedingung (guard clause) abhängen.

   Eine Regeldefinition besteht damit aus vier Teilen: `Rule(lhs, rhs, bool)`$(u_1, \ldots, u_n)$

MATHEMATICA	`lhs /; bool → rhs`
MAXIMA	`tellsimpafter(lhs, rhs, bool)`
MUPAD	`Rule(lhs, rhs, bool)`
REDUCE	`lhs => rhs when bool`

5. Regelanwendungen haben viel Ähnlichkeit mit der Auswertung von Ausdrücken. Insbesondere ist zwischen einfachen Regelanwendungen und iterierten Regelanwendungen zu unterscheiden.
6. Das Ergebnis hängt sowohl von der Reihenfolge der Regelanwendungen als auch von der Strategie der Mustererkennung ab.

Zusammenhang mit anderen CAS-Konzepten

Regelanwendungen haben viel Ähnlichkeit mit der Auswertung von Ausdrücken:

- Nach einmaliger Regelanwendungen kann es sein, dass dieselbe oder weitere Regeln anwendbar sind bzw. werden. Es ist also sinnvoll, Regeln iteriert anzuwenden.
- Iterierte Regelanwendungen bergen die Gefahr von Endlosschleifen in sich.
- Auswertungen können als Spezialfall von Regelanwendungen betrachtet werden, da die Einträge in der Symboltabelle als spezielles Regelwerk aufgefasst werden können.

Regelanwendungen können wie Wertzuweisungen lokal oder global vereinbart werden.

- Globale Regeldefinitionen ergänzen und modifizieren das automatische Transformationsverhalten des Systems und haben damit ähnliche Auswirkungen wie globale Wertzuweisungen.
- Lokale Regelanwendungen haben viel Ähnlichkeit mit der Substitutionsfunktion, indem sie das regelbasierte Transformationsverhalten auf einen einzelnen Ausdruck beschränken.
- Substitutionen und Wertzuweisungen können als spezielle Regelanwendungen formuliert werden. Einige CAS realisieren deshalb einen Teil dieser Funktionalität über Regeln.
- Das gleiche gilt für Funktionsdefinitionen. Diese können als spezielle Regeldefinitionen realisiert werden.

3.4 Simplifikation und mathematische Exaktheit

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

Kern allgemeingültiger Simplifikationen

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

Dazu gehört zunächst einmal die Strategie, spezielle Werte von Funktionsausdrücken, sofern diese durch „einfachere“ Symbole exakt ausgedrückt werden können, durch diese zu ersetzen wie in diesen MAXIMA-Beispielen.

`sqrt(36)`	⇒	6
`sin(%pi/4)`	⇒	$\frac{1}{\sqrt{2}}$
`tan(%pi/6)`	⇒	$\frac{1}{\sqrt{3}}$
`asin(1)`	⇒	$\frac{\pi}{2}$

Dies trifft auch für kompliziertere Funktionsausdrücke zu, die auf „elementarere“ Funktionen zurückgeführt werden, in denen mehr oder weniger gut studierte spezielle mathematische Funktionen auftreten wie in diesen MAXIMA-Beispielen.

`gamma(1/2)`	$\Rightarrow$	$\sqrt{\pi}$
`integrate(exp(-x^2),x,0,inf)`	$\Rightarrow$	$\frac{\sqrt{\pi}}{2}$
`assume(y>0)$ integrate(exp(-x^2),x,0,y)`	$\Rightarrow$	$\frac{\sqrt{\pi}\,\mathrm{erf}(y)}{2}$
`sum(1/i^2,i,1,inf), simpsum`	$\Rightarrow$	$\pi^2/6$
`sum(1/i^7,i,1,inf), simpsum`	$\Rightarrow$	zeta(7)

In den Beispielen treten als Transformationsergebnis die Gamma-Funktion $\Gamma(x)$, die Gaußsche Fehlerfunktion $\mathrm{erf}(x)$ sowie die Riemannsche Zeta-Funktion $\zeta(n)$ auf.

Weiterhin wird auch eine Reihe komplizierterer Umformungen von einigen der Systeme automatisch[1] ausgeführt wie z. B.:

$$\sqrt{24} = 2\sqrt{6}, \qquad \sqrt{2\sqrt{3}+4} = \sqrt{3}+1, \qquad \sqrt{11+6\sqrt{2}} + \sqrt{11-6\sqrt{2}} = 6\,.$$

Auch werden eindeutige Simplifikationen von Funktionsausdrücken ausgeführt wie etwa die folgenden von MAXIMA

`abs(abs(x))`	$\Rightarrow$	$\lvert x\rvert$
`tan(atan(x))`	$\Rightarrow$	x
`tan(asin(x))`	$\Rightarrow$	$\frac{x}{\sqrt{1-x^2}}$
`abs(-%pi*x)`	$\Rightarrow$	$\pi\,\lvert x\rvert$
`cos(-x)`	$\Rightarrow$	$\cos(x)$
`exp(3*log(x))`	$\Rightarrow$	x^3

Auf den ersten Blick mag es deshalb verwundern, dass folgende Ausdrücke von den meisten CAS[2] nicht vereinfacht werden:

`sqrt(x^2)`	$\Rightarrow$	$\sqrt{x^2}$
`log(exp(x))`	$\Rightarrow$	$\log(\exp(x))$
`arctan(tan(x))`	$\Rightarrow$	$\arctan(\tan(x))$

In jedem der drei Fälle würde der durchschnittliche Nutzer als Ergebnis wohl x erwarten. Für die letzte Beziehung ist das allerdings vollkommen falsch, wie ein Plot der Funktion mit MAXIMA unmittelbar zeigt:

```
plot2d(atan(tan(x)),[x,-5,5]);
```

Wir sehen, dass arctan nur im Intervall $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ die Umkehrfunktion von tan ist. Die korrekte Antwort lautet für $x \in \mathbb{R}$ also

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

wobei $\lfloor a \rfloor$ für den ganzen Teil der Zahl $a \in \mathbb{R}$ 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 und MAXIMA gegeben. MAPLE allerdings gibt nach expliziter Aufforderung

`simplify(sqrt(x^2))`	$\Rightarrow$	$\mathrm{csgn}(x)\,x$

[1] MAPLE automatisch, MUPAD erst mit `radsimp`, MATHEMATICA erst mit `FullSimplify`, MAXIMA gar nicht.
[2] MAXIMA vereinfacht die ersten beiden Ausdrücke unzulässigerweise.

zurück, obwohl MAPLE auch die Betragsfunktion kennt. Der Grund liegt darin, dass das nahe liegende 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, dessen Realteil positiv ist, 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. MUPAD und MATHEMATICA vereinfachen deshalb den Ausdruck auch unter `simplify` nicht.

> Die Exaktheit von Umformungen hängt auch von der mathematischen Theorie ab, innerhalb derer die entsprechenden Ausdrücke interpretiert werden.

So ist die dritte Beziehung $\log(\exp(x)) = x$ wegen der Monotonie der beteiligten Funktionen in der Theorie der reellwertigen Funktionen $f : \mathbb{R} \to \mathbb{R}$ richtig. Für komplexe Argumente kommt aber, ähnlich wie für die Wurzelfunktion, die Mehrdeutigkeit der Logarithmusfunktion ins Spiel.

Die in der gymnasialen Oberstufe und im Grundkurs Analysis diskutierte Theorie der stetigen reellwertigen Funktionen wird bei unvorsichtigem Gebrauch von Symbolen schnell verlassen. Betrachten wir dazu die Ausdrücke

$$\sqrt{\frac{1}{x} - \frac{1}{\sqrt{x}}} \qquad \text{und} \qquad \sqrt{x{\cdot}y} - \sqrt{x}{\cdot}\sqrt{y}\,,$$

die für solche Argumente, für welche sie „sinnvoll“ definiert sind (also hier etwa für positive reelle x), zu null vereinfacht werden können. Für negative reelle Argumente verlassen wir allerdings die Theorie der stetigen reellwertigen Funktionen und haben 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 $\mathtt{i}$. Setzen wir in obigen Ausdrücken $x = y = -1$ und führen diese Ersetzung aus, so erhalten wir im ersten Fall $\mathtt{i} - 1/\mathtt{i} = 2\,\mathtt{i}$ und im zweiten Fall $\sqrt{1} - \mathtt{i}^2 = 2$. Solche Inkonsistenzen tief in komplexen Berechnungen versteckt können zu vollkommen falschen Resultaten führen, ohne dass der Grund dafür offensichtlich wird.

Aus ähnlichen Gründen sind übrigens auch die Transformationen der Logarithmusfunktion nach den bekannten Logarithmengesetzen mathematisch nicht allgemeingültig:

$$\texttt{log((-1)*(-1))=log(-1)+log(-1)} \quad \Rightarrow \quad 0 = 2\,\mathtt{i}\,\pi$$

> Mit Blick auf die Bedeutung polynomialer Strukturen im Design der CAS und dem Umstand, dass Nullstellen polynomialer Gleichungssysteme grundsätzlich komplexe Zahlen sind, interpretieren moderne CAS deshalb ihre Terme, soweit dies sinnvoll ist, in der Theorie der meromorphen Funktionen über den komplexen Zahlen.

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 ($*$ bedeutet unsimplifiziert):

Ausdruck	Maxima	Maple	Mma	MuPAD	Reduce	Sage
	5.37	2016	11.0	8.0	3.8	8.0
$\lvert -\pi \cdot x\rvert$	$\pi\,\lvert x\rvert$	$\pi\,\lvert x\rvert$	$\pi\,\lvert x\rvert$	$\pi\,\lvert x\rvert$	$\pi\,\lvert x\rvert$	$\pi\,\lvert x\rvert$
$\arctan(\tan(x))$	$*$	$*$	$*$	$*$	$*$	$*$
$\arctan(\tan(\frac{25}{7}\pi))$	$-\frac{3}{7}\pi$	$-\frac{3}{7}\pi$	$-\frac{3}{7}\pi$	$-\frac{3}{7}\pi$	(2)	(3)
$\sqrt{x^2}$	$\lvert x\rvert$	$*$	$*$	$*$	$\lvert x\rvert$	$*$
$\sqrt{x\,y}-\sqrt{x}\,\sqrt{y}$	$*$	$*$	$*$	$*$	$*$	$*$
$\sqrt{\frac{1}{z}}-\frac{1}{\sqrt{z}}$	0	$*$	$*$	$*$	(1)	0
$\log(\exp(x))$	x	$*$	$*$	$*$	x	$*$
$\log(\exp(10\,\mathrm{i}))$	$10\,\mathrm{i}$	$*$	$10\,\mathrm{i}-4\pi\,\mathrm{i}$	$10\,\mathrm{i}-4\pi\,\mathrm{i}$	$10\,\mathrm{i}$	$*$

$$(1)=\frac{\sqrt{z}\sqrt{\frac{1}{z}}-1}{\sqrt{z}},\ (2)=\arctan\left(\tan\left(\frac{4}{7}\,\pi\right)\right),\ (3)=\arctan\left(-\tan\left(\frac{3}{7}\,\pi\right)\right)$$

Tabelle 1: Simplifikationsverhalten der verschiedenen Systeme an ausgewählten Beispielen

Assume-Mechanismus

Derartigen Fragen der mathematischen Exaktheit widmen die großen CAS seit Mitte der 90er Jahre verstärkte Aufmerksamkeit. Eine natürliche Lösung ist die Einführung von **Annahmen** (assumptions) zu einzelnen Bezeichnern. Hierfür haben in den letzten Jahren die meisten der großen CAS `assume`-Mechanismen eingeführt, mit denen es möglich ist, im Rahmen der durch das CAS vorgegebenen Grenzen einzelnen Bezeichnern einen gültigen Definitionsbereich als Eigenschaft zuzuordnen.

Maxima	`declare(x,real)`
Maple	`assume(x,real)`
Mathematica	`SetOptions[Assumptions -> x ∈ Reals]`
MuPAD	`assume(x,Type::Real)`

Tabelle 2: Variable x als reell deklarieren

Die folgenden Bemerkungen mögen zunächst die Schwierigkeiten des neuen Gegenstands umreißen:

- Die Probleme, mit welchen eine solche zusätzliche logische Schicht über der Menge der Bezeichner konfrontiert ist, umfasst die Probleme eines konsistenten Typsystems als Teilfrage.
- Annahmen wie etwa $x < y$ betreffen nicht nur einzelne Bezeichner, sondern Gruppen von Bezeichnern und sind nicht kanonisch einzelnen Bezeichnern als Eigenschaft zuzuordnen.
- Selbst für eine überschaubare Menge von erlaubten Annahmen über Bezeichner (vorgegebene Zahlbereichen, Ungleichungen) führt das Inferenzproblem, d. h. die Bestimmung von erlaubten Bereichen von Ausdrücken, welche diese Bezeichner enthalten, auf mathematisch und rechnerisch schwierige Probleme wie das Lösen von Ungleichungssystemen. Unter einigermaßen allgemeinen Annahmen kann bewiesen werden, dass das Inferenzproblem algorithmisch nicht lösbar ist.

Praktisch erlaubte Annahmen beschränken sich deshalb meist auf wenige Eigenschaften wie etwa

- Annahmen über die Natur einer Variablen (`assume(x,integer)`, `assume(x,real)`),
- die Zugehörigkeit zu einem reellen Intervall (`assume(x>0)`) oder
- die spezielle Natur einer Matrix (`assume(m,quadratic)`)

Das Inferenzproblem wird stets nur im schwachen Sinne gelöst: es wird eine (ggf. keine), nicht unbedingt die strengste ableitbare Annahme gesetzt.

Mit speziellen Funktionen (MAXIMA: `facts`, `properties`, MAPLE: `about`, MUPAD: `getprop`) können die gültigen Eigenschaften ausgelesen werden.

Wie bei Regeldefinitionen können Eigenschaften global oder lokal zur Anwendung kommen. MAXIMA, MAPLE und MUPAD erlauben im Rahmen eines speziellen Assume-Mechanismus die globale Definition von Annahmen. In MATHEMATICA werden globale Annahmen als Optionen in einer Systemvariablen `$Assumptions` gespeichert und können wie andere Optionen auch global mit `SetOptions[Assumptions -> ... ]` gesetzt und modifiziert werden. MAPLE und MATHEMATICA erlauben darüber hinaus die Vereinbarung lokaler Annahmen zur Auswertung oder Vereinfachung von Ausdrücken.

MAPLE führt unter zusätzlichen Annahmen die oben beschriebenen mathematischen Umformungen automatisch aus. Ähnlich ist das Vorgehen in MUPAD oder MAXIMA.

In diesem Beispiel ist $x < 0$ und $y \in \mathbb{R}$ angenommen und mit MAXIMA angeschrieben.
Die meisten CAS vereinfachen die ersten beiden Ausdrücke, nicht jedoch den dritten, da für $x < 0$ das einfache Expandieren der Wurzel nicht korrekt ist. MAXIMA liefert die ebenfalls korrekte Vereinfachung $\sqrt{-x}\,\sqrt{-y} - \sqrt{x}\,\sqrt{y}$.

```
declare(y,real); assume(x<0);
facts(x); facts(y);

     [0 > x] [kind(y, real)]

abs(-%pi*x); sqrt(x^2);
sqrt(x*y) - sqrt(x)*sqrt(y);
```

`assume` überschreibt in MAPLE und MUPAD die bisherigen Eigenschaften, während diese Funktion in MAXIMA kumulativ wirkt. Die (zusätzliche) Annahme $x > 0$ führt in diesem Fall zu einem inkonsistenten System von Eigenschaften, weshalb zunächst `forget(x<0)` eingegeben werden muss.

Neben globalen Annahmen sind in einigen CAS auch lokale Annahmen möglich. So vereinfacht MAPLE unter der lokal mit `assuming` zugeordneten Annahme $x < 0$.

```
sqrt(x^2) assuming x<0;
```

$$-x$$

In MATHEMATICA können Annahmen über die Option `Assumptions` für die Befehle `Simplify`, `FullSimplify`, `Refine` oder `FunctionExpand` angeschrieben werden.
Diese Optionen können in der Systemvariablen `$Assumptions` global gespeichert und durch das `Assuming`-Konstrukt auch in allgemeineren Kontexten lokal erweitert werden.

```
Simplify[√(x^2), Assumptions -> {x<0}]
```

$$-x$$

```
Simplify[√x √y - √(x y),
      Assumptions -> {x∈ Reals, y>0}]
```

$$0$$

```
Assuming[x<0, Simplify[Sqrt[x^2]]]
```

$$-x$$

In MAXIMA und REDUCE lässt sich außerdem die Strenge der mathematischen Umformungen durch verschiedene Schalter verändern. So kann man 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 MAXIMA können die entsprechenden Schalter wie `logexpand`, `radexpand` oder `triginverses` sogar drei Werte annehmen: `false` (keine Simplifikation), `true` (bedachtsame Simplifikation) oder `all` (ständige Simplifikation).

Die enge Verzahnung des Assume-Mechanismus mit dem Transformationskonzept ist nicht zufällig. Die Möglichkeit, einzelnen Bezeichnern Eigenschaften aus einem Spektrum von Vorgaben zuzuordnen, macht die Sprache des CAS reichhaltiger, ändert jedoch nichts am prinzipiellen Transformationskonzept, sondern wertet nur den konditionalen Teil auf.

Die mathematische Strenge der Rechnungen, die mit einem CAS allgemeiner Ausrichtung möglich sind, ist in den letzten 20 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 ([18, 17, 23, 24]), erkennen kann. Allerdings werden selbst innerhalb eines Systems an unterschiedlichen Stellen manchmal unterschiedliche Maßstäbe der mathematischen Strenge angelegt, insbesondere bei der Implementierung komplexerer Verfahren.

3.5 Das allgemeine Simplifikationsproblem

Wir hatten gesehen, dass sich das allgemeine Simplifikationsproblem grob als **das Erkennen der semantischen Gleichwertigkeit syntaktisch verschiedener Ausdrücke** charakterisieren lässt. Wir wollen dies nun begrifflich genauer fassen.

Die Formulierung des Simplifikationsproblems

Ausdrücke in einem CAS können nach der Auflösung von Operatornotationen und anderen Ambiguitäten als geschachtelte Funktionsausdrücke in linearer, eindimensionaler Notation angesehen werden. Als *wohlgeformte Ausdrücke* (oder kurz: Ausdrücke) bezeichnen wir

- alle Bezeichner und Konstanten des Systems (atomare Ausdrücke) sowie
- Listen $[f, a, b, c, \ldots]$, deren Elemente selbst wohlgeformte Ausdrücke sind (zusammengesetzte Ausdrücke), wobei der Ausdruck entsprechend der LISP-Notationskonvention für den Funktionsausdruck $f(a, b, c, \ldots)$ steht.

Ausdrücke sind also rekursiv aus Konstanten, Bezeichnern und anderen Ausdrücken zusammengesetzt.

Als *Teilausdruck erster Ebene* eines Ausdrucks $A = [f, a, b, c, \ldots]$ bezeichnen wir die Ausdrücke $f, a, b, c, \ldots$, als *Teilausdrücke* diese Ausdrücke sowie deren Teilausdrücke. Ein Bezeichner *kommt in einem Ausdruck vor*, wenn er ein Teilausdruck dieses Ausdrucks ist.

Sind $x_1, \ldots, x_n$ Bezeichner, $A, U_1, \ldots, U_n$ Ausdrücke und die Bezeichner $x_i, i = 1, \ldots, n$, kommen in keinem der $U_j, j = 1, \ldots, n$, vor, so schreiben wir $A(x \vdash U)$ für den Ausdruck, der entsteht, wenn alle Vorkommen von x_i in A durch U_i ersetzt werden.

$\mathcal{E}$ sei die Menge der wohlgeformten Ausdrücke, also der Zeichenfolgen, welche ein CAS „versteht“. Die semantische Gleichwertigkeit solcher Ausdrücke kann als eine (nicht notwendig effektiv berechenbare) Äquivalenzrelation $\sim$ auf $\mathcal{E}$ verstanden werden. Diese Relation wird stets durch die mathematische Theorie vorgegeben, in der die Ausdrücke interpretiert werden. Wir hatten bereits gesehen, dass die semantische Gleichwertigkeit von Ausdrücken von dieser mathematischen Theorie abhängen kann. So gelten eine Reihe von Beziehungen zwischen Funktionen in der Theorie der stetigen reellwertigen Funktionen, sind aber in der Theorie der meromorphen Funktionen nicht mehr gültig, etwa die Vereinfachung $\sqrt{x^2} = |x|$.

Wir wollen im Weiteren die *Kontextfreiheit* dieser semantischen Relation $\sim$ voraussetzen, dass also das Ersetzen eines Teilausdrucks durch einen semantisch äquivalenten Teilausdruck in einem Gesamtausdruck zu einem semantisch äquivalenten Gesamtausdruck führt.

Genauer fordern wir für alle x-freien Ausdrücke U, V mit $U \sim V$ und alle Ausdrücke $A(x), B(x)$ mit $A(x) \sim B(x)$, dass $A(x \vdash U) \sim B(x \vdash V)$ gilt.

Als *Simplifikator* bezeichnen wir eine effektiv berechenbare Funktion $S : \mathcal{E} \longrightarrow \mathcal{E}$, welche 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.

Wir hatten gesehen, dass in einem regelbasierten Transformationskonzept ein solcher Simplifikator als fortgesetzte Anwendung eines Arsenals von Transformationsregeln ausgeführt ist, wobei die Regeln immer wieder auf den entstehenden Zwischenausdruck angewendet werden, bis keine Ersetzungen mehr möglich sind. Sei dazu $\mathcal{R}$ ein (endliches) Regelsystem, also eine Menge von Regeln der Gestalt $\texttt{Rule}(L, R, B)(u)$.

Dabei bezeichnet $u = (u_1, \ldots, u_n)$ eine Liste von formalen Parametern und $L, R, B \in \mathcal{E}$ Ausdrücke, so dass

> für alle *zulässigen* Belegungen $u \vdash U$ der formalen Parameter mit u-freien Ausdrücken $U = (U_1, \ldots, U_n)$, d. h. solchen mit $\texttt{bool}(B(u \vdash U)) = \texttt{true}$, die entsprechenden Ausdrücke semantisch äquivalent sind, d. h. $L(u \vdash U) \sim R(u \vdash U)$ in $\mathcal{E}$ gilt.

$\texttt{bool} : \mathcal{E} \to \{\texttt{true}, \texttt{false}, \texttt{fail}\}$ ist dabei eine boolesche Auswertefunktion auf der Menge der Ausdrücke. Die letzte Bedingung ist wegen der Kontextfreiheit von $\sim$ insbesondere dann erfüllt, wenn bereits $L(u) \sim R(u)$ in $\mathcal{E}$ gilt. Eine konditionale Restriktion B hat in diesem Fall keine Auswirkungen auf die semantische Korrektheit. Allerdings kann eine konditionale Restriktion für die Termination des Regelsystems erforderlich sein.

Die Regel $r = \texttt{Rule}(L, R, B)(u) \in \mathcal{R}$ ist auf einen Ausdruck A *anwendbar*,

(1) wenn es eine zu u disjunkte Liste von Bezeichnern $x = (x_0, x_1, \ldots, x_n)$ gibt, so dass A x-frei ist. Zur Vermeidung von Namenskollisionen arbeiten wir mit den Regeln $L' = L(u \vdash x)$ und $R' = R(u \vdash x)$ **(gebundene Umbenennung)**

(2) wenn es weiter einen Ausdruck A' und Teilausdrücke $U = (U_1, \ldots, U_n)$ von A mit $A = A'(x \vdash U)$ gibt, so dass L' ein Teilausdruck von A' ist, d. h. $A' = A''(x_0 \vdash L')$ für einen weiteren Ausdruck A'' gilt **(Matching)**

(3) und $\texttt{bool}(B(u \vdash U)) = \texttt{true}$ gilt **(Konditionierung)**.

Dann wird also $A = A''(x_0 \vdash L'(x \vdash U)) = A''(x_0 \vdash L(u \vdash U))$ im Ergebnis der Anwendung der Regel r durch $A^{(1)} = A''(x_0 \vdash R(u \vdash U))$ ersetzt. Wir schreiben auch $A \to_r A^{(1)}$.

Weniger formal gesprochen bedeutet die Anwendung einer Regel also, in einem zu untersuchenden Ausdruck A

- einen Teilausdruck $L'' = L(u \vdash U)$ der in der Regel spezifizierten Form zu finden,
- die formalen Parameter in L'' zu „matchen“,
- aus den Teilen den Ausdruck $R'' = R(u \vdash U)$ zusammenzubauen und
- schließlich L'' durch R'' zu ersetzen.

Der zugehörige Simplifikator $S = S(\mathcal{R}) : \mathcal{E} \longrightarrow \mathcal{E}$ ist der transitive Abschluss der durch die Regelmenge $\mathcal{R}$ definierten Ersetzungsrelationen[3].

[3] Das ist nicht ganz korrekt, da das Ergebnis der fortgesetzten Regelanwendung auch im (hier vorausgesetzten) Terminationsfall von der Wahl der möglichen Matchings und passenden Regeln abhängt. Wir setzen deshalb stillschweigend eine feste Auswahlstrategie als gegeben voraus.

Termination

S ist somit *effektiv*, wenn die Implementierung von $\mathcal{R}$ die folgenden beiden Bedingungen erfüllt:

(Matching) Es lässt sich effektiv entscheiden, ob es zu einem gegebenem Ausdruck A und einer Regel $r \in \mathcal{R}$ ein Matching gibt.

(Termination) Nach endlich vielen Schritten $A \to_{r_1} A^{(1)} \to_{r_2} A^{(2)} \to_{r_3} \ldots \to_{r_N} A^{(N)}$ mit $r_1, \ldots, r_N \in \mathcal{R}$ ist keine Ersetzung mehr möglich.

Die erste Bedingung lässt sich offensichtlich durch entsprechendes Absuchen des Ausdrucks A unabhängig vom gegebenen Regelsystem erfüllen. Die einzige Schwierigkeit besteht darin, verschiedene Vorkommen desselben formalen Parameters in der linken Seite von r korrekt zu matchen. Dazu muss festgestellt werden, ob zwei Teilausdrücke U' und U'' von A syntaktisch übereinstimmen. Dies kann jedoch leicht durch eine `equal`-Funktion realisiert werden, die rekursiv die Teilausdrücke erster Ebene von U' und U'' vergleicht und bei atomaren Ausdrücken von der eindeutigen Darstellung in der Symboltabelle Gebrauch macht. Letzterer Vergleich wird auch als `eq`-Vergleich bezeichnet, da hier nur zwei Referenzen verglichen werden müssen. Aus Effizienzgründen wird man solche `eq`-Vergleiche auch für entsprechende Teilausdrücke von U' und U''durchführen, denn wenn es Referenzen auf denselben Ausdruck sind, kann der weitere Vergleich gespart werden.

Die zweite Bedingung dagegen hängt wesentlich vom Regelsystem $\mathcal{R}$ ab. Am einfachsten lässt sich die Termination sichern, wenn es eine (teilweise) Ordnungsrelation $\leq$ auf $\mathcal{E}$ gibt, bzgl. derer die rechten Seiten der Regeln „kleiner“ als die linken Seiten sind. In diesem Sinne ist dann auch $S(t)$ „einfacher“ als t, d. h. es gilt

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

Die Termination ist gewährleistet, wenn zusätzlich gilt:

(1) $\leq$ ist wohlfundiert.

(2) **Simplifikation**: Für jede Regel $r = \texttt{Rule}(L, R, B)(u) \in \mathcal{R}$ und jede zulässige Belegung $u \vdash U$ gilt $L(u \vdash U) > R(u \vdash U)$.

(3) **Monotonie**: Sind U_1, U_2 zwei x-freie Ausdrücke mit $U_1 > U_2$ und A ein weiterer Ausdruck, in welchem x vorkommt, so gilt $A(x \vdash U_1) > A(x \vdash U_2)$.

Die zweite und dritte Eigenschaft sichern, dass in einem elementaren Simplifikationsschritt die Vereinfachung zu einem „kleineren“ Ausdruck führt, die erste, dass eine solche Simplifikationskette nur endlich viele Schritte haben kann. In der Tat, ist $A \to A^{(1)}$ durch

$$A''(x_0 \vdash L(u \vdash U)) \to A''(x_0 \vdash R(u \vdash U))$$

beschrieben, so sichert (2), dass $U_1 = L(u \vdash U) > U_2 = R(u \vdash U)$ gilt, und (3) schließlich $A = A''(x_0 \vdash U_1) > A''(x_0 \vdash U_2) = A^{(1)}$.

Es reicht aus, dass es sich bei $>$ um eine teilweise Ordnung mit den Eigenschaften (1) – (3) handelt. Eine solche partielle Ordnung wird z. B. durch

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

für ein geeignetes *Kompliziertheitsmaß* $l : \mathcal{E} \longrightarrow \mathbb{N}$ beschrieben. In diesem Fall sind nur die Eigenschaften (2) und (3) zu prüfen. l kann z. B. die verwendeten Zeichen, die Klammern, die Additionen, den `LeafCount` usw. zählen. Obwohl Kompliziertheitsmaße auf $\mathcal{E}$ nur für sehr einfache Regelsysteme explizit angegeben werden können, spielen sie eine zentrale Rolle beim Beweis der Termination von Regelsystemen.

Allgemein ist die Termination von Regelsystemen schwer zu verifizieren und führt schnell zu (beweisbar) algorithmisch nicht lösbaren Problemen.

Simplifikation und Ergebnisqualität

Ein Simplifikationsprozess kann wesentlich von den gewählten Simplifikationsschritten abhängen, wenn für einen (Zwischen-)Ausdruck mehrere Simplifikationsmöglichkeiten und damit Simplifikationspfade existieren. Dies kann nicht nur Einfluss auf die Rechenzeit, sondern auch auf das Ergebnis selbst haben. Sinnvolle Aussagen dazu sind nur innerhalb eingeschränkterer Klassen von Ausdrücken möglich. Sei dazu $\mathcal{U} \subset \mathcal{E}$ eine solche Klasse von Ausdrücken.

Als *kanonische Form* innerhalb $\mathcal{U}$ bezeichnet man einen Simplifikator $S : \mathcal{U} \longrightarrow \mathcal{U}$, der zusätzlich der Bedingung

$$s \sim t \Rightarrow S(s) = S(t) \qquad \text{für alle } s, t \in \mathcal{U} \tag{C}$$

genügt. Für einen solchen Simplifikator führen wegen (E) alle möglichen Simplifikationspfade zum selben Ergebnis. $S(t)$ bezeichnet man deshalb auch als ***die*** *kanonische Form* des Ausdrucks t innerhalb der Klasse $\mathcal{U}$. Ein solcher Simplifikator hat eine in Bezug auf unsere Ausgangsfrage starke Eigenschaft:

> Ein kanonischer Simplifikator erlaubt es, semantisch äquivalente Ausdrücke innerhalb einer Klasse $\mathcal{U}$ an Hand des (syntaktischen) Aussehens der entsprechenden kanonischen Form eindeutig zu identifizieren.

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 aus der Klasse $\mathcal{U}$ sind genau dann äquivalent, wenn ihre kanonischen Formen literal (Zeichen für Zeichen) übereinstimmen.

Solche Simplifikatoren existieren nur für spezielle Klassen von Ausdrücken $\mathcal{E}$. Deshalb ist auch die folgende Abschwächung dieses Begriffs von Interesse. Nehmen wir an, dass $\mathcal{U}/\sim$, wie in den meisten Anwendungen in der Computeralgebra, die Struktur einer additiven Gruppe trägt, d. h. ein spezielles Symbol $0 \in \mathcal{U}$ für das Nullelement sowie eine Funktion $M : \mathcal{U} \times \mathcal{U} \longrightarrow \mathcal{U}$ existiert, welche die Differenz zweier Ausdrücke (effektiv syntaktisch) aufzuschreiben vermag. Letzteres bedeutet insbesondere, dass

$$s \sim t \Leftrightarrow M(s, t) \sim 0$$

gilt. Als *Normalformoperator* in der Klasse $\mathcal{U}$ bezeichnen wir dann einen Simplifikator $S : \mathcal{U} \longrightarrow \mathcal{U}$, für den zusätzlich

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

gilt, d. h. jeder Nullausdruck $t \in \mathcal{U}$ wird durch S auch als solcher erkannt[4].

Diese Forderung ist schwächer als die der Existenz einer kanonischen Form: Es kann sein, dass für zwei Ausdrücke $s, t \in \mathcal{U}$, 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 innerhalb der Klasse $\mathcal{U}$ lösen, denn zwei **vorgegebene** Ausdrücke s und t sind offensichtlich genau dann semantisch äquivalent, wenn ihre Differenz zu null vereinfacht werden kann:
>
> $$s \sim t \Leftrightarrow S(M(s, t)) = 0.$$

Kern des Arguments ist die Existenz einer booleschen Funktion $\texttt{iszero} : \mathcal{U} \to \texttt{boolean}$ mit der Eigenschaft

$$t \sim 0 \Leftrightarrow \texttt{iszero}(t) = \texttt{true} \qquad \text{für alle } t \in \mathcal{U}$$

Eine solche Funktion, die nicht unbedingt über einen Simplifikator definiert sein muss, bezeichnet man auch als *starken Nulltester*.

[4] Für $t = 0$ ergibt sich insbesondere $S(0) = 0$.

3.6 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.

Polynome in distributiver Darstellung

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 $a = (a_1, \ldots, a_n) \in \mathbb{N}^n$ ein Multiindex ist und X^a kurz für $x_1^{a_1} \cdots x_n^{a_n}$ steht. Eine solche Darstellung kann in den meisten CAS durch die Funktion `expand` erzeugt werden. REDUCE verwendet sie standardmäßig für die Darstellung von Polynomen.

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 $T(X) = \{X^a : a \in \mathbb{N}^n\}$, 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 welcher 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 darüber hinaus *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 2 *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.*
Hat R einen starken Nulltester, so auch 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.

Polynome in rekursiver Darstellung

Als *rekursive Darstellung* bezeichnet man die Darstellung von Polynomen aus S als Polynome in x_n mit Koeffizienten aus $R[x_1, \ldots, x_{n-1}]$, wobei die Summanden nach fallenden Potenzen von x_n angeordnet und die Koeffizienten rekursiv nach demselben Prinzip dargestellt sind. Da die rekursive Darstellung für $n = 1$ mit der distributiven Darstellung zusammenfällt, führt die rekursive Natur des bewiesenen Satzes unmittelbar zu folgendem

Folgerung 1 *Existiert auf R eine kanonische Form, dann ist auch die rekursive Darstellung von Polynomen $f \in R[x]$ mit Koeffizienten in kanonischer Form eine kanonische Form auf $R[x]$.*
Hat R einen starken Nulltester, so auch $R[x]$.

Die 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

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

Im Gegensatz dazu führt die **faktorisierte Darstellung von Polynomen** nicht zu einer kanonischen Form, da eine Faktorzerlegung in $R[x_1, \ldots, x_n]$ immer nur eindeutig bis auf Einheiten aus R bestimmt werden kann. Auch für die Polynomoperationen ist diese Darstellung eher ungeeignet.

Rationale Funktionen

Wir hatten bereits mehrfach gesehen, dass CAS hervorragend in der Lage sind, auch rationale Funktionen zu vereinfachen. Allerdings ist es in einigen Beispielen besser, die spezielle Simplifikationsstrategie `normal` zu verwenden als den allgemeinen Ansatz `simplify`, wie folgendes Beispiel (MuPAD) demonstriert:

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

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

```
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);
```

$$a + b + c$$

`normal` verfährt nach folgendem einfachen Schema: Es bestimmt einen gemeinsamen Hauptnenner und überführt dann das entstehende Zählerpolynom in seine distributive Form. Auf diese Weise entsteht 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. Es gilt damit folgender

Satz 3 *Existiert auf R eine Normalform, dann liefert die beschriebene Normalisierungsstrategie eine Normalform auf dem Ring $R(x_1, \ldots, x_n)$ der rationalen Funktionen über R.*

Hat R also einen starken Nulltester, so auch der Ring $R(x_1, \ldots, x_n)$ der rationalen Funktionen über R.

Eine solche Darstellung bezeichnet man deshalb auch als **rationale Normalform**. Von ihr 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 [2]. Da die Berechnung des gcd aber im Vergleich zu den anderen arithmetischen Operationen aufwändiger 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;
```

$$\frac{x^4 + x^3 + x^2 + x + 1}{x^2 + x + 1}$$

Die meisten CAS wandeln Eingaben nicht automatisch in eine rationale Normalform um, sondern nur auf spezielle Anforderung hin (wobei dann auch der gcd von Zähler und Nenner ausgeteilt

wird) oder im Zuge von Funktionsaufrufen wie `factor`. In REDUCE (immer) oder MAXIMA können Teile des Ergebnisses in dieser „compilierten“ Form vorliegen, was bei der Suche nach Mustern im Rahmen des regelbasierten Transformationszugangs zu berücksichtigen ist. MAXIMA zeigt durch eine Marke `/R/` an, wenn ein Ausdruck oder Teile davon in dieser rationalen Normalform vorliegen.

Rationale Normalformen sind auf der Basis der distributiven Darstellung von Polynomen (MAPLE, MUPAD, MATHEMATICA) oder aber der rekursiven Darstellung (REDUCE, MAXIMA) möglich.

MAXIMA	`ratsimp(f)`
MAPLE	`normal(f)`
MATHEMATICA	`Together[f]`
MUPAD	`simplifyFraction(f)`
REDUCE	standardmäßig

Tabelle 3: Rationale Normalform bilden

Verallgemeinerte Kerne

Auf Grund der guten Eigenschaften, welche die beschriebenen Simplifikationsverfahren polynomialer und rationaler Ausdrücke besitzen, werden sie auch auf Ausdrücke angewendet, die nur teilweise eine solche Struktur besitzen.

Betrachten wir etwa die Vereinfachung, die REDUCE automatisch an einem trigonometrischen Ausdruck vornimmt, und vergleichen ihn mit einem analogen polynomialen Ausdruck.
Die innere Struktur beider Ausdrücke ist ähnlich, einzig statt der Variablen a und b stehen verallgemeinerte Variablen (sin x) und (cos x) (die Lisp-Notation von sin(x) und cos(x)).

```
u1:=(sin(x)+cos(x))^3;
```

$$\cos(x)^3 + 3\cos(x)^2\sin(x) + 3\cos(x)\sin(x)^2 + \sin(x)^3$$

```
u2:=(a+b)^3;
```

$$a^3 + 3a^2b + 3ab^2 + b^3$$

```
lisp prettyprint prop 'u2;
((avalue scalar
  (!*sq
    ((((a . 3) . 1)
      ((a . 2) ((b . 1) . 3))
      ((a . 1) ((b . 2) . 3))
      ((b . 3) . 1))
     . 1)
    t)))
```

```
lisp prettyprint prop 'u1;
((avalue scalar
  (!*sq
    (((((cos x) . 3) . 1)
      (((cos x) . 2) (((sin x) . 1) . 3))
      (((cos x) . 1) (((sin x) . 2) . 3))
      (((sin x) . 3) . 1))
     . 1)
    t)))
```

Ein ähnliches Ergebnis liefert die Funktion `expand` bei „konservativeren“ Systemen wie z. B. MUPAD. Die interne Struktur als rationaler Ausdruck kann hier mit `rationalize` bestimmt werden.

```
u:=expand((sin(x)+cos(x))^3);
rationalize(u);
```

$$D3^3 + D4^3 + 3\,D3\,D4^2 + 3\,D3^2\,D4, \quad \{D3 = \cos(x), D4 = \sin(x)\}$$

In beiden Fällen entstehen polynomiale Ausdrücke, aber nicht in (freien) Variablen, sondern in allgemeineren Ausdrücken, wie in diesem Fall `sin(x)` und `cos(x)`, welche für die vorzunehmen-

de Normalform-Expansion als algebraisch unabhängig angenommen werden. Die zwischen ihnen bestehende algebraische Relation $\sin(x)^2 + \cos(x)^2 = 1$ muss ggf. durch andere Simplifikationsmechanismen zur Wirkung gebracht werden. Solche allgemeineren Ausdrücke werden als *Kerne* bezeichnet.

In Maxima kann man statt mit `ratsimp` Ausdrücke mit solchen Kernen auch mit der Funktion `rat` in deren rationale Normalform umwandeln. Dabei merkt sich das System, dass es sich um eine rationale Normalform handelt. Die Marke /R/ in der Ausgabe weist darauf hin.

```
u:(sin(x)+cos(x)^3)$
showratvars(u);
```

$$[\cos(x),\ \sin(x)]$$

```
rat(u);
```

$$/R/ \qquad \cos(x)^3 + 3\,\cos(x)^2\,\sin(x) + 3\,\cos(x)\,\sin(x)^2 + \sin(x)^3$$

CAS stellen allgemeine Ausdrücke dar als rationale Ausdrücke in verallgemeinerten Variablen, die als *Kerne* bezeichnet werden. Ein solcher Kern kann dabei ein Symbol oder ein Ausdruck mit einem nicht-arithmetischen Funktionssymbol als Kopf sein.

Maxima	`showratvars(f)`
Maple	`indets(f)`
Mathematica	`Variables[f]`
MuPAD	`indets(f,RatExpr)`
Reduce	`prop 'f;`

Tabelle 4: Kerne eines Ausdrucks bestimmen

Hier noch ein etwas komplizierteres Beispiel, das von den verschiedenen CAS auf unterschiedliche Weise „compiliert“ wird.

```
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));
```

$$\frac{\cos(x)\,\sin(x)^4 + 2\,\cos(x)^3\,\sin(x)^2 + \cos(x)^5 + \mathrm{e}^{x}\,\cos(x)}{x^2 - x^2\,\mathrm{e}^{-2\,x}} - \frac{\cos(x)\,\sin(x)^2 + \cos(x)^3 + \mathrm{e}^{-x}\,\cos(x)}{x^2\,\mathrm{e}^{x} - x^2\,\mathrm{e}^{-x}}$$

Maple und MuPAD interpretieren den Ausdruck ähnlich

```
indets(u); // Maple
```

$$\{x,\ \cos(x),\ \sin(x),\ \exp(x),\ \exp(-x),\ \exp(-2\,x)\}$$

```
rationalize(u); // MuPAD
```

$$\frac{D8\,D9 + D9^5 + D9\,D10^4 + 2\,D9^3\,D10^2}{x^2 - x^2\,D7} - \frac{D6\,D9 + D9^3 + D9\,D10^2}{x^2\,D8 - x^2\,D6}$$

$$\{D9 = \cos(x),\ D8 = \exp(x),\ D10 = \sin(x),\ D6 = \exp(-x),\ D7 = \exp(-2\,x)\}$$

Die Wirkung von `normal` folgt der beobachteten Zerlegung. Insbesondere werden die Terme $\exp(x)$, $\exp(-x)$ und $\exp(-2\,x)$ als eigenständige Kerne behandelt. MuPAD fasst $\exp(m\,x)\cdot\exp(n\,x)$ für $m, n \in \mathbb{Z}$ zu $\exp((m+n)\,x)$ automatisch zusammen.

```
v=simplifyFraction(u);
```

$$\frac{\left(\begin{array}{c} \mathrm{e}^{3\,x}\cos(x)-\cos(x)-\mathrm{e}^{x}\cos(x)^3+\mathrm{e}^{2\,x}\cos(x)^5+ \\ 2\,\mathrm{e}^{2\,x}\cos(x)^3\sin(x)^2-\mathrm{e}^{x}\cos(x)\sin(x)^2+\mathrm{e}^{2\,x}\cos(x)\sin(x)^4 \end{array}\right)}{x^2\left(\mathrm{e}^{2\,x}-1\right)}$$

Ersetzt man in zusätzlich $\sin(x)^2$ durch $1-\cos(x)^2$, so ergibt sich eine besonders einfache Form des Ausdrucks.

```
simplify(v);
```

$$\frac{e^{x}\cos(x)+\cos(x)}{x^2}$$

Maple belässt bei der Berechnung von `normal(u)` den Nenner in faktorisierter Form

$$x^2\left(\mathrm{e}^{-2x}-1\right)\left(\mathrm{e}^{x}-\mathrm{e}^{-x}\right),$$

was die Normalformeigenschaft nicht zerstört, aber die Ähnlichkeit der beiden Faktoren $\mathrm{e}^{-2x}-1$ und $\mathrm{e}^{x}-\mathrm{e}^{-x}$ nicht erkennt und damit zu einem komplizierteren Ausdruck führt als MuPAD.

`v1:=normal(u,expanded)` expandiert zusätzlich den Nenner, fasst aber auch die verschiedenen exp-Kerne zusammen, was die Zahl der Kerne reduziert und auf dasselbe Ergebnis wie MuPAD führt.

Maxima, Reduce und Mathematica wenden sofort die Regel $\exp(n\,x)=\exp(x)^n$ für $n\in\mathbb{Z}$ an und reduzieren auf diese Weise die Zahl der Kerne.

```
showratvars(u1);
```

$$[x,\ \mathrm{e}^{x},\ \cos(x),\ \sin(x)]$$

Hier das Ergebnis der Berechnung der rationalen Normalform mit Maxima. An der Darstellung ist zu erkennen, dass intern eine rekursive Polynomdarstellung verwendet wird.

```
ratsimp(u1);
```

$$\frac{\left(\begin{array}{c} e^{2x}\,\cos(x)\,\sin^4(x)+\left(2\,e^{2x}\,\cos^3(x)-e^{x}\,\cos(x)\right)\sin^2(x)+ \\ e^{2x}\,\cos^5(x)-e^{x}\,\cos^3(x)+\left(e^{3x}-1\right)\cos(x) \end{array}\right)}{x^2\,e^{2x}-x^2}$$

Mathematica behauptet zwar, dass $\exp(x)$ nicht mit zu den Kernen dieses Ausdrucks gehört, aber die Wirkung von `Together` zeigt, dass das nicht stimmt.

```
Variables[u]
```

```
{x, Cos[x], Sin[x]}
```

```
Together[u]
```

$$\frac{\cos(x)\left(-1+\mathrm{e}^{3x}-\mathrm{e}^{x}\cos(x)^2+\mathrm{e}^{2x}\cos(x)^4-\mathrm{e}^{x}\sin(x)^2+2\,\mathrm{e}^{2x}\cos(x)^2\sin(x)^2+\mathrm{e}^{2x}\sin(x)^4\right)}{\left(-1+\mathrm{e}^{2x}\right)x^2}$$

Eine eigenständige, im Systemkern verankerte Simplifikationsschicht für polynomiale und rationale Ausdrücke in solchen Kernen spielt eine wichtige Rolle im Simplifikationsdesign aller CAS. Sowohl die Herstellung rationaler Normalformen und anschließende Anwendung weitergehender Vereinfachungen wie in obigem Beispiel als auch die gezielte Umformung anderer funktionaler Abhängigkeiten in rationale wie etwa beim Expandieren trigonometrischer Ausdrücke werden angewendet.

3.7 Trigonometrische Ausdrücke und Regelsysteme

In diesem Abschnitt werden wir uns mit dem Aufstellen von Regelsystemen für Simplifikationszwecke näher vertraut machen. Als Anwendungsbereich werden wir dabei **trigonometrische Ausdrücke** betrachten, worunter wir arithmetische Ausdrücke verstehen, deren Kerne trigonometrische Funktionssymbole enthalten.

Genauer werden wir zunächst die Klasse *TrigPoly* der polynomialen Ausdrücke mit rationalen Koeffizienten betrachten, in denen Kerne der Form $\sin(A)$ und $\cos(A)$ vorkommen, wobei A für ganzzahlige Linearkombinationen verschiedener Variablen und Konstanten, also etwa $A = x - 2\,y + \frac{\pi}{4}$ steht, und diese Ausdrücke in der Theorie der holomorphen Funktionen interpretieren.

Die verschiedenen Additionstheoreme zeigen, dass zwischen derartigen Kernen eine große Zahl von Abhängigkeiten besteht. Diese Ausdrücke bilden damit eine Klasse, in der besonders viele syntaktisch verschiedene, aber semantisch äquivalente Darstellungen möglich und für die verschiedensten Zwecke auch nützlich sind.

Schauen wir uns zunächst an, wie die verschiedenen CAS selbst solche Umformungen einsetzen.

Dazu betrachten wir drei einfache trigonometrische Ausdrücke, 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.

$$f_1 := \sin(3x)\,\sin(5x)$$

$$f_2 := \sin\left(2x - \frac{\pi}{6}\right)\,\cos\left(3x + \frac{\pi}{4}\right)$$

$$f_3 := \cos\left(\frac{x}{2}\right)\,\cos\left(\frac{x}{3}\right)$$

f_3 enthält die zusätzliche Schwierigkeit zu erkennen, dass der Ausdruck nach einer einfachen Umformung in einen Ausdruck mit dem Kern $y = \frac{x}{6}$ in die Klasse *TrigPoly* fällt. Eine solche Umformung kann vom Nutzer vorab angestoßen werden, wenn sie vom CAS nicht von sich aus erkannt wird.

Neben unserer eigentlichen Problematik erlaubt die Art der Antwort der einzelnen Systeme zugleich einen gewissen Einblick, wie diese die entsprechenden Integrationsaufgaben lösen. An der typische Antwort von Maple können wir das Vorgehen gut erkennen:

Zur Berechnung des Integrals wurde das Produkt von Winkelfunktionen durch Anwenden der entsprechenden Additionstheoreme in eine Summe einzelner Winkelfunktionen mit Vielfachen von x als Argument umgewandelt.

$$g_1 := \int f_1\,dx = \frac{\sin(2x)}{4} - \frac{\sin(8x)}{16}$$

$$g_1' := \frac{\cos(2x)}{2} - \frac{\cos(8x)}{2}$$

Eine solche Linearkombination von Kernen kann man leicht integrieren. Wollen wir aber prüfen, dass in der Tat $f_1 = g_1'$ gilt, so sind die vielen verschiedenen Kerne eher hinderlich, denn wir müssen diese Winkelfunktionen mit verschiedenen Argumenten vergleichen. Dazu ist es sinnvoll, die Anzahl der Kerne zu reduzieren, um die polynomialen Normalformeigenschaften gut auszunutzen. Um die Vielfachen von x als Argument wieder aufzulösen, sind die entsprechenden Regeln in der entgegengesetzten Richtung anzuwenden.

Welcher Art von Simplifikationsregeln werden in beiden Fällen verwendet? Für die erste Aufgabe sind Produkte in Summen zu verwandeln. Das erreicht man durch (mehrfaches) Anwenden der Regeln *trigsum*.

$$\begin{aligned}\sin(x)\sin(y) &\Rightarrow 1/2\,(\cos(x-y)-\cos(x+y))\\ \cos(x)\cos(y) &\Rightarrow 1/2\,(\cos(x-y)+\cos(x+y))\\ \sin(x)\cos(y) &\Rightarrow 1/2\,(\sin(x-y)+\sin(x+y))\end{aligned}$$

Diese Regeln sind invers zu den Regeln *trigexpand*, die man beim Expandieren von Winkelfunktionen, deren Argumente Summen oder Differenzen sind, anwendet.

$$\begin{aligned}\sin(x+y) &\Rightarrow \sin(x)\cos(y)+\cos(x)\sin(y)\\ \cos(x+y) &\Rightarrow \cos(x)\cos(y)-\sin(x)\sin(y)\end{aligned}$$

Bei der Formulierung der entsprechenden Regeln für $x-y$ spielt die interne Darstellung von solchen Differenzen eine Rolle.

Wird sie als Summe $x+(-y)$ dargestellt, so müssen wir keine weiteren Simplifikationsregeln für eine *binäre* Operation MINUS, wohl aber für die *unäre* Operation MINUS angeben.

$$\begin{aligned}\sin(-x) &\Rightarrow -\sin(x)\\ \cos(-x) &\Rightarrow \cos(x)\end{aligned}$$

Normalerweise wird diese Simplifikation aber automatisch ausgeführt, weil das CAS aus anderen Quellen weiß, dass sin eine ungerade und cos eine gerade Funktion ist.

Das Regelsystem trigsum

In all diesen Regeln sind x und y als formale Parameter zu betrachten, so dass die obigen Regeln in REDUCE-Notation wie folgt anzuschreiben sind:

```
trigsum0:={ % 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))};
```

Regeln werden in REDUCE als `lhs => rhs where bool` notiert. Die Tilde vor einer Variablen bedeutet, dass sie als formaler Parameter verwendet wird. 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[5] und genau wie die Vereinfachung rationaler Funktionen gesondert und automatisch behandelt wird.

Wenden wir die Regeln `trigsum0` auf einen polynomialen Ausdruck in den Kernen `sin(x)` und `cos(x)` an, so sollten am Ende alle Produkte trigonometrischer Funktionen zugunsten von Mehrfachwinkelargumenten aufgelöst sein, womit der Ausdruck in eine besonders einfach zu integrierende Form überführt wird. Die Termination dieser Simplifikation ergibt sich daraus, dass bei jeder Regelanwendung in jedem Ergebnisterm die Zahl der Faktoren um eins geringer ist als im Ausgangsterm.

Leider klappt das nicht so, wie erwartet, wie etwa dieses Beispiel zeigt. Hier ist der Ausdruck $\sin(x)^2$ nicht weiter vereinfacht worden, weil er nicht die Gestalt `(* (sin A) (sin B))`, sondern `(expt (sin A) 2)` hat.

```
sin(x)*sin(2x)*cos(3x) where trigsum0;
```

$$\frac{-\cos(6\,x)+\cos(4\,x)-2\sin(x)^2}{4}$$

[5] wie man mit `flagp('sin,'odd)` und `flagp('cos,'even)` erkennt

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 $\sin(x)^2 = \dfrac{1-\cos(2x)}{2}$ (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 Rekursionstiefe 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. Eine entsprechende Erweiterung der Regeln `trigsum` sähe dann so aus:

```
trigsum1:={
        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 exakten Ersetzungsregeln

beachten. Betrachten wir dazu die distributive Normalform von $(a+1)^5$, also den Ausdruck

$$A = a^5 + 5\,a^4 + 10\,a^3 + 10\,a^2 + 5\,a + 1,$$

und ersetzen in ihm a^2 durch x.

MATHEMATICA liefert ein anderes Ergebnis

```
A /. (a^2 -> x)
```

$$1 + 5\,a + 10\,a^3 + 5\,a^4 + a^5 + 10\,x$$

als REDUCE

```
(a+1)^5 where (a^2 => x);
```

$$a\,x^2 + 10\,a\,x + 5\,a + 5\,x^2 + 10\,x + 1$$

Im ersten Fall wurden nur solche Muster ersetzt hat, die *exakt* auf den Ausdruck a^2 passen (literales Matching), während im zweiten Fall alle Ausdrücke $a^k, k > 2$, welche den Faktor a^2 enthalten, ersetzt worden sind (algebraisches Matching).

Algebraisches Matching kann in MATHEMATICA durch mehrfaches Anwenden dieser Regel r erreicht werden. Dies ist zugleich das allgemeine Vorgehen, wie sich algebraische Regeln in einem System mit exaktem Matching anschreiben lassen.

```
r = a^(n_Integer)/;(n>1) -> a^(n-2)*x;

A //. r
```

$$1 + 5\,a + 10\,x + 10\,a\,x + 5\,x^2 + a\,x^2$$

Unser gesamtes Regelsystem `trigsum` für REDUCE lautet damit:

```
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(2*x))/2,
        sin(~x)^2 => (1-cos(2*x))/2 };
```

Die Anwendung dieses Regelsystems `trigsum` führt auf eine kanonische Darstellung innerhalb der Klasse *TrigPoly*, ist also für Simplifikationszwecke hervorragend geeignet. Genauer gilt folgender

Satz 4 *Jeder polynomiale Ausdruck $P(\sin(x), \cos(x))$ mit rationalen Koeffizienten kann mit obigem Regelsystem `trigsum` in einen Ausdruck der Form*

$$\sum_{k>0} (a_k \sin(kx) + b_k \cos(kx)) + c \tag{TS}$$

mit $a_k, b_k, c \in \mathbb{Q}$ verwandelt werden.

Diese Darstellung ist eindeutig. Genauer: Werden die rationalen Koeffizienten in ihrer kanonischen Form als gekürzte Brüche dargestellt, so ist diese Darstellung (TS) eine kanonische Form in der Klasse der betrachteten Ausdrücke.

Beweis: Da das Regelsystem alle Produkte und Potenzen von trigonometrischen Kernen ersetzt und sich in jedem Transformationsschritt die Anzahl der Multiplikationen verringert[6], 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 in der Theorie der stetigen reellwertigen Funktionen als Fourierkoeffizienten gewinnen. Dazu berechnen wir für ein festes ganzzahliges $n > 0$ das Integral

$$\begin{aligned}
2\int_0^{2\pi} f(x)\sin(nx)\,dx &= \sum_{k>0} a_k \int_0^{2\pi} 2\sin(kx)\sin(nx)\,dx \\
&\quad + \sum_{k>0} b_k \int_0^{2\pi} 2\cos(kx)\sin(nx)\,dx + 2c\int_0^{2\pi} \sin(nx)\,dx \\
&= \sum_{k>0} a_k \int_0^{2\pi} (\cos((k-n)x) - \cos((k+n)x))\,dx \\
&\quad + \sum_{k>0} b_k \int_0^{2\pi} (\sin((n-k)x) + \sin((n+k)x))\,dx \\
&= 2\pi\, a_n
\end{aligned}$$

und sehen, dass $f(x)$ jeden Koeffizienten a_n eindeutig bestimmt. Wir haben dabei von unseren Formeln Produkt-Summe sowie den Beziehungen

$$\begin{aligned}
\int_0^{2\pi} \sin(m\,x)\,dx &= 0 \\
\int_0^{2\pi} \cos(m\,x)\,dx &= \begin{cases} 0 & \text{für } m \neq 0 \\ 2\,\pi & \text{für } m = 0 \end{cases}
\end{aligned}$$

Gebrauch gemacht. Analog ergeben sich die Koeffizienten b_n und c aus $\int_0^{2\pi} f(x)\cos(nx)\,dx$ bzw. $\int_0^{2\pi} f(x)\,dx$. $\square$

Die einfache Struktur des Simplifikationssystems verwendet implizit die Tatsache, dass REDUCE standardmäßig polynomiale Ausdrücke in ihre distributive Normalform transformiert. Dies muss dem Regelsystem hinzugefügt werden, wenn eine solche Simplifikation – wie in MATHEMATICA – nicht automatisch erfolgt.

[6] Präziser: Ist a_d die Anzahl der Terme vom Grad d in einem polynomialen Ausdruck P mit den gegebenen Kernen und $\phi(P) = \sum_d a_d\, T^d \in \mathbb{N}[T]$ die zugehörige erzeugende Funktion, so wird $\phi(P)$ bzgl. der Wohlordnung

$$\sum_d a_d\, T^d > \sum_d b_d\, T^d \Leftrightarrow a_e > b_e \text{ für } e = \max(d : a_d \neq b_d)$$

in jedem Schritt kleiner.

```
trigsum0={ (* 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) };
```

Regeln werden in MATHEMATICA in der Form `lhs /; bool -> rhs` notiert, was eine Kurzform für die interne Darstellung als `Rule[Condition[lhs, bool], rhs]` ist. Wie für viele zweistellige Funktionen stellt MATHEMATICA Infix-Notationen in Operatorform für Regelanwendungen zur Verfügung, wobei zwischen `/.` (`ReplaceAll` – einmalige Anwendung) und `//.` (`ReplaceRepeated` – rekursive Anwendung) unterschieden wird, was sich in der Wirkung ähnlich unterscheidet wie Evaluationstiefe 1 und maximale Evaluationstiefe. Formale Parameter werden durch einen Unterstrich, etwa als `x_`, gekennzeichnet, dem weitere Information über den Typ oder eine Default-Initialisierung folgen können. Wir wollen darauf nicht weiter eingehen, da sich derartige Informationen auch in den konditionalen Teil der Regel integrieren lassen.

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 //. trigsum0
```

$$\frac{(1-\cos(2\,x))^3\,\sin(x)}{8}$$

Dieser Ausdruck muss erst expandiert werden, damit die Regeln erneut angewendet werden können.

```
% // Expand;
% //. trigsum0
```

$$\frac{\sin(x)}{8}+\frac{3\,(1+\cos(4\,x))\,\sin(x)}{16}-\frac{3\,(-\sin(x)+\sin(3\,x))}{16}-\frac{(1+\cos(4\,x))\,(-\sin(x)+\sin(3\,x))}{32}$$

usw., ehe nach vier solchen Schritten schließlich 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 nach jedem Simplifikationsschritt noch die Überführung in die rationale Normalform, also die Funktionalität von `Expand`. Allerdings kann man nicht einfach

```
expandrule = { x_ -> Expand[x] }
```

hinzufügen, denn diese Regel würde ja immer passen und damit das Regelsystem nicht terminieren. In Wirklichkeit ist es sogar noch etwas komplizierter. Zunächst muss für einen Funktionsaufruf wie `Expand` genau wie bei Zuweisungen unterschieden werden, ob der Aufruf bereits während der Regeldefinition (etwa, um eine komplizierte rechte Seite kompakter zu schreiben) oder erst bei der Regelanwendung auszuführen ist. Das spielte bisher keine Rolle, weil auf der rechten Seite stets Funktionsausdrücke standen. MATHEMATICA kennt zwei Regelarten, die mit `->` (Regeldefinition mit Auswertung) und `:>` (Regeldefinition ohne Auswertung) angeschrieben werden. Hier benötigen wir die zweite Sorte von Regeln:

```
expandrule = { x_ :> Expand[x] }
```

Jedoch ist das Ergebnis nicht zufriedenstellend:

```
Sin[x]^7 //. Join[trigsum0,expandrule]
```

$$\frac{\sin(x)^5}{2} - \frac{\cos(2\,x)\,\sin(x)^5}{2}$$

Zwar wird expandiert, aber dann mitten in der Rechnung aufgehört. Der Grund liegt in der Art, wie MATHEMATICA Regeln anwendet. Um unendliche Schleifen zu vermeiden, wird die Arbeit beendet, wenn sich nach Regelanwendung am Ausdruck nichts mehr ändert. Außerdem werden Regeln erst auf den Gesamtausdruck angewendet und dann auf Teilausdrücke. Da auf den Gesamtausdruck des Zwischenergebnisses (nur) die `expandrule`-Regel passt, deren Anwendung aber nichts ändert, hört MATHEMATICA deshalb auf und versucht sich gar nicht erst an Teilausdrücken.

Wir brauchen also statt der bisher betrachteten Lösungen eine zusätzliche Regel, die genau dem Distributivgesetz für Produkte von Summen entspricht und auch nur in diesen Fällen greift. Das kann man durch eine einzige weitere Regel erreichen:

```
expandrule = { (a_+b_)*c_ -> a*c+b*c }
```

Nun zeigt die Simplifikation das erwünschte Verhalten:

```
Sin[x]^7 //. Join[trigsum0,expandrule]
```

$$\frac{35\,\sin(x)}{64} - \frac{21\,\sin(3\,x)}{64} + \frac{7\,\sin(5\,x)}{64} - \frac{\sin(7\,x)}{64}$$

und wir setzen

```
trigsum = Join[trigsum0,expandrule]
```

Das Regelsystem trigexpand

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 Kerne eines Ausdrucks verringert. Auf diese Weise liefert die anschließende Berechnung der rationalen Normalform oft ein besseres Ergebnis.

Mit dem REDUCE-Regelsystem

```
trigexpand0:={
      sin(~x+~y) => sin(x) * cos(y) + cos(x) * sin(y),
      cos(~x+~y) => cos(x) * cos(y) - sin(x) * sin(y) };
```

lassen sich zunächst Linearkombinationen von Argumenten trigonometrischer Funktionen auflösen. Wie oben bleiben danach Kerne mit Mehrfachwinkel-Argumenten der Form $\sin(n\,x)$ oder $\cos(n\,x)$ mit $n \in \mathbb{N}$ übrig, die weiter expandiert werden müssen.

Mathematisch kann man die entsprechenden Regeln aus der *Moivreschen Formel* für komplexe Zahlen und den Potenzgesetzen herleiten: Aus

$$\cos(n\,x) + \mathrm{i}\,\sin(n\,x) = e^{\,\mathrm{i}nx} = (e^{\,\mathrm{i}x})^n = (\cos(x) + \mathrm{i}\,\sin(x))^n$$

und den binomischen Formeln ergibt sich durch Vergleich der Real- und Imaginärteile unmittelbar

$$\cos(n\,x) = \sum_{2k\le n} (-1)^k \binom{n}{2k} \sin(x)^{2k}\cos(x)^{n-2k}$$

und

$$\sin(n\,x) = \sum_{2k<n} (-1)^k \binom{n}{2k+1} \sin(x)^{2k+1} \cos(x)^{n-2k-1}$$

Auch so komplizierte Formeln lassen sich in Regeln unterbringen, denn der Aufbau des Substituenten ist nicht auf einfache arithmetische Kombinationen beschränkt, sondern kann beliebige, auch selbst definierte Funktionsaufrufe heranziehen, mit welchen die rechte Seite der Regelanwendung aus den unifizierten Teilen zusammengebaut wird. In REDUCE etwa könnte man für die zweite Formel eine Funktion `Msin` als

```
procedure Msin(n,x);
  begin scalar k,l;
  while (2k<n) do
  << l:=l+(-1)^k*binomial(n,2k+1)*sin(x)^(2k+1)*cos(x)^(n-2k-1); k:=k+1; >>;
  return l;
end;
```

definieren, eine Regel

```
trig2:={sin(~n*~x) => Msin(n,x) when fixp(n) and (n>1)};
```

vereinbaren und damit $\sin(7\,y)$ auflösen zu

```
     sin(7*y) where trig2;
```

$$\sin(y)\,\left(7\,\cos(y)^6 - 35\,\cos(y)^4\,\sin(y)^2 + 21\,\cos(y)^2\,\sin(y)^4 - \sin(y)^6\right).$$

Einfacher ist es allerdings auch in diesem Fall, die Regeln aus dem Ansatz

$$\sin(k\,x) = \sin((k-1)\,x + x) = \sin((k-1)\,x)\,\cos(x) + \cos((k-1)\,x)\,\sin(x)$$

herzuleiten, den wir wieder rekursiv zur Anwendung bringen. Unser gesamtes REDUCE-Regelsystem hat dann die 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, trigonometrische Ausdrücke aus der Klasse *TrigPoly* durch polynomiale Ausdrücke mit nur noch wenigen einfachen trigonometrischen Kernen zu ersetzen. Hängen die Argumente nur ganzzahlig von einer Variablen x ab, so können wir das System wiederum zu einer kanonischen oder Normalform erweitern. Dazu müssen allein noch mit der Regel $\sin(x)^2 + \cos(x)^2 = 1$ höhere Potenzen eines der Kerne $\sin(x)$ oder $\cos(x)$ durch den anderen ersetzt werden. Vereinbaren wir zum Beispiel das Regelsystem

```
  trigexpandsin:=append(trigexpand, { sin(~x)^2 => 1-cos^2(x) });
```

in REDUCE-Notation, so gilt der folgende Satz:

Satz 5 *Man kann jeden polynomialen Ausdruck in* $\sin(k\,x)$ *und* $\cos(k\,x)$, $k \in \mathbb{Z}$, *mit rationalen Koeffizienten durch obiges Regelsystem* ***trigexpandsin*** *in einen Ausdruck der Form*

$$P(\cos(x)) + \sin(x)\,Q(\cos(x)) \tag{TE}$$

verwandeln, wobei $P(z)$ *und* $Q(z)$ *Polynome mit rationalen Koeffizienten sind.*

Diese Darstellung ist eindeutig. Genauer: Werden die rationalen Koeffizienten in ihrer kanonischen Form als gekürzte Brüche dargestellt, so ist die Darstellung (TE) eine kanonische Form für die Klasse der betrachteten Ausdrücke.

Beweis: Es ist wiederum nur die Eindeutigkeit zu zeigen. Die Identität

$$P_1(\cos(x)) + \sin(x)\,Q_1(\cos(x)) = P_2(\cos(x)) + \sin(x)\,Q_2(\cos(x))$$

als stetige reellwertige Funktionen gilt aber genau dann, wenn $P(\cos(x)) + \sin(x)\,Q(\cos(x))$ mit den Polynomen $P = P_1 - P_2$ und $Q = Q_1 - Q_2$ als reellwertige Funktion die Nullfunktion $0(x)$ ist. Wir haben also nur zu zeigen, dass aus $P(\cos(x)) + \sin(x)\,Q(\cos(x)) = 0(x)$ bereits folgt, dass $P(z)$ und $Q(z)$ beides Nullpolynome sind.

Aus $P(\cos(x)) + \sin(x)\,Q(\cos(x)) = 0(x)$ folgt aber nach der Substitution $x = -x$ auch $P(\cos(x)) - \sin(x)\,Q(\cos(x)) = 0(x)$ und schließlich $P(\cos(x)) = \sin(x)\,Q(\cos(x)) = 0(x)$.

Aus der zweiten Beziehung folgt zunächst $Q(\cos(x_0)) = 0$ für alle x_0 mit $\sin(x_0) \neq 0$. $Q(\cos(x))$ verschwindet also überall außer auf einer diskreten Menge von Argumenten und ist als stetige Funktion damit selbst die Nullfunktion: $Q(\cos(x)) = 0(x)$.

Damit müssen aber P und Q selbst Nullpolynome sein, da ein nichttriviales Polynom nur endlich viele Nullstellen besitzt, aber unendlich viele Werte x_i angegeben werden können, für die $\cos(x_i) = c_i$ jeweils verschiedene Werte annimmt, für die dann $P(c_i) = Q(c_i) = 0$ gilt.

Daraus folgt die Behauptung. □

Mit diesen beiden Strategien lassen sich die drei Integrationsaufgaben, mit denen wir diesen Abschnitt begonnen hatten, zufriedenstellend lösen, wobei bei f_3 ggf. durch Substitution $y = \frac{x}{6}$ der Hinweis gegeben werden muss, dass es sich um ganzzahlige Vielfache eines gemeinsamen trigonometrischen Arguments handelt. Dem Nutzer stehen in den meisten CAS verschiedene fest eingebaute Transformationsfunktionen zur Verfügung, die eine der beschriebenen Simplifikationsstrategien zur Anwendung bringen. In MuPAD sind dies insbesondere die Befehle `expand, combine` und `rewrite`.

Simplifikation rationaler trigonometrischer Ausdrücke

Mit *TrigRat* bezeichnen wir die Menge der rationaler trigonometrischer Ausdrücke, worunter wir Quotienten von Ausdrücken der Klasse *TrigPoly* verstehen wollen. Mathematisch handelt es sich dabei um meromorphe Funktionen, und wir wollen die Ausdrücke auch in dieser Klasse mathematisch interpretieren.

Für Integrale von derartigen Ausdrücken liefert FriCAS 1.3.2, MuPAD 8, Maple 2016, Reduce 2017, Mathematica 11 und Maxima 5.37[7] die folgenden Resultate:

$$\int \frac{1}{\cos(x)^4}\,dx$$

$$= \frac{\sin(x)\,\left(2\,\cos(x)^2 + 1\right)}{3\,\cos(x)^3} \qquad \text{(FriCAS, Maple, MuPAD)}$$

$$= \frac{1}{3}\,\tan(x)\,\sec^2(x) + \frac{2}{3}\,\tan(x) \qquad \text{(Mathematica)}$$

$$= \frac{1}{3}\,\tan(x)^3 + \tan(x) \qquad \text{(Maxima)}$$

[7] Das erste Resultat von Maxima ist die direkte Ausgabe von `integrate`, das zweite ergibt sich nach Anwendung von `trigsimp`.

$$\int \frac{1}{\sin(x)^2\,(1-\cos(x))}\,dx$$

$$= \frac{-2\,\cos(x)^2+2\,\cos(x)+1}{(3\,\cos(x)-3)\,\sin(x)} \qquad \text{(FriCAS)}$$

$$= \frac{1}{4}\,\tan\left(\frac{x}{2}\right) - \frac{1}{2\,\tan\left(\frac{x}{2}\right)} - \frac{1}{12\,\tan\left(\frac{x}{2}\right)^3} \qquad \text{(Maple, MuPAD)}$$

$$= \frac{3\,\tan\left(\frac{x}{2}\right)^4 - 6\,\tan\left(\frac{x}{2}\right)^2 - 1}{12\,\tan\left(\frac{x}{2}\right)^3} \qquad \text{(Reduce)}$$

$$= \frac{1}{12}\,(\cos(2\,x)-2\,\cos(x))\,\csc\left(\frac{x}{2}\right)^3\sec\left(\frac{x}{2}\right) \qquad \text{(Mathematica)}$$

$$= 2\left(\frac{\sin(x)}{8\,(\cos(x)+1)} - \frac{(\cos(x)+1)^3\left(\frac{6\,\sin(x)^2}{(\cos(x)+1)^2}+1\right)}{24\,\sin(x)^3}\right)$$

$$= \frac{2\,\cos(x)^3-3\,\cos(x)-1}{3\,\sin(x)^3} \qquad \text{(Maxima)}$$

$$\int \frac{1}{\sin(2\,x)\,(3\,\tan(x)+5)}\,dx$$

$$= \frac{1}{20}\left(-\log\left(\frac{9\,\tan(x)^2+30\,\tan(x)+25}{\tan(x)^2+1}\right)+\log\left(\frac{\tan(x)^2}{\tan(x)^2+1}\right)\right) \qquad \text{(FriCAS)}$$

$$= -\frac{1}{10}\,\log\left(\frac{5}{\tan(x)}+3\right) \qquad \text{(MuPAD)}$$

$$= -\frac{1}{10}\,\log\left(3\,\tan(x)+5\right)+\frac{1}{10}\,\log\left(\tan(x)\right) \qquad \text{(Maple, Reduce)}$$

$$= \frac{1}{10}\,(\log(\sin(x)-\log(3\,\sin(x)+5\,\cos(x))) \qquad \text{(Mathematica)}$$

$$= -\log\left(\frac{17\,\sin(2\,x)^2+30\,\sin(2\,x)+17\,\cos(2\,x)^2+16\,\cos(2\,x)+17}{17}\right)$$

$$-\log\left(\sin(x)^2+\cos(x)^2+2\,\cos(x)+1\right)-\frac{1}{20}\log\left(\sin(x)^2+\cos(x)^2-2\,\cos(x)+1\right)$$

$$= -\frac{1}{20}\left(\log\left(\frac{34+30\,\sin(2\,x)+16\,\cos(2\,x)}{17}\right)-\log\left(2+2\,\cos(x)\right)-\log\left(2-2\,\cos(x)\right)\right)$$

(Maxima)

Es ist bereits eine kleine Herausforderung, die semantische Gleichwertigkeit dieser syntaktisch verschiedenen Ergebnisse festzustellen. Nach unseren bisherigen Überlegungen kann das entsprechende Normalformproblem in der Klasse *TrigRat* wie folgt gelöst werden:

- Ersetze zunächst alle trigonometrischen Funktionen ausschließlich durch sin und cos.
- Drücke alle Argumente der verbleibenden Funktionen als ganzzahlige Vielfache eines einzigen Arguments x aus.
- Wende auf diesen Ausdruck das Regelsystem `trigexpand` an, um einen rationalen Ausdruck allein in den Kernen $\sin(x)$ und $\cos(x)$ zu produzieren.
- Berechne die rationale Normalform dieses Ausdrucks, welche die Gestalt $F = \frac{P(\sin(x),\cos(x))}{Q(\sin(x),\cos(x))}$ mit Ausdrücken $P(\sin(x),\cos(x)),\ Q(\sin(x),\cos(x)) \in TrigPoly$ hat.
- Berechne mit `trigexpandsin` die Normalform von $P(\sin(x),\cos(x))$.
  Es gilt $F = 0$ genau dann, wenn $P(\sin(x),\cos(x))$ zu null vereinfacht.

Die Berechnungsverfahren für die oben angeführten Integrationsaufgaben gründen meist auf der Möglichkeit, die Kerne $\sin(x)$ und $\cos(x)$ durch $\tan\left(\frac{x}{2}\right)$ auszudrücken und damit die Zahl der verschiedenen Kerne zu reduzieren. Dies ist mit folgendem Regelsatz möglich, nachdem alle anderen Winkelfunktionen allein durch $\sin(x)$ und $\cos(x)$ 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 6 *Sei $R(z) = \frac{P(z)}{Q(z)}$ eine rationale Funktion. Dann kann*

$$\int R\left(\tan\left(\frac{x}{2}\right)\right)\,dx$$

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

$$\int R\left(\tan\left(\frac{x}{2}\right)\right)\,dx = \int \frac{R(y)}{1+y^2}\,dy$$

erhalten. □

Das Regelsystem trigtoexp

Auf trigonometrische Ausdrücke angewendet bewirken `combine` (Produkte in Winkelsummen) und `expand` (Winkelsummen in Produkte) die obigen Umformungen, während man mit `rewrite` (MuPAD) oder `convert` (Maple) trigonometrische Funktionen und deren Umkehrfunktionen in Ausdrücke mit exp und log von komplexen Argumenten umformen kann. Dies entspricht dem Reduce-Regelsatz

```
trigtoexp:={
        sin(~x) => (exp(i*x)-exp(-i*x))/(2*i),
        cos(~x) => (exp(i*x)+exp(-i*x))/2 };
```

der zusammen mit Regeln für $\tan(x)$ und $\cot(x)$ weiter ausgebaut werden kann, um rationale Ausdrücke in diesen Kernen, also im Wesentlichen Ausdrücke der Klasse *TrigRat*, in rationale Ausdrücke mit einem einzigen Kern $\mathrm{e}^{\mathrm{i}\,x}$ umzuwandeln.

Auch dieses Regelsystem führt auf eine effizient berechenbare Normalform in der Klasse *TrigRat*, wenn das CAS in der Lage ist, die Kerne $\mathrm{e}^{\mathrm{i}\,x}$ zu identifizieren. Es ist dabei zu berücksichtigen, dass Koeffizienten aus dem Grundbereich $\mathbb{Q}[\mathrm{i}]$ der rationalen Gaussschen Zahlen entstehen und die Ausdrücke semantisch als Funktionen über den komplexen Zahlen zu interpretieren sind, die Begründung für die Gültigkeit der angewendeten Umformungen also in eine adäquate mathematische Theorie (etwa der komplexen meromorphen Funktionen) einzubetten ist.

Trigonometrische Simplifikationsstrategien in verschiedenen CAS

In der folgenden Tabelle sind die Namen der Funktionen in den einzelnen Systemen einander gegenübergestellt, die dieselben Transformationen wie oben beschrieben bewirken.

Wirkung	Argumentsummen auflösen	Produkte zu Mehrfachwinkeln	Trig $\mapsto$ Exp	Exp $\mapsto$ Trig
Maxima	`trigexpand`	`trigreduce`	`exponentialize`	`demoivre`
Maple	`expand`	`combine`	`convert(u,exp)`	`convert(u,trig)`
Mathematica	`TrigExpand`	`TrigReduce`	`TrigToExp`	`ExpToTrig`
MuPAD	`expand(u)`	`combine(u, 'sincos')`	`rewrite(u,'exp')`	`rewrite(u, 'sincos')`
Reduce	`trigsimp(u, expand)`	`trigsimp(u, combine)`	`trigsimp(u, expon)`	`trigsimp(u, trig)`

Tabelle 5: 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.

Maxima und Mathematica erlauben die Definition eigener Regelsysteme, mit denen man die intern vorhandenen Transformationsmöglichkeiten erweitern kann. Allerdings kann man in Maxima Regelsysteme nur unter großen Schwierigkeiten lokal anwenden.

Auch Reduce kennt nur wenige eingebaute Regeln, was sich insbesondere für die Arbeit mit trigonometrischen Funktionen als nachteilig erweist. Seit der Version 3.6 gibt es allerdings das Paket `trigsimp`, das Simplifikationsroutinen für trigonometrische Funktionen zur Verfügung stellt. 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 in Analogie zum Substitutionsbefehl auch als lokal gültige Regeln anwenden kann. Insbesondere letzteres erlaubt es, gezielt Umformungen mit überschaubaren Seiteneffekten auf Ausdrücken vorzunehmen.

3.8 Das allgemeine Simplifikationsproblem

Betrachten wir zum Abschluss dieses Kapitels, wie sich die verschiedenen CAS bei der Simplifikation komplexerer Ausdrücke verhalten.

1. Beispiel:

```
u1:=(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, den wir bereits weiter oben betrachtet hatten, 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

$$A_1 = \frac{\cos(x)\,(\,\mathrm{e}^x + 1)}{x^2}\,.$$

Eine genauere Analyse mit REDUCE zeigt, dass hierfür (neben der Reduktion der verschiedenen exp-Kerne auf einen einzigen) nur die Regel $\cos(x)^2 \Rightarrow 1 - \sin(x)^2$ anzuwenden ist.

```
u1 where sin(x)^2 => 1-cos(x)^2;
```

MAPLE hatte noch in der Version 3 Schwierigkeiten, den kleinsten gemeinsamen Nenner von u_1 zu erkennen, der ja hinter vielen verschiedenen `exp`-Kernen verborgen ist. In den Versionen 4 – 6 hat `normal` das korrekte Ergebnis e^x gefunden. Seither hat MAPLE mit diesem Ausdruck die alten Schwierigkeiten. Die Zusammenfassung der `exp`-Kerne ist nur mit `normal(.., expanded)` möglich.

`simplify` vereinfacht den Ausdruck in MAPLE und auch in MUPAD zu A_1, REDUCE und MAXIMA kommen mit `trigsimp`, MATHEMATICA mit `Simplify` bzw. `Together` zu demselben Ergebnis.

2. Beispiel:

```
u2:=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)*(exp(3/2*x)+exp(x/2))*(1-exp(-2*x));
```

Hier sind die Simplifikationsregeln für trigonometrische und hyperbolische Funktionen anzuwenden. Ein Plot der Funktion legt nahe, dass der Ausdruck zum Nullausdruck vereinfachen sollte.

Das kann man etwa durch Umformung in Exponentialausdrücke erreichen. Mit REDUCE und MAXIMA kann man auf diese Weise erkennen, dass dieser Ausdruck null-äquivalent ist. Auch MAPLE 2016 findet dies mit `convert(u2, exp)` heraus, während `simplify(u2)` nicht zum Ziel führt. In MUPAD und MATHEMATICA genügt ein Aufruf der Funktion `Simplify`.

System	Beispiel `u1`	Beispiel `u2`
MAXIMA	`trigsimp(u1))`	`expand(exponentialize(u2))`
MAPLE	`simplify(u1)`	`convert(u2,exp)`
MATHEMATICA	`u1 // Simplify`	`u2 // Simplify`
MUPAD	`simplify(u1)`	`simplify(u2)`
REDUCE	`trigsimp(u1)`	`trigsimp(u2,expon)`

Tabelle 6: Simplifikation von u_1 und u_2 auf einen Blick

3. Beispiel: (aus [5, S. 81])

```
u3:=log(tan(x/2)+sec(x/2))-arcsinh(sin(x)/(1+cos(x)));
```

Ein Plot über einem reellen Intervall in der Nähe von $x = 0$ legt wieder nahe, dass es sich um einen Nullausdruck handelt. Dies vermag jedoch keines der Systeme ohne weitere Hinweise zu erkennen, obwohl auch hier das Vorgehen für einen Mathematiker mit geübtem Blick sehr transparent ist: Wegen $\operatorname{arcsinh}(a) = \log\left(a + \sqrt{a^2+1}\right)$ ist arcsinh durch einen logarithmischen Ausdruck zu ersetzen und dann die beiden Logarithmen zusammenzufassen. Dies ist in MAPLE und MUPAD durch eingebaute Funktionen (`convert(u3,ln)` oder `rewrite(u3,'log')`) 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 A dieser beiden Logarithmen mit `simplify`, so wartet etwa MATHEMATICA mit folgendem Ergebnis auf:

$$\log\left(\sec\left(\frac{x}{2}\right) + \tan\left(\frac{x}{2}\right)\right) - \log\left(\sqrt{\sec\left(\frac{x}{2}\right)^2} + \tan\left(\frac{x}{2}\right)\right)$$

Die Simplifikation endet an der Stelle, wo $\sqrt{x^2}$ nicht zu x vereinfacht wird, obwohl dies hier aus dem Kontext heraus erlaubt wäre (wenn man voraussetzt, dass es sich um reelle Funktionen handelt): $\log\left(\sec\left(\frac{x}{2}\right)+\tan\left(\frac{x}{2}\right)\right)$ hat als Definitionsbereich $D=\left\{x : \cos\left(\frac{x}{2}\right)>0\right\}$. Mit zusätzlichen Annahmen kommt MATHEMATICA (ab 5.0) weiter.

```
u3a = u3 /. Arch2Log // Simplify
Simplify[u3a, Assumptions -> {Element[x,Reals]}]
```

$$-\log\left(\left|\sec\left(\frac{x}{2}\right)\right|+\tan\left(\frac{x}{2}\right)\right)+\log\left(\sec\left(\frac{x}{2}\right)+\tan\left(\frac{x}{2}\right)\right)$$

Wir sehen an dieser Stelle, dass die Vereinfachung von u_3 zu null ohne weitere Annahmen über x mathematisch nicht korrekt wäre, da für negative Werte für $\sec\left(\frac{x}{2}\right)$ die beiden Logarithmanden verschieden sind. Beschränken wir x auf das reelle Intervall $-1<x<1$, in dem alle beteiligten Funktionen definiert und reellwertig sind, so vermag MATHEMATICA (ab 5.0) die Null-Äquivalenz dieses Ausdrucks zu erkennen.

```
Simplify[u3a, Assumptions -> { (-1<x) And (x<1) }]
```

$$0$$

Sehen wir, was die anderen CAS unter dieser Voraussetzung erkennen.

MAPLE 2016: Wir müssen auch hier nachhelfen und das System veranlassen, die Hyperbelfunktionen in Logarithmen umzuwandeln (`convert`) und diese zusammenzufassen (`combine`). Das Ergebnis kann MAPLE aber auch unter der Annahme $-1<x<1$ nicht vernünftig vereinfachen.

```
u3a:=convert(u3,ln);
u3b:=combine(u3a,ln);
assume(-1<x,x<1);
simplify(u3b);
```

$$\ln(2)+\ln\left(1+\sin\left(\frac{x}{2}\right)\right)+\ldots$$

Ein ähnliches Vorgehen führt in MUPAD 8 zum Ziel.

```
assume(-1<x & x<1);
u3a=rewrite(u3,'log')
simplify(u3a)
```

$$0$$

Mit REDUCE kann man dieselben Umformungen durch geeignete Regelsysteme erreichen.

```
A1:=u3 where asinh(~x)=>log(x+sqrt(1+x^2));
A2:=e^A1;
```

Im Ausdruck A1 wurden die Logarithmen nicht zusammengefasst, was aber mit dem Schalter `on combinelogs` erreicht werden kann. Bildet man e^{A1}, so ist es auch mathematisch korrekt die Logarithmen zusammenzufassen, da die verschiedenen Werte des Logarithmus sich vom Hauptwert nur um ein ganzzahliges Vielfaches von $2\pi\,\mathrm{i}$ unterscheiden. Die weitere Vereinfachung ergibt

```
trigsimp(A2); A3:=subs(y=2x,ws);
```

$$\frac{\left|1-\sin(y)^2\right|\,(1+\sin(y))}{|1-\sin(y)^2|\,\sin(y)+\sqrt{1-\sin(y)^2}\,\cos(y)}$$

```
A3 where sin(y)^2=>1-cos(y)^2;
```

liefert dann

$$\frac{\cos(y)\,(\sin(y)+1)}{|\cos(y)|+\cos(y)\,\sin(y)}.$$

Wir sehen hieran bereits, dass man sich offensichtlich stets genügend komplizierte Simplifikationsprobleme ausdenken kann, die mit dem vorhandenen Instrumentarium nicht aufgelöst werden können. Es gilt jedoch noch mehr:

Satz 7 *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 gehört damit zur Klasse der algorithmisch unlösbaren Probleme.

Der Beweis dieses Satzes fußt auf der negativen Antwort zum 10. Hilbertschen Problem (Lösbarkeit polynomialer Gleichungen in ganzen Zahlen), die Matiyasewitsch und Roberts Ende der 1960er Jahre gefunden haben.

Kapitel 4

Algebraische Zahlen

Wir hatten im Kapitel 3 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 kanonische Formen (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 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 Maxima allein durch den Aufruf von `simplify` gelöst. Mathematica stellt mit `RootReduce` spezielle Simplifikationsroutinen für geschachtelte Wurzelausdrücke zur Verfügung, was im stärkeren `FullSimplify` integriert ist. Maxima bietet eine solche Simplifikationsroutine `sqrtdenest` im Paket `sqdnst`. Mit Reduce und Axiom habe ich keinen direkten Weg gefunden, diese Aufgabe zu lösen.

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 $p = x^3 + x + 1$ 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 p 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.

4.1 Rechnen mit Nullstellen dritten Grades

Wir wollen dabei die sprachlichen Mittel, welche 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, keine Liste, sondern eine *Ausdruckssequenz* zurück, welche die verschiedenen Nullstellen zusammenfasst. Wir wollen sie gleich in eine Liste verwandeln:

```
sol:=[solve(x^3+x+1,x)];
```

$$\left[-\frac{\%1}{6}+\frac{2}{\%1},\ \frac{\%1}{12}-\frac{1}{\%1}+\frac{\mathrm{i}}{2}\sqrt{3}\left(-\frac{\%1}{6}-\frac{2}{\%1}\right),\ \frac{\%1}{12}-\frac{1}{\%1}-\frac{\mathrm{i}}{2}\sqrt{3}\left(-\frac{\%1}{6}-\frac{2}{\%1}\right)\right]$$

$$\%1 := \sqrt[3]{108+12\sqrt{93}}$$

Wir sehen, dass Maple das Problem der voneinander abhängigen Kerne auf eine clevere Art und Weise löst: Das Endergebnis wird unter Verwendung einer neuen Variablen %1 formuliert[1], mit der ein gemeinsamer Teilausdruck, welcher in der Formel mehrfach vorkommt, identifiziert wird. Auf diese Information wird bei späteren Simplifikationen zurückgegriffen, was für unsere Aufgabenstellung bereits ausreichend ist:

```
p:=product(x-op(i,sol),i=1..3);
```

$$\left(x+\frac{\%1}{6}-\frac{2}{\%1}\right)\left(x-\frac{\%1}{12}+\frac{1}{\%1}-\frac{\mathrm{i}}{2}\sqrt{3}\left(-\frac{\%1}{6}-\frac{2}{\%1}\right)\right)\left(x-\frac{\%1}{12}+\frac{1}{\%1}+\frac{\mathrm{i}}{2}\sqrt{3}\left(-\frac{\%1}{6}-\frac{2}{\%1}\right)\right)$$

```
p1:=expand(p);
```

$$\frac{1}{2}+x+x^3+\frac{\sqrt{93}}{18}-\frac{8}{108+12\sqrt{93}}$$

`expand` ist der polynomiale Normalformoperator, d.h. multipliziert Produkte aus und fasst gleichartige Terme zusammen, wobei x und $\%1 = \sqrt[3]{108+12\sqrt{93}}$ als Kerne betrachtet werden. Allerdings wird der zweite Kern von `indets(p)` nicht mit ausgegeben.

Der gemeinsame Teilterm $u = 108+12\sqrt{93}$ kann sogar durch eine Variable ersetzt werden, so dass wir diese Umformungen nachvollziehen können. Beachten Sie, dass der Kern `%1` selbst nicht so einfach zugänglich ist, da er im Ausdruck in zwei Formen vorkommt, als $u^{1/3}$ und als $u^{-1/3}$, denn Maple stellt den Nenner `1/%1` nicht als $(u^{1/3})^{-1}$, sondern als $u^{-1/3}$ dar.

```
p2:=subs((108+12*93^(1/2))=u,p);
```

[1] Das ist nicht ganz korrekt und in der Terminalversion von Maple 2016 auch anders: In der Dokumentation wird darauf hingewiesen, dass es sich nur um „Pattern“, nicht aber um Variablen handelt. Gleichwohl ist Maple aber in der Lage, solche gleichartigen Teile eines Ausdrucks ohne viel Aufwand zu identifizieren, was sicher damit zu tun hat, dass es sich um Referenzen auf dieselben Objekte handelt – die hier entscheidende Eigenschaft einer Variablen.

$$\left(x+\frac{\sqrt[3]{u}}{6}-\frac{2}{\sqrt[3]{u}}\right)\cdot\left(x-\frac{\sqrt[3]{u}}{12}+\frac{1}{\sqrt[3]{u}}-\frac{\mathrm{i}\sqrt{3}}{2}\left(-\frac{\sqrt[3]{u}}{6}-\frac{2}{\sqrt[3]{u}}\right)\right)$$
$$\cdot\left(x-\frac{\sqrt[3]{u}}{12}+\frac{1}{\sqrt[3]{u}}+\frac{\mathrm{i}\sqrt{3}}{2}\left(-\frac{\sqrt[3]{u}}{6}-\frac{2}{\sqrt[3]{u}}\right)\right)$$

```
expand(p2);
```

$$x+x^3+\frac{u}{216}-\frac{8}{u}$$

Durch Rationalmachen, d.h. durch Multiplizieren mit dem zu u konjugierten Wurzelausdruck, kann man das Absolutglied des Polynoms p_1 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(p1);
```

$$x^3+x+1$$

Alternativ führt `normal(p1)` zum selben Ergebnis, da p_1 ein rationaler Ausdruck in den Kernen x und $v=\sqrt{93}$ ist.

Experimentieren wir weiter mit den Ausdrücken aus der Liste *sol*. 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. Nach unseren bisherigen Betrachtungen (die Ausdrücke enthalten nur zwei Kerne), sollte für diese Vereinfachungen das Kommando `normal` ausreichen.

```
sums:=[seq(sum(sol[i]^k,i=1 .. 3),k=2 .. 9)];
normal(sums);
```

$$\left[-2,\,-3,\,2,\,5,\,6\,\frac{29+3\,\sqrt{3}\sqrt{31}}{\left(9+\sqrt{3}\sqrt{31}\right)^2},\,-7,\,-36\,\frac{29+3\,\sqrt{3}\sqrt{31}}{\left(9+\sqrt{3}\sqrt{31}\right)^2},\,144\,\frac{135+14\,\sqrt{3}\sqrt{31}}{\left(9+\sqrt{3}\sqrt{31}\right)^3}\right]$$

Bei größeren k ergeben sich Probleme mit faktorisierten Nennern in der Normalform, die sich beheben lassen, wenn man mit expandierten Nennern rechnet (was im Zusammenspiel mit algebraischen Vereinfachungen zu einer Normalform führt, wie wir noch sehen werden).

```
normal(sums,expanded);
```

$$[-2,-3,2,5,1,-7,-6,6]$$

Ähnlich ist in MuPAD vorzugehen, wobei zu berücksichtigen ist, dass Nullstellen von Polynomen dritten Grades in der RootOf-Notation ausgegeben werden. Wir kommen darauf weiter unten zurück. Mit der Option `MaxDegree` werden die Nullstellen als Wurzelausdrücke dargestellt:

```
sol=solve(x^3+x+1,x,'MaxDegree',3)
p=(x-sol(1))*(x-sol(2))*(x-sol(3))
p1:=expand(p);
```

$$x+x^3-\frac{\sqrt{31}\sqrt{108}}{108}+\frac{1}{27\left(\frac{\sqrt{31}\sqrt{108}}{108}-\frac{1}{2}\right)}+\frac{1}{2}$$

p hat eine ähnlich komplizierte Gestalt wie das MAPLE-Ergebnis. Der Ausdruck enthält die vier Kerne $\sqrt{3}$, i, $u^{1/3}$ und $u^{-1/3}$, von denen in der expandierten Form p_1 noch u übrig ist, das seinerseits die Kerne $k_1 = \sqrt{31}$ und $k_2 = \sqrt{108}$ in der Konstellation $k_1 \cdot k_2$ enthält. Ein weiteres `simplify` liefert dann bereits $x^3 + x + 1$ zurück.

In der Version 1.4.2 wurde als Resultat von `expand(p)` der Ausdruck

$$x^3 + 3\,x \sqrt[3]{\left(\frac{\sqrt{31}\sqrt{108}}{108} + \frac{1}{2}\right)} \sqrt[3]{\left(\frac{\sqrt{31}\sqrt{108}}{108} - \frac{1}{2}\right)} + 1$$

als Ergebnis ausgegeben, was offensichtlich damit zusammenhing, dass bereits in p neben dem Kern $u = \left(\frac{\sqrt{31}\sqrt{108}}{108} - \frac{1}{2}\right)^{1/3}$ ein zweiter Kern $u' = \left(\frac{\sqrt{31}\sqrt{108}}{108} + \frac{1}{2}\right)^{1/3}$ auftauchte und die rationale Beziehung $u' = \frac{1}{27\,u}$ zwischen beiden „vergessen“ wurde. Da $\sqrt[3]{x}\,\sqrt[3]{y}$ nicht ohne Weiteres zu $\sqrt[3]{x\,y}$ zusammengefasst werden darf, kann diese Beziehung nicht rekonstruiert werden.

MATHEMATICA liefert für unsere Gleichung dritten Grades folgende Antwort:

```
sol=Solve[x^3+x+1==0,x]
```

$$\left\{\left\{x \to -\left(\frac{2}{3\,\left(-9+\sqrt{93}\right)}\right)^{1/3} + \frac{1}{3^{2/3}}\left(\frac{-9+\sqrt{93}}{2}\right)^{1/3}\right\},$$

$$\left\{x \to \frac{-\left(1+\mathrm{i}\sqrt{3}\right)}{2\cdot 3^{2/3}}\left(\frac{-9+\sqrt{93}}{2}\right)^{1/3} + \frac{1-\mathrm{i}\sqrt{3}}{2^{2/3}\left(3\,\left(-9+\sqrt{93}\right)\right)^{1/3}}\right\},$$

$$\left\{x \to \frac{-\left(1-\mathrm{i}\sqrt{3}\right)}{2\cdot 3^{2/3}}\left(\frac{-9+\sqrt{93}}{2}\right)^{1/3} + \frac{1+\mathrm{i}\sqrt{3}}{2^{2/3}\left(3\,\left(-9+\sqrt{93}\right)\right)^{1/3}}\right\}\right\}$$

Das Ergebnis des `Solve`-Operators ist keine Liste oder Menge von Lösungen, sondern eine Substitutionsliste, wie wir sie im Abschnitt 2.6 kennengelernt haben. Solche Substitutionslisten werden auch von MAPLE 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, MAXIMA und REDUCE, konsistenter.

Die einzelnen Komponenten der Lösung können wir durch die Aufrufe `x /. sol[[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 /. sol[[i]])` das erste x als Symbol erhalten bleibt, während für das zweite x die entsprechende Ersetzung $x \mapsto x_i$ ausgeführt wird.

```
p=Product[x-(x /. sol[[i]]),{i,1,3}]
```

`Expand` vereinfacht wieder im Sinne der polynomialen Normalform in einen Ausdruck der Form $x^3 + x + U$, wobei U selbst wieder ein rationaler Ausdruck in im Wesentlichen dem Kern $\sqrt{93}$ ist, welcher mit `Together` zu $x^3 + x + 1$ vereinfacht wird.

Für die Potenzsummen der Nullstellen, die man mit dem Sequenzierungsoperator `Table` aufsammeln kann. Früher bekam man bereits mit `Simplify` eine Liste von ganzen Zahlen. In MATHEMATICA 11 muss man noch ein `Expand` dazwischenschieben:

```
sums=Table[Sum[(x/.sol[[i]])^k,{i,1,3}],{k,2,9}];
sums//Expand//Simplify
```

$$\{-2,-3,2,5,1,-7,-6,6\}$$

`Together` reicht ebenfalls nicht aus und operiert auf einer Reihe von Listenelementen nicht idempotent.

Ähnlich gehen die Systeme AXIOM und MAXIMA vor, wobei hier oft eine genauere Kenntnis speziellerer Designmomente notwendig ist. So liefert etwa AXIOM mit

```
solve(x^3+x+1)
```

das etwas verwunderliche Ergebnis $[x^3 + x + 1 = 0]$. Erst die explizite Aufforderung nach einer Lösung in Radikalen

```
sol:=radicalSolve(x^3+x+1)
```

$$\left[x_{1,2} = \frac{\left(-3\sqrt{-3}\pm 3\right)\left(\sqrt[3]{\frac{\sqrt{\frac{31}{3}}-3}{6}}\right)^2 \pm 2}{\left(3\sqrt{-3}\pm 3\right)\sqrt[3]{\frac{\sqrt{\frac{31}{3}}-3}{6}}},\quad x_3 = \frac{3\left(\sqrt[3]{\frac{\sqrt{\frac{31}{3}}-3}{6}}\right)^2 - 1}{3\sqrt[3]{\frac{\sqrt{\frac{31}{3}}-3}{6}}}\right]$$

wartet mit einem Ergebnis auf, welches 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 (mit `reduce`) und bedarf keiner zusätzlichen Simplifikationsaufrufe, da in diesem CAS der 3. Generation alle Operationen „wissen, was zu tun ist“.

```
p:=reduce(*,[x-subst(x,sol.i) for i in 1..3])
```

$$x^3 + x + 1$$

```
[reduce(+,[subst(x^k,sol.i) for i in 1..3]) for k in 2..9]
```

$$[-2,-3,2,5,1,-7,-6,6]$$

MAXIMA liefert als Ergebnis des `solve`-Operators die Lösungsmenge in Form einer Substitutionsliste in ähnlicher Form wie MATHEMATICA.

```
sol:solve(x^3+x+1,x);
```

$$\begin{aligned}\Big[x &= \left(\frac{\sqrt{31}}{6\sqrt{3}}-\frac{1}{2}\right)^{1/3}\left(-\frac{\sqrt{3}\,\mathrm{i}}{2}-\frac{1}{2}\right) - \frac{\frac{\sqrt{3}\,\mathrm{i}}{2}-\frac{1}{2}}{3\left(\frac{\sqrt{31}}{6\sqrt{3}}-\frac{1}{2}\right)^{1/3}},\\ x &= \left(\frac{\sqrt{31}}{6\sqrt{3}}-\frac{1}{2}\right)^{1/3}\left(\frac{\sqrt{3}\,\mathrm{i}}{2}-\frac{1}{2}\right) - \frac{-\frac{\sqrt{3}\,\mathrm{i}}{2}-\frac{1}{2}}{3\left(\frac{\sqrt{31}}{6\sqrt{3}}-\frac{1}{2}\right)^{1/3}},\\ x &= \left(\frac{\sqrt{31}}{6\sqrt{3}}-\frac{1}{2}\right)^{1/3} - \frac{1}{3\left(\frac{\sqrt{31}}{6\sqrt{3}}-\frac{1}{2}\right)^{1/3}}\Big]\end{aligned}$$

Auch hier kann (z.B.) durch geeignete Listenoperationen das Produkt p gebildet werden.

```
p:apply("*",map(lambda([u],x-subst(u,x)),sol));
```

Wie in Maple reicht die Berechnung der rationalen Normalform mit `ratsimp` aus, um $x^3 + x + 1$ zurückzugewinnen.

Auf dieselbe Weise lassen sich die Potenzsummen berechnen:

```
sums:makelist(apply("+",map(lambda([u],subst(u,x^k)),sol)),k,2,9);
ratsimp(sums);
```

$$[-2, -3, 2, 5, 1, -7, -6, 6]$$

4.2 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 Substitution $x \mapsto y - \frac{a}{3}$ in die *reduzierte Form* (alle Rechnungen mit Maxima)

```
f:x^3+a*x^2+b*x+c;
ratsimp(subst(x=y-a/3,f));
```

$$y^3 + y\left(b - \frac{a^2}{3}\right) + \left(c - \frac{ab}{3} + \frac{2a^3}{27}\right) = y^3 + p\,y + q$$

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(subst(y=u+v,g));
```

$$q + p\,u + p\,v + u^3 + v^3 + 3\,u\,v^2 + 3\,u^2\,v$$

Diesen Ausdruck formen wir um zu

$$(u + v)\,(3\,u\,v + p) + (u^3 + v^3 + q)$$

und wählen u und v so, dass

$$3\,u\,v + p = u^3 + v^3 + q = 0$$

gilt. Aus der ersten Beziehung erhalten wir $v = \frac{-p}{3\,u}$, welches, in die zweite Gleichung eingesetzt, nach kurzer Umformung eine quadratische Gleichung für u^3 ergibt:

$$u^3 + v^3 + q = u^3 - \frac{p^3}{27\,u^3} + q = 0 \Rightarrow (u^3)^2 + q \cdot u^3 - \frac{p^3}{27} = 0\,.$$

Die Diskriminante D dieser Gleichung ist

$$D = \left(\frac{q}{2}\right)^2 + \left(\frac{p}{3}\right)^3 \qquad \left(= -\frac{a\,b\,c}{6} + \frac{b^3}{27} + \frac{c^2}{4} + \frac{a^3\,c}{27} - \frac{a^2\,b^2}{108}\right)$$

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 $u^3 = -\frac{q}{2} \pm \mathrm{i} \cdot \sqrt{-D}$ (falls $D < 0$).

Betrachten wir zunächst den Fall $D \geq 0$. 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 $v_1 = \frac{-p}{3\,u_1} = \sqrt[3]{-\frac{q}{2} \mp \sqrt{D}}$. 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 $\omega = -\frac{1}{2} + \frac{\mathrm{i}}{2}\sqrt{3}$ oder $\omega^2 = \bar{\omega} = -\frac{1}{2} - \frac{\mathrm{i}}{2}\sqrt{3}$. Wegen $u_1 v_1 = u_2 v_2 = u_3 v_3$ erhalten wir

$$u_2 = \omega u_1,\ v_2 = \bar{\omega} v_1, \qquad u_3 = \bar{\omega} u_1,\ v_3 = \omega v_1$$

Zusammen ergeben sich die beiden komplexen Lösungen als

$$y_{2,3} = -\frac{u_1 + v_1}{2} \pm \frac{\mathrm{i}}{2}\sqrt{3}\,(u_1 - v_1).$$

Im Falle $D = 0$ gilt $u_1 = v_1$, so dass die beiden komplexen Lösungen zu einer reellen Doppellösung zusammenfallen.

Wir sehen weiter, dass es günstig ist, die beiden Teile u_i, v_i gemeinsam zu führen, d.h. in obiger Formel gleich $v_i = -\frac{p}{3\,u_i}$ zu setzen.

$$\begin{aligned} y_1 &= u_1 - \frac{p}{3\,u_1} \\ y_{2,3} &= -\frac{1}{2}\left(u_1 - \frac{p}{3\,u_1}\right) \pm \frac{\mathrm{i}}{2}\sqrt{3}\left(u_1 + \frac{p}{3\,u_1}\right) \qquad \text{mit} \quad u_1 = \sqrt[3]{-\frac{q}{2} + \sqrt{D}} \end{aligned}$$

In diesen Formeln erkennen wir ohne Mühe die Ausgaben der verschiedenen CAS wieder. Die Trennung der dritten Wurzeln u_1 und v_1 führt zu den Schwierigkeiten, welche weiter oben für MuPAD 1.4.2 dargestellt wurden.

Wesentlich interessanter ist der Fall $D < 0$. 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 der Cardanoschen Formel 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.

$$u^3 = -\frac{q}{2} \pm \mathfrak{i}\sqrt{-D} = R \cdot (\cos(\phi) \pm \mathfrak{i}\sin(\phi))$$

$$\text{mit} \qquad R := \sqrt{\left(\frac{q}{2}\right)^2 - D} = \sqrt{-\left(\frac{p}{3}\right)^3}$$

$$\text{und} \qquad \phi := \arccos\left(\frac{-q}{2R}\right)$$

Wir erhalten daraus die drei komplexen Lösungen

$$u_{k+1} = r \cdot \left(\cos\left(\frac{\phi + 2\,k\pi}{3}\right) \pm \mathfrak{i} \cdot \sin\left(\frac{\phi + 2\,k\pi}{3}\right)\right), \quad k = 0, 1, 2$$

$$\text{mit} \qquad r := \sqrt[3]{R} = \sqrt{-\frac{p}{3}}$$

Da $u \cdot v = -\frac{p}{3}$ 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

$$y_{k+1} = 2\,r \cdot \cos\left(\frac{\phi + 2\,k\,\pi}{3}\right), \quad k = 0, 1, 2.$$

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 $y = 2\sqrt{-\frac{p}{3}}\cos(a)$ gilt wegen der Produkt-Summe-Regel für trigonometrische Funktionen (MAXIMA)

```
y:2*sqrt(-p/3)*cos(a);
u:y^3+p*y+q;
expand(trigreduce(u));
```

$$A = q + 2\,\cos(3\,a)\,\left(-\frac{p}{3}\right)^{3/2} = q + 2\,\cos(3\,a)\,R.$$

Für $a = \frac{\phi + 2\,k\,\pi}{3}$ gilt $3\,a = \phi + 2\,k\,\pi$, somit $\cos(3\,a) = cos(\phi) = -\frac{q}{2\,R}$ und schließlich $A = 0$.
Fassen wir unsere Ausführungen in dem folgenden Satz zusammen.

Satz 8 *Sei $g := y^3 + p\,y + q$ 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

$$y_{2,3} = -\frac{u_1 + v_1}{2} \pm \frac{\mathfrak{i}}{2}\sqrt{3}\,(u_1 - v_1).$$

Für das Rechnen in einem CAS ist es sinnvoll, nur u_1 als Kern einzuführen und $v_1 = -\frac{p}{3\,u_1}$ zu setzen.

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*

$$y_{k+1} = 2\sqrt{-\frac{p}{3}} \cdot \cos\left(\frac{\phi + 2\,k\,\pi}{3}\right),\ k = 0, 1, 2.$$

4.3 * 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 Substitution $x \mapsto y - \frac{a}{4}$ 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

$$\begin{aligned} u\,v\,w &= -\frac{q}{8} \\ u^2 + v^2 + w^2 &= -\frac{p}{2} \\ u^2v^2 + u^2w^2 + v^2w^2 &= \frac{p^2 - 4\,r}{16} \end{aligned}$$

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(subst(y=u+v+w,g));
```

$$\begin{aligned} &r + qu + qv + qw + 2puv + 2puw + 2pvw + u^4 + v^4 + w^4 + pu^2 + pv^2 + pw^2 + 4uv^3 + 4u^3v + \\ &4uw^3 + 4u^3w + 4vw^3 + 4v^3w + 12uvw^2 + 12uv^2w + 12u^2vw + 6u^2v^2 + 6u^2w^2 + 6v^2w^2 \\ &= (u + v + w)(8uvw + q) + (uv + uw + vw)(4(u^2 + v^2 + w^2) + 2p) + (u^2 + v^2 + w^2)^2 + \\ &4(u^2v^2 + u^2w^2 + v^2w^2) + p(u^2 + v^2 + w^2) + r, \end{aligned}$$

der für alle Tripel (u, v, w), welche 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

$$z^3 + \frac{p}{2}\,z^2 + \frac{p^2 - 4\,r}{16}\,z - \frac{q^2}{64} = 0.$$

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

$$\begin{aligned} u\,v\,w &= \pm\frac{q}{8} \\ u^2 + v^2 + w^2 &= -\frac{p}{2} \\ u^2v^2 + u^2w^2 + v^2w^2 &= \frac{p^2 - 4\,r}{16}, \end{aligned}$$

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 in Radikalen 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.

4.4 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.

Wir haben gesehen, dass keines der betrachteten CAS[2] die beiden Fälle $D > 0$ und $D < 0$ bei der Gestaltung der Ausgabe unterscheidet, sondern (außer Reduce) die Cardanosche Formel auch für komplexe Radikanden angesetzt wird.

Maples Antwort etwa lautet

```
s:=[solve(x^3-3*x+1,x)];
```

$$\left[\frac{1}{2}\sqrt[3]{-4+4\,\mathrm{i}\sqrt{3}}+2\,\frac{1}{\sqrt[3]{-4+4\,\mathrm{i}\sqrt{3}}},\right.$$
$$-\frac{1}{4}\sqrt[3]{-4+4\,\mathrm{i}\sqrt{3}}-\frac{1}{\sqrt[3]{-4+4\,\mathrm{i}\sqrt{3}}}+\frac{1}{2}\,\mathrm{i}\sqrt{3}\left(\frac{1}{2}\sqrt[3]{-4+4\,\mathrm{i}\sqrt{3}}-2\,\frac{1}{\sqrt[3]{-4+4\,\mathrm{i}\sqrt{3}}}\right),$$
$$\left.-\frac{1}{4}\sqrt[3]{-4+4\,\mathrm{i}\sqrt{3}}-\frac{1}{\sqrt[3]{-4+4\,\mathrm{i}\sqrt{3}}}-\frac{1}{2}\,\mathrm{i}\sqrt{3}\left(\frac{1}{2}\sqrt[3]{-4+4\,\mathrm{i}\sqrt{3}}-2\,\frac{1}{\sqrt[3]{-4+4\,\mathrm{i}\sqrt{3}}}\right)\right]$$

Letzteres hat den Nachteil, dass man der Lösung selbst nach einer näherungsweisen Berechnung nicht ansieht, dass es sich um reelle Zahlen handelt:

```
evalf(s);
```

$$\left[1.5321-0.1\,10^{-9}\ \mathrm{i},\ -1.8794-0.17321\,10^{-9}\ \mathrm{i},\ 0.34730+0.17321\,10^{-9}\ \mathrm{i}\right]$$

Auch wenn MuPAD an dieser Stelle die kleinen imaginären Anteile unterdrückt und so den Anschein erweckt zu wissen, dass es sich um reelle Lösungen handelt, so ist es doch für alle CAS eine Herausforderung, ohne Kenntnis des Ursprungs dieser Einträge (insbesondere ohne Auswertung von D) *algebraisch* zu deduzieren, dass die *exakten* Werte in s reell sind.

```
sol=solve(x^3-3*x+1,x,'MaxDegree',3)
vpa(sol)
```

$$[1.5321,\ -1.8794,\ 0.34730]$$

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

[2] Allein Derive verhielt sich anders.

```
on fullroots;
s:=solve(x^3-3*x+1,x);
```

$$\left\{x = -2\cos\left(\frac{2\,\pi}{9}\right),\ x = \cos\left(\frac{2\,\pi}{9}\right) + \sqrt{3}\sin\left(\frac{2\,\pi}{9}\right),\ x = \cos\left(\frac{2\,\pi}{9}\right) - \sqrt{3}\sin\left(\frac{2\,\pi}{9}\right)\right\},$$

aber im Fall $D > 0$ das unverständliche Ergebnis

```
s:=solve(x^3+3*x+1,x);
```

$$\left\{\begin{array}{l} x = \sqrt{3}\cosh\left(\dfrac{\operatorname{arcsinh}(1/2)}{3}\right)\ \mathrm{i} + \sinh\left(\dfrac{\operatorname{arcsinh}(1/2)}{3}\right), \\ x = -\sqrt{3}\cosh\left(\dfrac{\operatorname{arcsinh}(1/2)}{3}\right)\ \mathrm{i} + \sinh\left(\dfrac{\operatorname{arcsinh}(1/2)}{3}\right), \\ x = -2\sinh\left(\dfrac{\operatorname{arcsinh}(1/2)}{3}\right) \end{array}\right\}$$

liefert. Die Cardanosche Formel wird ausgegeben, wenn diese Darstellung mit `off trigform` abgeschaltet wird.

Die explizite Verwendung der Lösungsformel für kubische Gleichungen, besonders die trigonometrische Form mit ihren verschiedenen Kernen, birgt auch Klippen für die weitere Vereinfachung der dabei entstehenden Ausdrücke. 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 `subst(a=3,s)` zur Cardanoschen Formel und bei `subst(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.

Reduce liefert in der Standardeinstellung, ähnlich wie Maple, Mathematica und MuPAD,

```
solve(x^3+x+1,x);
```

die auf den ersten Blick verblüffende Antwort

$$\{x = \mathrm{RootOf}(X^3 + X + 1, X)\}$$

Es wird ein Funktionsausdruck mit dem Funktionssymbol RootOf 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. Für Nullstellen allgemeiner Gleichungen höheren Grades gibt es gar keine Darstellung in Radikalen, so dass für solche Zahlen *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 9 *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.*

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 in $R[a]$.*

Eine solche Nullstelle a eines Polynoms $p(x) \in R[x]$ bezeichnet man auch als *algebraische (über R) Zahl*.

Lemma 1 *Unter allen Polynomen*

$$P := \{q(x) \in R[x] \,:\, q(a) = 0 \text{ und } lc(q) = 1\}$$

mit Leitkoeffizient 1 und Nullstelle a gibt es genau Polynom $p(x)$ kleinsten Grades. Dieses ist irreduzibel und jedes andere Polynom $q(x) \in P$ ist ein Vielfaches von $p(x)$.

Beweis: (des Lemmas) Division mit Rest in $R[x]$ ergibt

$$q(x) = s(x) \cdot p(x) + r(x)$$

mit $r = 0$ oder $\deg(r) < \deg(p)$. Wäre $r \neq 0$, so ergäbe sich wegen $q(a) = p(a) = 0$ auch $r(a) = 0$ und $\frac{1}{lc(r)} r(x) \in P$ im Gegensatz zur Annahme, dass $p(x) \in P$ minimalen Grad hat.

Wäre $p(x)$ reduzibel, so wäre a auch Nullstelle eines der Faktoren. Dieser würde also zu P gehören im Widerspruch zur Auswahl von $p(x)$. □

Beweis: (des Satzes) 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 (nicht trivialen) Polynoms $q(x) = \sum_{i=0}^{k-1} r_i x^i$ vom Grad $deg\ q < k$ im Widerspruch zur Aussage des Lemmas. □

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.

RootOf-Symbole werden von fast allen CAS verwendet. Der `Solve`-Operator von Maple und MuPAD etwa 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 3 (MuPAD) eingesetzt. Im 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 3, was man aber durch die Optionen *Cubics* → *True* und *Quartics* → *True* abstellen kann) vor.

Einzig Maxima kennt diese Notation nicht. Allerdings kann man auch hier mit `tellrat(p(a))` ein Symbol a als algebraische Zahl mit dem Minimalpolynom $p(a) = 0$ vereinbaren. Setzt man weiter den Schalter `algebraic:true`, so wird der oben beschriebene Simplifikator bei der Berechnung der rationalen Normalform mit `ratsimp` angewendet.

```
tellrat(a^3+3*a+1);
ratsimp(a^5),algebraic:true;
```

$$-a^2 + 9\,a + 3$$

4.5 Mit algebraischen Zahlen rechnen

Wir hatten gesehen, dass mit Nullstellen von Polynomen, also algebraischen Zahlen, am besten gerechnet werden kann, wenn deren Minimalpolynom bekannt ist.

Oft sind algebraische Zahlen aber in einer Form angegeben, aus der sich dieses Minimalpolynom nicht unmittelbar ablesen lässt. Am einfachsten geht das noch bei Wurzelausdrücken:

Beispiele (über $k = \mathbb{Q}$):

$$\begin{aligned} \alpha_1 &= \sqrt{2} & p_1(x) &= x^2 - 2 \\ \alpha_2 &= \sqrt[3]{5} & p_2(x) &= x^3 - 5 \\ \alpha_3 &= \sqrt{1-\sqrt{2}} & p_3(x) &= x^4 - 2\,x^2 - 1 \end{aligned}$$

Die Irreduzibilität von p_3 ist nicht ganz offensichtlich, kann aber leicht mit MAXIMA 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);
```

$$\left(x^2 - 4\,x - 1\right)\,\left(x^2 + 4\,x - 1\right)$$

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 $n = 3, 4, 6$. MUPAD und MATHEMATICA kennen auch für $n = 5$ interessante Ausdrücke.

```
cos(sym(pi)/5), cos(2*sym(pi)/5);
```

$$\frac{1}{4}\,\sqrt{5} + \frac{1}{4},\ \frac{1}{4}\,\sqrt{5} - \frac{1}{4}$$

In MAPLE können solche Darstellungen seit Version 7 mit `convert(cos(Pi/5),radical)` erzeugt werden. Damit lassen sich Radikaldarstellungen von deutlich mehr algebraischen Zahlen trigonometrischer Natur finden, etwa die von 3^o:

```
convert(cos(Pi/60),radical);
```

$$\left(-\frac{1}{16}\,\sqrt{2} + \frac{1}{8}\,\sqrt{5+\sqrt{5}} + \frac{1}{16}\,\sqrt{2}\,\sqrt{5}\right)\,\sqrt{3} + \frac{1}{8}\,\sqrt{5+\sqrt{5}} + \frac{1}{16}\,\sqrt{2} - \frac{1}{16}\,\sqrt{2}\,\sqrt{5}.$$

Zur Bestimmung des charakteristischen Polynoms von $\cos\left(\frac{\pi}{5}\right)$ benutzen wir

$$\cos\left(5 \cdot \frac{\pi}{5}\right) = \cos(\pi) = -1.$$

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. Die entsprechende MAXIMA-Rechnung lautet

```
u:trigexpand(cos(5*x)+1);
p:expand(subst(sin(x)=sqrt(1-cos(x)^2),u));
```

$$16\,\cos(x)^5 - 20\,\cos(x)^3 + 5\,\cos(x) + 1\,.$$

Um diese Umformungen nicht jedes Mal nacheinander aufrufen zu müssen, definieren wir uns zwei Funktionen

```
sinexpand(u):= expand(subst(cos(x)=sqrt(1-sin(x)^2),trigexpand(u)));
cosexpand(u):= expand(subst(sin(x)=sqrt(1-cos(x)^2),trigexpand(u)));
```

und bekommen nun obige Darstellung durch einen einzigen Aufruf `cosexpand(cos(5*x)+1)`. Ein Regelsystem wäre an dieser Stelle natürlich besser, denn diese Funktionen nehmen die Vereinfachungen nur für Vielfache von x als Argument der trigonometrischen Funktionen vor und nicht für allgemeinere Kerne.

Weiter mit unserem Beispiel:

```
p:subst(cos(x)=z,p);
```

$$16\,z^5 - 20\,z^3 + 5\,z + 1$$

Diese Polynom ist allerdings noch nicht das Minimalpolynom.

```
factor(p);
```

$$(z+1)\,\left(4\,z^2 - 2\,z - 1\right)^2$$

Dasselbe Programm kann man für $\sin\left(\frac{\pi}{5}\right)$ absolvieren:

```
p:subst(sin(x)=z,sinexpand(sin(5*x)));
```

$$16\,z^5 - 20\,z^3 + 5\,z$$

```
factor(p);
```

$$z\,\left(16\,z^4 - 20\,z^2 + 5\right)$$

Also ist der zweite Faktor $q = 16\,z^4 - 20\,z^2 + 5$ das Minimalpolynom von $\sin\left(\frac{\pi}{5}\right)$.

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 $p(x) = x^4 - 10\,x^2 + 1$. Es stellt sich heraus, dass dies auch allgemein gilt:

Satz 10 *Jeder polynomiale Ausdruck $P(a_1, a_2, \ldots, a_s)$ mit rationalen Koeffizienten, der aus algebraischen Zahlen $a_1, a_2, \ldots, a_s$ gebildet werden kann, 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 MAXIMA ergibt

```
tellrat(a^2-2,b^3-5);
algebraic:true;
l:makelist((a+b)^n,n,0,10);
ratsimp(l);
```

$$\left[\begin{array}{l} 1,\ b+a,\ b^2+2\,a\,b+2,\ 3\,a\,b^2+6\,b+2\,a+5,\\ 12\,b^2+(5+8\,a)\,b+20\,a+4,\\ (5+20\,a)\,b^2+(20+25\,a)\,b+4\,a+100,\\ (60+30\,a)\,b^2+(150+24\,a)\,b+200\,a+33,\\ (210+84\,a)\,b^2+(81+350\,a)\,b+183\,a+700,\\ (249+560\,a)\,b^2+(1400+264\,a)\,b+1120\,a+1416,\\ (2520+513\,a)\,b^2+(1944+2520\,a)\,b+4216\,a+3485,\\ (2970+5040\,a)\,b^2+(8525+6160\,a)\,b+6050\,a+21032 \end{array}\right]$$

Wir sehen, dass sich alle Potenzen als Linearkombinationen von sechs Termen 1, a, b, b^2, $a\,b$, $a\,b^2$ darstellen lassen. Mehr als 6 solcher Ausdrücke sind dann sicher linear abhängig. Wir finden 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, $a\,b$, $a\,b^2$ alle verschwinden:

```
f:sum(concat(r,i)*x^i,i,0,6);
p1:subst(x=a+b,f);
p2:expand(ratsimp(p1));
```

$$\begin{array}{l} (r_2+12\,r_4+5\,r_5+60\,r_6+(30\,r_6+20\,r_5+3\,r_3)\,a)\,b^2\\ \qquad +(r_1+6\,r_3+5\,r_4+20\,r_5+150\,r_6+(24\,r_6+25\,r_5+8\,r_4+2\,r_2)\,a)\,b\\ \qquad +(r_1+2\,r_3+20\,r_4+4\,r_5+200\,r_6)\,a+33\,r_6+100\,r_5+4\,r_4+5\,r_3+2\,r_2+r_0 \end{array}$$

```
sys:flatten(makelist(makelist(coeff(coeff(p2,b,j),a,i),j,0,2),i,0,1));
```

$$\begin{array}{l} [\ 33\,r_6+100\,r_5+4\,r_4+5\,r_3+2\,r_2+r_0,\ 150\,r_6+20\,r_5+5\,r_4+6\,r_3+r_1,\\ \ 60\,r_6+5\,r_5+12\,r_4+r_2,\ 200\,r_6+4\,r_5+20\,r_4+2\,r_3+r_1,\\ \ 24\,r_6+25\,r_5+8\,r_4+2\,r_2,\ 30\,r_6+20\,r_5+3\,r_3\ ] \end{array}$$

Das ist ein homogenes lineares Gleichungssystem, desha lb fixieren wir r_6, ehe wir die Lösung bestimmen.

```
sol:solve(append(sys,[r6=1]),vars);
```

$$[[r_0=17, r_1=-60, r_2=12, r_3=-10, r_4=-6, r_5=0, r_6=1]]$$

Ein Polynom mit der Nullstelle $c=a+b$ lautet also

```
p:subst(sol[1],f);
```

$$x^6\ -\ 6\,x^4\ -\ 10\,x^3\ +\ 12\,x^2\ -\ 60\,x\ +\ 17$$

Schließlich können wir noch mit `factor(p)` testen, ob dieses Polynom irreduzibel ist, was hier der Fall ist. 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 $\alpha_1, \ldots, \alpha_s$ algebraische Zahlen über k vom Grad $d_1, \ldots, d_s$, so ergeben sich aus den entsprechenden Minimalpolynomen

$$p_i(x) = x^{d_i} - r_i(x)$$

Ersetzungsformeln

$$\{\alpha_i^{d_i} \Rightarrow r_i(\alpha_i),\ i = 1, \ldots, s\},$$

die es erlauben, jeden polynomialen Ausdruck aus $R := k[\alpha_1, \ldots, \alpha_s]$ in dessen **reduzierte Form** zu transformieren, d.h. ihn als Linearkombination der $D := d_1 \cdots d_s$ Produkte aus der Menge

$$T_{red} := \left\{ \alpha_1^{j_1} \cdots \alpha_s^{j_s} \ :\ 0 \le j_i < d_i \right\}$$

zu schreiben.

Ist nun $c = P(a_1, a_2, \ldots, a_s)$ 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. $\square$

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:

Folgerung 2 *Sind $\alpha_1, \ldots, \alpha_s$ algebraische Zahlen über k vom Grad $d_1, \ldots, d_s$, so bildet die Menge der k-linearen Kombinationen von Elementen aus T_{red} einen Ring.*

Allerdings bilden diese reduzierten Formen nur im Falle $s = 1$ eine kanonische Form (und natürlich nur, wenn die Elemente aus k in einer kanonischen Form darstellbar sind), da zwischen den verschiedenen algebraischen Zahlen $\alpha_1, \ldots, \alpha_s$ algebraische Abhängigkeitsrelationen bestehen können, die lineare Abhängigkeitsrelationen in der Menge T_{red} nach sich ziehen.

Beispiel: $\alpha_1 = \sqrt{2} + \sqrt{3}, \alpha_2 = \sqrt{6}$. Es gilt $\alpha_1^2 - 2\,\alpha_2 - 5 = 0$.

Das Identifikationsproblem kann also für mehrere algebraische Zahlen so nicht gelöst werden.

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 als Linearkombination von Termen aus T_{red} 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 liefert 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. In Reduce muss der Schalter `rationalize` eingeschaltet sein. In Maxima benötigt man noch intimere Systemkenntnisse: Es ist die rationale Normalform im Kontext `algebraic:true` zu berechnen:

```
a:sqrt(2)+sqrt(3)+sqrt(5);
ratsimp(1/a), algebraic:true;
```

$$-\frac{\sqrt{2}\,\sqrt{3}\,\sqrt{5}-2\,\sqrt{3}-3\,\sqrt{2}}{12}$$

Mathematica konnte ich nicht dazu veranlassen, eine entsprechende Darstellung zu finden. Am weitesten kommt man noch mit

```
ToRadicals[RootReduce[1/a]]
```

$$\frac{1}{2}\sqrt{\frac{1}{3}\left(5+\sqrt{6}-\sqrt{10}-\sqrt{15}\right)}$$

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

$$p(x)=x^8-40\,x^6+352\,x^4-960\,x^2+576,$$

d.h. es gilt $a^8-40\,a^6+352\,a^4-960\,a^2+576=0$ oder

$$a^{-1}=\frac{-a^7+40\,a^5-352\,a^3+960\,a}{576}\,.$$

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:

$$a^{-1}=\texttt{subst}\left(a=\sqrt{2}+\sqrt{3}+\sqrt{5},\frac{-a^7+40\,a^5-352\,a^3+960\,a}{576}\right)\,.$$

Das gilt auch allgemein:

Satz 11 *Ist $Q(x)\in k[x]$ das Minimalpolynom der algebraischen Zahl $a\neq 0$, so gilt*

$$a^{-1}=-\,\frac{1}{Q(0)}\cdot\frac{Q(x)-Q(0)}{x}\bigg|_{x=a}$$

Hierbei ist $Q(0)\neq 0$ das Absolutglied des irreduziblen Polynoms $Q(x)$, so dass $Q(x)-Q(0)$ durch x teilbar ist.

$a \neq 0$ besitzt also stets eine Inverse, die sich polynomial durch Potenzen von a und damit als Linearkombination von Elementen aus T_{red} darstellen lässt, wenn wie oben $a \in k[\alpha_1, \ldots, \alpha_s]$ gilt. Aus dem Beweis ergibt sich, dass $Q(x)$ nicht unbedingt das Minimalpolynom sein muss, sondern wir nur von der Eigenschaft $Q(0) \neq 0$ Gebrauch machen.

Für $a \neq 0$ lässt sich ein solches Polynom in den meisten Fällen mit dem weiter oben beschriebenen Verfahren finden. Allerdings kann es für $s > 1$ nichttriviale Linearkombinationen von Elementen aus T_{red} geben, die zu null vereinfachen.

Beispiel: $a = \sqrt{2} + \sqrt{3}, b = \sqrt{6}$. Es gilt $c = a^2 - 2\,b = 5$.

Die entsprechende Berechnung von $Q(x)$ für $c = a^2 - 2\,b$ mit MAXIMA aus den Minimalpolynomen $a^4 - 10\,a + 1$ und $b^2 - 6$ führt zu folgendem Ergebnis:

```
tellrat(a^4-10*a^2+1,b^2-6);
algebraic:true;
vars:makelist(concat(r,i),i,0,3);
f:sum(concat(r,i)*x^i,i,0,3);
p1:subst(x=a^2-2*b,f);
p2:expand(ratsimp(p1));
sys:flatten(makelist(makelist(coeff(coeff(p2,b,j),a,i),j,0,2),i,0,3));
sol:solve(append(sys,[r3=1]),vars);
p:subst(sol[1],f);
```

$$x^3 - 15\,x^2 - 21\,x + 355$$

Dieses Polynom ist allerdings nicht irreduzibel, sondern zerfällt in die Faktoren

$$(x^2 - 10\,x - 71)\,(x - 5).$$

c ist Nullstelle des zweiten Faktors, denn es gilt $c = 5$. Beide Polynome, $Q_1 = x^3 - 15\,x^2 - 21\,x + 355$ und $Q_2 = x - 5$, liefern dieselbe Inverse

$$c^{-1} = -\frac{1}{-5} = -\frac{1}{355}\left(c^2 - 15\,c - 21\right) = -\frac{-71}{355}.$$

Generell kann jedes Polynom $Q(x)$ mit $Q(c) = 0$ und nicht verschwindendem Absolutglied verwendet werden. Existiert so ein Polynom, so folgt zugleich $c \neq 0$. Wird nur ein Polynom $Q_0(x)$ gefunden, aus dem ein Faktor x abgespalten werden kann, so muss geprüft werden, ob vielleicht c eine nichttriviale Linearkombination aus T_{red} zu null ist. Das kann oft schon numerisch widerlegt werden. In diesem Fall hat Q_0 eine Darstellung $Q_0(x) = x^s\,Q_1(x)$ mit $Q_1(0) \neq 0$ und wegen $Q_1(c) = 0$ kann dieser Faktor verwendet werden. Für den Fall $c = 0$ ist c^{-1} natürlich nicht definiert.

Folgerung 3 *Sind $\alpha_1, \ldots, \alpha_s$ algebraische Zahlen über k vom Grad $d_1, \ldots, d_s$, so lässt sich jeder k-rationale Ausdruck $\frac{P(\alpha)}{Q(\alpha)} \in k(\alpha_1, \ldots, \alpha_s)$, für dessen Nenner $Q(\alpha) \neq 0$ gilt, als k-lineare Kombination von Elementen aus T_{red} darstellen.*

Natürlich ist es müßig, in jedem Fall erst das Minimalpolynom des jeweiligen Nenners zu bestimmen. Wir können stattdessen versuchen, diese Darstellung als k-lineare Kombination von Elementen aus T_{red} durch einen Ansatz mit unbestimmten Koeffizienten zu ermitteln.

Betrachten wir dazu den Ausdruck

$$A := \frac{\sqrt[3]{4} - 2\sqrt[3]{2} + 1}{\sqrt[3]{4} + 2\sqrt[3]{2} + 1} = \frac{a^2 - 2\,a + 1}{a^2 + 2\,a + 1}$$

mit $a = \sqrt[3]{2}$, d.h. $a^3 = 2$.

MAPLE liefert mit obiger Bibliothek, ebenso wie REDUCE und MAXIMA

```
s:subst(a=2^(1/3),(a^2-2*a+1)/(a^2+2*a+1));
ratsimp(s), algebraic:true;
```

$$\frac{2^{7/3}-5}{3}$$

Offensichtlich ist $(2^{2/3}-1)$ der „richtige“ Term, mit dem man A erweitern muss, um den Nenner in einen rationalen Ausdruck zu verwandeln.

Der Ansatz

$$A = \frac{a^2-2\,a+1}{a^2+2\,a+1} = a^2\,r_2 + a\,r_1 + r_0$$

mit unbestimmten Koeffizienten r_2, r_1, r_0 (ausgeführt wieder mit MAXIMA) liefert

```
tellrat(a^3-2);
algebraic:true;
u:(a^2*r2+a*r1+r0);
p2:expand(ratsimp(((a^2-2*a+1)-(a^2+2*a+1)*u)));
sys:makelist(coeff(p2,a,i),i,0,2);
sol:solve(sys,[r0,r1,r2]);
```

$$\left[\left[r_2 = 0, r_1 = \frac{4}{3}, r_0 = -\frac{5}{3}\right]\right]$$

und damit als Ergebnis unserer Vereinfachung

```
p:subst(sol[1],u);
```

$$\frac{4\,a-5}{3}$$

Für den Fall $s = 1$, also die Hinzunahme einer einzelnen algebraischen Zahl α, gilt der folgende Satz

Satz 12 *Ist α eine algebraische Zahl über k vom Grad d, so bildet die Menge $R := k[\alpha]$ der k-linearen Kombinationen von Termen aus $T_{red} := \left\{\alpha^i,\ i = 0, \ldots, d-1\right\}$ einen Körper.*

Kann man in k effektiv rechnen, so auch in R: Jeder rationale Ausdruck $A = \frac{P(\alpha)}{Q(\alpha)} \in k(\alpha)$ (mit $Q(\alpha) \neq 0$) kann eindeutig als k-lineare Kombination von Termen aus T_{red} dargestellt werden.

Ist das Minimalpolynom von α bekannt, so kann diese reduzierte Form effektiv berechnet werden, was auf eine kanonische Form in R, die algebraische Normalform*, führt, wenn wir eine kanonische Form in k voraussetzen.*

Zur Berechnung der reduzierten Form von A reicht es aus, ein d-dimensionales lineares Gleichungssystem mit Koeffizienten in k zu lösen.

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 reduzierten Form von Ausdrücken $A = \frac{P(\alpha)}{Q(\alpha)} \in R$, das wir oben an einem Beispiel demonstriert haben, stets zum Ziel führt. Es ist dazu das lineare Gleichungssystem in $r_0, \ldots, r_{d-1}$ zu lösen, das man aus

$$P(\alpha) = Q(\alpha) \cdot \left(\sum_{i=0}^{d-1} r_i \alpha^i\right)$$

nach (algebraischer) Ersetzung $\{\alpha^d \ \Rightarrow\ q(\alpha)\}$ durch Koeffizientenvergleich erhält.

Bei diesem Gleichungssystem handelt es sich um ein inhomogenes System von d linearen Gleichungen mit d Unbekannten. Dessen zugehöriges homogenes System

$$0 = Q(\alpha) \cdot \left(\sum_{i=0}^{d-1} r_i \alpha^i \right)$$

besitzt nur die triviale Lösung, da wegen $Q(\alpha) \neq 0$ jede nichttriviale Lösung zu einem Polynom $R(x) = \sum_{i=0}^{d-1} r_i x^i$ vom Grad $< d$ mit Nullstelle α führt.

Folglich hat (nach dem Lösungskriterium aus der linearen Algebra) das inhomogene System für jede rechte Seite genau eine Lösung, d.h. die Koeffizienten $r_0, \ldots, r_{d-1}$ lassen sich eindeutig bestimmen. □

Literaturverzeichnis

[1] B. Buchberger. Symbolisches Rechnen. In P. Rechenberg and H. Pomberger, editors, *Informatik-Handbuch*, chapter E5, pages 799 – 817. Hanser, München, 1997.

[2] B. Buchberger and R. Loos. Algebraic simplification. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra - Symbolic and Algebraic Computation*, pages 11–43. Springer, Wien, second edition, 1983.

[3] C.A. Cole and S. Wolfram. Smp – a symbolic manipulation program. In *Proc. SYMSAC*, pages 20 – 22, 1981.

[4] H. Engesser, editor. *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, E. Kaltofen, and V. Weispfenning, editors. *Computer Algebra Handbook. Foundations – Applications – Systems*. Springer, Berlin, 2003.

[8] U. Graf. *Applied Laplace Transforms and z-Transforms for Scientists and Engineers*. Birkhäuser, Basel, 2004.

[9] H.G. Kahrimanian. Analytic differentiation by a digital computer. Master's thesis, Temple Univ. Philadelphia, 1953.

[10] D.E. Knuth. *The art of computer programming*. Addison Wesley, 1991.

[11] R. Loos. Introduction. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra - Symbolic and Algebraic Computation*, pages 1–10. Springer, Wien, second edition, 1983.

[12] J. Nolan. Analytic differentiation on a digital computer. Master's thesis, Math. Dept., MIT, Cambridge, Mass., 1953.

[13] T. Ottmann and P. Widmeyer. *Algorithmen und Datenstrukturen*. BI Wissenschaftsverlag, 2 edition, 1993.

[14] H. Pieper. *Die komplexen Zahlen*. Dt. Verlag der Wissenschaften, Berlin, 1988.

[15] J.K. Prentice and M. Wester. Code generation using Computer Algebra Systems. In M. Wester, editor, *Computer Algebra Systems: A Practical Guide*, chapter 13, pages 233 – 254. Wiley, Chichester, 1999.

[16] A. Rich and D.R. Stoutemyer. Capabilities of the muMATH-79 computer algebra system for the INTEL-8080 microprocessor. In *Proc. EUROSAM*, pages 241 – 248, 1979.

[17] U. Schwardmann. *Computeralgebrasysteme*. Addison-Wesley, 1995.

[18] B. Simon. Comparative CAS review. *Notices AMS*, 39:700 – 710, sept 1992.

[19] D.R. Stoutemyer. PICOMATH-80, an even smaller computer algebra package. *SIGSAM Bull.*, 14.3:5–7, 1980.

[20] B. Stroustrup. *Die C++-Programmiersprache*. Addison-Wesley, 2 edition, 1992.

[21] J.A. van Hulzen and J. Calmet. Computer Algebra Systems. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra - Symbolic and Algebraic Computation*, pages 221 – 243. Springer, Wien, second edition, 1983.

[22] J. Weizenbaum. *Die Macht der Computer und die Ohnmacht der Vernunft*, volume 274 of *Taschenbuch Wissenschaft*. Suhrkamp, 9 edition, 1994.

[23] M. Wester. A review of CAS mathematical capabilities. *CAN Nieuwsbrief*, 13:41–48, dec 1994.

[24] M. Wester, editor. *Computer Algebra Systems: A Practical Guide*. Wiley, Chichester, 1999.

[25] N. Wirth. *Algorithmen und Datenstrukturen*. B.G. Teubner Verlag Stuttgart, 4 edition, 1986.

[26] S. Wolfram. *The Mathematica Book*. Wolfram Media and Cambridge University Press, 4 edition, 1999.

Anhang: Die Übungsaufgaben

1. a) Bestimmen Sie die Anzahl der Stellen der Dezimaldarstellung von $n!$ für $n \in \{10, 100, 1000\}$.
   b) Bestimmen Sie die Anzahl der Stellen der Darstellung von $n!$ im Binärsytem (Positionssystem zur Basis 2) für $n \in \{10, 100, 1000\}$.

2. Sei $a_n = 3^n - 2$.
   a) Für welche $n < 100$ ist a_n eine Primzahl ?
   b) Bestimmen Sie für $n < 40$ die Primfaktorzerlegung der Zahlen a_n.
   c) Leiten Sie aus den berechneten Zerlegungen allgemeine Vermutungen her, für welche n die a_n durch 5 und für welche n durch 7 teilbar ist.
   d) Beweisen Sie diese Vermutungen.

3. Untersuchen Sie, für welche $n < 30$ die Faktorzerlegung von $f(n) = n! - 1$ Primfaktoren mehrfach enthält.

4. Eine Zahl $2^p - 1$ ist höchstens dann eine Primzahl, wenn p selbst prim ist. Primzahlen dieser Form nennt man *Mersennesche Primzahlen*. Die größten bekannten Primzahlen haben diese Form. Es ist unbekannt, ob es unendlich viele Mersennesche Primzahlen gibt.

   Bestimmen Sie für $p < 100$ alle Mersenneschen Primzahlen. Geben Sie Ihr Ergebnis als Tabelle mit den Spalten p und $2^p - 1$ an.

5. Untersuchen Sie, auf wie viele Nullen die Zahl $N = 3^{100^{100}} - 1$ endet.
   a) Schätzen Sie die Zahl der Stellen von N ab.
   b) Überlegen Sie sich einen Zugang zur Aufgabe und stellen Sie eine Vermutung auf.
   c) Beweisen Sie Ihre Vermutung.
   d) Verallgemeinern Sie Ihre Aussage auf Zahlen $N_k = 3^{100^k} - 1$ und beweisen Sie diese Verallgemeinerung.

   Hinweis: Nicht alle CAS verstehen 3^{a^b} richtig als $3^{(a^b)}$ (denn $(3^a)^b$ ist ja $3^{a \cdot b}$ nach den Potenzgesetzen).

   Vorsicht außerdem, denn manche CAS hängen sich bei zu umfangreichen Rechnungen mit der Langzahlarithmetik auf und lassen sich auch nicht mehr über die Tastatur abbrechen.

6. Die Folge
$$s_1 := 1,\ s_{n+1} := \frac{1}{2}\left(s_n + \frac{2}{s_n}\right)$$
   konvergiert bekanntlich gegen $\sqrt{2}$.
   a) Bestimmen Sie die ersten 8 Werte der Folge als exakte Brüche.

b) Bestimmen Sie, wie viele Ziffern z_n die Zähler von s_n für $n = 1, \ldots, 12$ enthalten. Analysieren Sie die Wachstumsordnung der Folge z_n.

c) Bestimmen Sie die Konvergenzgeschwindigkeit der Folge s_n gegen den Grenzwert $\sqrt{2}$. (Beachten Sie, dass die Standardgenauigkeit Ihres CAS für numerische Rechnungen dafür möglicherweise nicht ausreicht)

Zur Bestimmung der *Wachstumsordnung* einer Folge a_n: Man unterscheidet zwischen polynomialem und exponentiellem Wachstum. Im ersten Fall wird als Wachstumsordnung eine Zahl α mit $a_n \sim n^\alpha$ bezeichnet, im zweiten eine Zahl α mit $\log(a_n) \sim n^\alpha$.

Als *Konvergenzgeschwindigkeit* einer Folge a_n gegen einen Grenzwert s bezeichnet man die größte Zahl α, so dass eine Konstante C mit $|a_{n+1} - s| < C\ |a_n - s|^\alpha$ für $n \gg 0$ existiert.

7. Zeigen Sie, dass eine ungerade perfekte Zahl wenigstens drei Primteiler haben muss. Ist sie nicht durch 3 teilbar, so müssen es sogar mindestens 7 Primteiler sein.

   Hinweis: Zeigen Sie zunächst, dass für eine perfekte Zahl $n = p_1^{a_1} \cdot \ldots \cdot p_m^{a_m}$ stets

   $$2 < \prod_{i=1}^{m} \frac{p_i}{p_i - 1}$$

   gelten muss.

8. Führen Sie für die Funktion $f(x) = \sin(x) - x \cdot \tan(x)$ eine Kurvendiskussion durch:

   a) Bestimmen Sie die Null- und Polstellen der Funktion $f(x)$ (notfalls näherungsweise).

   b) Zeigen Sie, dass die Funktion außerhalb des Intervalls $]-\frac{\pi}{2}, \frac{\pi}{2}[$ in ihren Stetigkeitsintervallen monoton ist.

   c) Untersuchen Sie das Verhalten der Funktion im Intervall $]-\frac{\pi}{2}, \frac{\pi}{2}[$.

   Geben Sie in jedem Fall **eine schlüssige mathematische Begründung** für Ihre Aussagen über den Verlauf der Funktion.

9. $a = e^{\pi\sqrt{163}}$ kommt einer ganzen Zahl b sehr nahe.

   a) Bestimmen Sie diese Zahl b und die Größenordnung o der Abweichung.

   b) Es gilt $\sqrt[3]{a} \sim 640\,320$ auf 9 Dezimalstellen genau. Finden Sie die ganze Zahl c, so dass die Approximation $\sqrt[3]{a+c} \sim 640\,320$ bestmöglich ist und geben Sie auch hier die Größenordnung der Abweichung an.

   Hinweis: Als *Größenordnung der Abweichung* zweier Zahlen a, b bezeichnet man die größte ganze Zahl o, für die $|a - b| < 10^{-o}$ gilt.

10. Bestimmen Sie die Wachstumsordnung d und -rate C der Funktion

    $$f(x) := \sin(\tan(x)) - \tan(\sin(x))$$

    in der Nähe von $x = 0$, d.h. solche Zahlen $C \in \mathbb{R},\ d \in \mathbb{N}$, dass $f(x) = C \cdot x^d + o\left(x^d\right)$ gilt.

    Warum ist eine numerische Lösung dieser Aufgabe nicht sinnvoll?

11. Der Ellipse $9\,x^2 + 16\,y^2 = 144$ soll ein flächenmäßig möglichst großes Rechteck einbeschrieben werden, dessen Seiten parallel zu den Koordinatenachsen liegen. Bestimmen Sie die Abmessungen dieses Rechtecks.

12. Erste Beweise, dass es unendlich viele Primzahlen gibt, gehen bis auf Euklid zurück. Dagegen kennt man bis heute noch keinen strengen Beweis dafür, dass es unendlich viele Primzahlzwillinge (p und $p+2$ sind prim) bzw. unendlich viele Germain-Primzahlen (p und $2p+1$ sind prim) gibt. Letztere spielten im 2002 gefundenen Beweis, dass es einen Primtestalgorithmus mit polynomialer Laufzeit gibt, eine Rolle.

    Gleichwohl zeigen numerische Experimente, dass es von beiden „relativ viele“ gibt. In der analytischen Zahlentheorie wird dazu das asymptotische Verhalten von Zählfunktionen wie

    $$\begin{aligned}\pi(x) &= |\{p \le x \,|\, p \text{ ist prim }\}| \\ t(x) &= |\{p \le x \,|\, p \text{ und } p+2 \text{ sind prim }\}| \\ g(x) &= |\{p \le x \,|\, p \text{ und } 2p+1 \text{ sind prim }\}|\end{aligned}$$

    untersucht, wobei $|\cdots|$ für die Anzahl der Elemente einer Menge steht. Für erste Vermutungen haben Zahlentheoretiker wie Gauss lange Listen von Primzahlen aufgestellt und ausgezählt. Dabei wurde festgestellt, dass für die Funktionen $\frac{\pi(x)}{x}$, $\frac{t(x)}{x}$ und $\frac{g(x)}{x}$ in erster Näherung $\sim C \cdot \ln(x)^a$ für verschiedene Konstanten C und Exponenten a zu gelten scheint.

    Erstellen Sie mit einem Computeralgebrasystem geeignetes experimentelles Zahlenmaterial bis wenigstens 10^6 und extrahieren Sie daraus plausible Werte für C und a für die drei angegebenen zahlentheoretischen Funktionen.

13. In der Vorlesung wurde der rationale Ausdruck

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

    betrachtet und festgestellt, dass sich dieser für kleine Werte $n \in \mathbb{N}$ zu einem Polynom in a, b, c vereinfachen lässt. Beweisen Sie diese Eigenschaft allgemein.

    a) Zeigen Sie die Gültigkeit der Rekursionsbeziehung

    $$u_n = \frac{b^{n-1} - c^{n-1}}{b-c} + a \cdot u_{n-1}\,.$$

    b) Leiten Sie daraus ab, dass u_n für jedes $n \in \mathbb{N}$ als polynomialer Ausdruck in a, b, c dargestellt werden kann.

    c) Zeigen Sie weiter, dass u_n für $n > 1$ mit der vollen symmetrischen Funktion h_{n-2} übereinstimmt, d. h. $u_n = h_{n-2}(a, b, c)$ gilt.

14. Mit dieser Aufgabe soll ein CAS als Problemlösungsumgebung eingesetzt werden. Wir wollen dazu die Frage studieren, ob es Fibonaccizahlen gibt, die mit vielen Neunen enden. Die Fibonaccizahlen sind bekanntlich durch die Rekursionsrelation

    $$F_0 = 0,\ F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ für } n > 1$$

    definiert.

    a) Finden Sie die erste Fibonaccizahl, die auf 9 endet.

    b) Finden Sie die erste Fibonaccizahl, die auf 99 endet.

    c) Untersuchen Sie, ob es Fibonaccizahlen gibt, die auf 99999 enden. Überlegen Sie sich dazu einen geeigneten Ansatz, mit dem die auszuführenden Rechnungen überschaubar bleiben.

    Erläutern Sie diesen Ansatz und geben Sie alle $n < 10^6$ an, für die F_n auf 99999 endet.

    d) Beweisen Sie, dass es Fibonaccizahlen gibt, die auf beliebig viele Neunen enden.

    Genauer: Zeigen Sie, dass es zu jedem $k \in \mathbb{N}$ eine Fibonaccizahl F_{n_k} gibt, die auf k Neunen endet.

Lösen sie die folgenden drei Abituraufgaben mit Hilfe eines CAS.

15. In einem kartesischen Koordinatensystem sind für jedes t ($t \in \mathbb{R}$, $t > 0$) die Punkte $A\,(6,0,0)$, $B_t\,(8,t^2,0)$, $C_t\,(4,3\,t,0)$ und $D\,(2,2,0)$ gegeben.

    Jedes Viereck AB_tC_tD ist die Grundfläche einer Pyramide mit der Spitze $S\,(5,3,6)$.

    a) Ermitteln Sie den Abstand des Punktes C_1 von der Ebene, in der die Seitenfläche AB_1S liegt.

    b) Berechnen Sie den Schnittwinkel zwischen dieser Seitenflächenebene und der Grundflächenebene.

    c) Zeigen Sie, dass es genau einen Wert t gibt, für den die zugehörige Pyramide eine quadratische Grundfläche besitzt, und bestimmen Sie diesen Wert.

    d) Berechnen Sie das Volumen dieser Pyramide mit quadratischer Grundfläche.

    (Quelle: Sächsisches Abitur 2001, Leistungskurs Mathematik)

16. Gegeben ist die Funktion $f(x) = x + \frac{1}{x}$ mit dem Definitionsbereich $\{x \in \mathbb{R}, x > 0\}$. In die Fläche zwischen Kurve und x-Achse ist ein Streifen mit der Breite 3 parallel zur y-Achse so einzufügen, dass seine Fläche möglichst klein wird.

    Berechnen Sie den Ort der beiden Parallelen und die resultierende minimale Fläche.

17. Der Kreis $x^2 + y^2 + 6y - 91 = 0$ und die Kurve $y = a\,x^2 + b$ schneiden einander im Punkt $P(6,y),\ y > 0$, unter einem Winkel von 90°.

    a) Berechnen Sie a und b.

    b) Die Schnittfläche der beiden Kurven rotiert um die y-Achse. Berechnen Sie das Volumen des entstehenden Rotationskörpers.

18. Als Kettenbruchzerlegung einer reellen Zahl x bezeichnet man eine Darstellung als

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots}}}}$$

    mit $a_0, a_1, \ldots \in \mathbb{N}$.

    a) Schreiben Sie eine Prozedur `CF(x,n)`, die eine Liste $\{a_0, a_1, \ldots, a_n\}$ der ersten n Kettenbrucheinträge zurückgibt und finden Sie die Kettenbruchzerlegungen `CF(x,50)` von $x_1 = \sqrt{2}$ und $x_2 = \frac{1}{2}\left(1 + \sqrt{5}\right)$. (*Hinweis:* Schreiben Sie eine eigene Prozedur und verwenden Sie dabei *nicht* eine ggf. vom CAS bereitgestellte Funktion, welche diese Aufgabe erfüllt.

    b) Leiten Sie daraus ab, wie sich x_1 und x_2 als *unendliche* periodische Kettenbrüche darstellen lassen, und beweisen Sie, dass diese Darstellung korrekt ist.

19. Gegeben ist die Funktionenschar $f_k, k \in \mathbb{R}$, welche durch $f_k(x) = e^{-x/2} + \frac{k}{x}$ für $x \neq 0$ definiert ist.

    a) Zeigen Sie, dass sich die Graphen zweier beliebiger Funktionen der Funktionenschar f_k nicht schneiden.

    b) Geben Sie für die Funktionen f_{-2} und f_2 jeweils Nullstellen, Koordinaten der lokalen Extrempunkte und die Art der Extrema an.

c) Leiten Sie aus b) eine Vermutung über die Existenz von lokalen Extrempunkten der Funktionen f_k für allgemeines k (in Abhängigkeit von k) ab und beweisen Sie Ihre Vermutung.

Analysieren Sie insbesondere den Fall $k < 0$ genau.

(Quelle: Sächsisches Abitur 2001, Leistungskurs Mathematik)

20. Geben Sie für die folgenden mathematischen Ausdrücke an, wie sie in Ihrem CAS erzeugen lassen, finden Sie die interne Darstellung des Ergebnisses heraus, erläutern Sie die semantische Struktur der Darstellung und benennen Sie Besonderheiten:

a) $x - y + z$ und $a/b * c$,

b) die Liste der Primzahlen bis 20,

c) die Matrix $\begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix}$,

d) `solve`$(x^2 + x + 1, x)$.

Mathematisch ist zwischen einer Funktion f, ihrem Funktionswert $f(2)$ an einer konkreten Stelle $x = 2$ und dem Funktionsausdruck $f(x)$ an der (symbolischen) Stelle x zu unterscheiden.

e) Untersuchen Sie, ob man mit dem von Ihnen verwendeten CAS zwischen Funktionen f als Symbol und Funktionsausdrücken $f(x)$ unterscheiden kann.

Bestimmen Sie die Ableitung des Funktionsausdrucks $\arctan(x)$ sowie die Ableitung der Funktion arctan und finden Sie in beiden Fällen die interne Darstellung des Ergebnisses heraus.

21. In dieser Aufgaben soll der `sum`-Befehl Ihres CAS untersucht werden, mit dem auch symbolische Summen vereinfacht werden können.

a) Wie kann in Ihrem CAS ein geschlossener Ausdruck für $\sum_{i=1}^{n} i^2$ bestimmt werden? Geben Sie diesen Ausdruck an.

Es sei $S_n := \sum_{k=1}^{n} \frac{1}{k}$.

b) Geben Sie an, was zur Berechnung von S_{100} einzugeben ist und bestimmen Sie die Struktur des ausgegebenen Ausdrucks.

c) Geben Sie an, was zur Berechnung von S_n einzugeben ist und bestimmen Sie die Struktur des ausgegebenen Ausdrucks.

d) Speichern Sie den Wert von S_n in einer Variablen u. Geben Sie an, wie in Ihrem CAS $n = 100$ in u substituiert werden kann und bestimmen Sie die Struktur des so entstehenden Ausdrucks.

Welcher (minimale) Aufwand ist erforderlich, um dieses Ergebnis in das Ergebnis der Aufgabe a) umzuformen?

22. a) Lösen Sie in Ihrem CAS das Gleichungssystem

$$\left\{x^2 + y^2 = 4,\ x + y = 1\right\}.$$

Speichern Sie dazu das System in einer Variablen `sys` ab und verwenden Sie einen adäquaten `solve`-Befehl. Die gefundene Lösung soll explizit sein, keine `RootOf`-Symbole enthalten und in einer Variablen `sol` abgespeichert werden.

b) Wie kann mit Ihrem CAS allein unter Verwendung der Map-Funktion und des Substitutionsoperators sowie der Werte von `sys` und `sol` die Korrektheit der Antwort durch eine Probe geprüft werden?

Zur Lösung der folgenden Aufgaben sollen das Konzept der Substitutionslisten und die in der Vorlesung vorgestellten Listenoperationen verwendet werden.

23. Stellen Sie die x-Werte, an denen die Funktion

$$g(x) = 12 - 24x + 22x^2 - 8x^3 + x^4$$

lokale Extrema hat, in einer Liste l zusammen und erzeugen Sie daraus die Liste der Extremwertkoordinaten (x_i, y_i) der lokalen Extrema von $g(x)$.

24. Aus den Polynomen $f_1 = x^2 + y^2$, $f_2 = x^3 - 3\,x\,y^2$, $f_3 = y^3 - 3\,x^2\,y$ lassen sich die vier Produkte $f_1^3, f_2^2, f_2\,f_3, f_3^2$ vom Grad 6 bilden.

Bestimmen Sie eine lineare Abhängigkeitsrelation $a_1\,f_1^3 + a_2\,f_2^2 + a_3\,f_2\,f_3 + a_4\,f_3^2 = 0$ zwischen diesen Produkten und zeigen Sie mit einer Probe die Korrektheit Ihrer Antwort.

Starten Sie dazu mit der Substitutionsliste

```
sys:=[f1=x^2+y^2, f2=x^3-3*x*y^2, f3=y^3-3*x^2*y]
```

und verwenden Sie in Ihren Rechnungen die Bezeichner `f1 f2 f3 x y` ausschließlich im Symbolmodus.

25. Zur Glättung von Daten $l_1 = ((x_1, y_1), \ldots, (x_n, y_n))$ kann man die Liste der gleitenden k-Durchschnitte, d.h. $l_k = \left(\left(\frac{1}{k}\sum_{j=i}^{i+k-1} x_j, \frac{1}{k}\sum_{j=i}^{i+k-1} y_j\right), 1 \le i \le n-k+1\right)$ berechnen.

a) Schreiben Sie für ein CAS Ihrer Wahl eine Funktion `glD(l,k)`, die zur Liste $l = l_1$ die Liste l_k erzeugt.

b) Erzeugen Sie eine Liste l mit 101 Datenpunkten, deren x-Werte das Intervall $0 \le x \le 2$ in gleiche Teile teilen und deren y-Werte um den Graphen der Funktion $y = x^2$ zufällig streuen, und berechnen Sie dazu die geglättete Liste l_7=`glD(l,7)`.

c) Stellen Sie die Listen sowie $y = x^2$ im angegebenen Intervall als Punkte der Ebene in zwei Bildern grafisch so dar, dass der Glättungseffekt sichtbar wird.

26. Zerlegen Sie das Polynom $x^4 + 1$ in Faktoren

a) über den ganzen Zahlen,

b) über dem Restklassenkörper $\mathbb{Z}_p$ für Primzahlen $p \le 30$.

und prüfen Sie das Ergebnis jeweils durch Ausmultiplizieren.

27. Zeigen Sie, dass sich $x^4 + 1$ für jede Primzahl p über $\mathbb{Z}_p$ in quadratische Faktoren zerlegen lässt.

28. Finden Sie durch geeignete Anwendung der Listenoperationen `map`, `subs` und `select` alle Lösungen des Gleichungssystems

$$\{x^3 + y = 2,\ y^3 + x = 2\},$$

für die $x \neq y$ gilt.

Verwenden Sie dazu die `solve`-Funktion Ihres CAS, organisieren Sie deren Ausgabe (ggf.) in eine Liste, ersetzen Sie „suspekte“ Terme durch numerische Näherungswerte – oder verwenden Sie gleich ein numerisches `Solve` – und wählen Sie dann diejenigen Terme aus, für die $x \neq y$ gilt. Das sind 6 der 9 (komplexen) Lösungen des gegebenen Gleichungssystems.

Diskutieren Sie das Antwortverhalten des von Ihnen verwendeten CAS.

29. Die meisten CAS kennen exakte Werte von Winkelfunktionen mit dem Argument $\frac{m}{n}\,\pi$ und $n \leq 6$.

    Bestimmen Sie daraus exakte Werte von $\sin(15°)$ sowie von $\sin(6°)$ und erläutern Sie Ihr Vorgehen.

    Hinweis: Wegen $6° = \frac{\pi}{30}$ gilt $\sin(6°) = \sin(\frac{\pi}{5} - \frac{\pi}{6})$. Eine ähnliche Beziehung gilt für $15°$.

30. Lösen Sie mit einem CAS die folgenden Gleichungen und geben Sie die Lösungen im Intervall $[-\pi, \pi]$ exakt als möglichst einfache Formel sowie näherungsweise in Grad an.

    Begründen Sie, dass die jeweiligen Lösungsmengen vollständig sind.

    a) $\sin(x)\,\sin(3\,x) = \frac{1}{2}$
    b) $\sin(x)\,\sin(2\,x)\,\sin(3\,x) = \frac{1}{4}\,\sin(4\,x)$
    c) $\sin(2\,x) + \cos(3\,x) = 1$.

31. Falten Sie ein A4-Blatt ($x = \sqrt{2}$ LE – Länge der längeren Seite, 1 LE – Länge der kürzeren Seite) auf folgende Weise:

    – Zuerst so, dass zwei gegenüberliegende Ecken des A4-Blatts aufeinander zu liegen kommen (es entsteht ein sehr breites Fünfeck mit zwei kurzen und drei langen Seiten).
    – Nun dessen „Flügel“ so, dass die kurzen Seiten genau auf der Symmetrieachse dieser Figur zu liegen kommen.

    Sie erhalten den Umriss eines „sehr regelmäßigen“ Fünfecks.

    a) Untersuchen Sie, ob es sich wirklich um ein regelmäßiges Fünfeck handelt.
       Bestimmen Sie exakte Werte für die Seitenlängen dieses Fünfecks in LE.
       Welche der Fünfecksseiten sind gleichlang?
    b) Bestimmen Sie das Verhältnis x der längeren zur kürzeren Seite des Ausgangsrechtecks, für welches das gefaltete Fünfeck regelmäßig ist.
       Geben Sie einen exakten Wurzelausdruck für x an.
    c) Bestimmen Sie exakte Formeln für die Seitenlängen des Fünfecks, wenn die längere Seite des Ausgangsrechtecks x LE und die kürzere 1 LE lang ist. Geben Sie auch hier exakte Wurzelausdrücke an.

Beim Vereinfachen von geschachtelten Wurzelausdrücken zeigen sich CAS oft unerwartet schwerfällig. Um so überraschender mag es sein, dass Vereinfachungen wie

$$\begin{aligned}\sqrt{11 + 6\,\sqrt{2}} + \sqrt{11 - 6\,\sqrt{2}} &= 6\\ \sqrt{5 + 2\,\sqrt{6}} + \sqrt{5 - 2\,\sqrt{6}} &= 2\sqrt{3}\\ \sqrt{5 + 2\,\sqrt{6}} - \sqrt{5 - 2\,\sqrt{6}} &= 2\sqrt{2}\end{aligned}$$

teilweise automatisch ausgeführt werden.

32. Durch Quadrieren überzeugt man sich leicht von der Beziehung

$$\sqrt{7 + \sqrt{40}} = \sqrt{2} + \sqrt{5}.$$

    a) Finden Sie ein konstruktives Kriterium, nach dem sich für vorgegebene $a, b \in \mathbb{N}$ (b kein volles Quadrat) entscheiden lässt, ob der Ausdruck $\sqrt{a + \sqrt{b}}$ zu einem Ausdruck der Form $\sqrt{c} + \sqrt{d}$ mit geeigneten $c, d \in \mathbb{N}$ vereinfacht werden kann. Geben Sie Ihre Antwort in Form einer Regel

```
Rule(sqrt(a+sqrt(b)), A(a,b), B(a,b))(a,b)
```

an, wobei a, b formale Parameter sind, $A(a,b)$ der zu substituierende Ausdruck und $B(a,b)$ die Bedingung angibt, unter welcher die Ersetzung ausgeführt werden darf.

b) Zeigen Sie, dass das von Ihnen gefundene Kriterium auch notwendig ist, d. h. alle Fälle erfasst, wo Simplifikation möglich ist.

c) Geben Sie zur Demonstration für drei nicht triviale Zahlenbeispiele diese Vereinfachung jeweils an.

Diskutieren Sie, welche Probleme sich bei der Implementierung dieser Vereinfachungsregel in einem CAS ergeben.

33. `abl` sei ein neues Funktionssymbol, das semantisch für die Ableitungsfunktion steht, d. h. `abl(f,x)` soll die Ableitung des Ausdrucks f nach der Variablen x berechnen.

a) Stellen Sie ein Regelsystem `ablrules` auf, mit dem sich die Ableitung polynomialer Ausdrücke in expandierter Form korrekt berechnen lässt und demonstrieren Sie die Wirkung Ihres Regelsystems am Ausdruck $f = x^3 + 5\,x^2 + 7\,x + 4$.

b) Erweitern Sie das Regelsystem so, dass sich auch Ableitungen polynomialer Ausdrücke in faktorisierter Form korrekt berechnen lassen und demonstrieren Sie die Wirkung Ihres Regelsystems am Ausdruck $f = (x^2 + x + 1)^3\,(x^2 - x + 1)$.

c)* Erweitern Sie das Regelsystem so, dass sich auch Ableitungen rationaler Ausdrücke korrekt berechnen lassen und demonstrieren Sie die Wirkung Ihres Regelsystems am Ausdruck $f = (x^2 + x + 1)^3/(x^2 - x + 1)$.

Diese Aufgabe sollte mit einem CAS berabeitet werden, das Definition und Anwendung von Regelsystemen erlaubt. Alternativ können Sie eine Transformationsfunktion `abl` mit den angegebenen Eigenschaften implementieren.

Die Ergebnisse in b) und c) sind jeweils in der rationalen Normalform anzugeben.

34. Für $n \geq m \geq 0$ und eine Variable q bezeichnet man

$$\begin{bmatrix} n \\ m \end{bmatrix}_q := \frac{[n]_q}{[m]_q[n-m]_q} \qquad \text{mit} \qquad [n]_q := \prod_{i=1}^{n} (1 - q^i)$$

als *q-Binomialkoeffizienten*.

a) Untersuchen Sie für $0 \leq m \leq n \leq 10$, ob sich $\begin{bmatrix} n \\ m \end{bmatrix}_q$ zu einem polynomialen Ausdruck vereinfachen lässt.

Welche Simplifikation ist der Aufgabenstellung angemessen?

b) Verwenden Sie Ihre Berechnungen, um eine Vermutung für den Wert von

$$\lim_{q \to 1} \begin{bmatrix} n \\ m \end{bmatrix}_q$$

aufzustellen.

c) Beweisen Sie Ihre Vermutung aus b).

35. $T_n(x) := \cos(n \cdot \arccos(x))$ lässt sich zu einem polynomialen Ausdruck vereinfachen, den *Tschebyschewpolynomen* (erster Art).

a) Finden Sie die ersten 10 dieser Polynome und überprüfen Sie an ihnen die Beziehung

$$T_{n+m}(x) + T_{n-m}(x) = 2\,T_n(x)\,T_m(x)$$

b) Beweisen Sie diese Beziehung.

Zusatz: Untersuchen Sie, für welche n das Polynom $T_n(x)$ irreduzibel ist und stellen Sie dazu einen Theoriebezug her.

36. Beweisen Sie CAS-gestützt die Beziehung

$$\cos\left(\frac{\pi}{7}\right) - \cos\left(\frac{2\,\pi}{7}\right) + \cos\left(\frac{3\,\pi}{7}\right) = \frac{1}{2}\,,$$

indem Sie den Ausdruck auf der linken Seite so umformen, dass er polynomial in nur noch einem Kern ist. In der Beweisführung muss deutlich werden, welche mathematischen Zusammenhänge verwendet wurden. Ein einfaches Anwenden etwa von `FullSimplify` reicht nicht aus.

Hinweis: Ersetzen Sie $\frac{\pi}{7} = x$ und vergleichen Sie Ihr Ergebnis mit dem expandierten Ausdruck von $\cos(7\,x) + 1$. Beachten Sie dabei, dass $\cos(7\,\frac{\pi}{7}) + 1 = 0$ gilt.

Bonusserie: Bei der Berechnung des für die Scheinvergabe zu erreichenden Satzes von 50 % der Punkte aus den Übungsserien gehen die in dieser Serie erreichten Punkte in den Zähler, die erreichbaren Punkte aber nicht in den Nenner ein.

37. Zur schnellen Berechnung von π auf viele Stellen benötigt man gut konvergierende Reihen. Eine davon ist

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots$$

mit möglichst kleinem Argument.

$$\frac{\pi}{4} = \arctan(1) = 1 - \frac{1}{3} + \frac{1}{5} - \ldots$$

konvergiert zwar, aber viel zu langsam. Im Buch

> π - *Algorithmen, Computer, Arithmetik*
> von J. Arndt und C. Hänel (Springer-Verlag, 2000)

werden auf S. 77 ff. eine Reihe anderer Ausdrücke mit besserem Konvergenzverhalten genannt, mit denen $\frac{\pi}{4}$ berechnet werden kann:

$$\begin{aligned}
u_1 &= \arctan(1/2) + \arctan(1/3)\\
u_2 &= 4\,\arctan(1/5) - \arctan(1/239)\\
u_3 &= 8\,\arctan(1/10) - 4\,\arctan(1/515) - \arctan(1/239)\\
u_4 &= 12\,\arctan(1/18) + 8\,\arctan(1/57) - 5\,\arctan(1/239)\\
u_5 &= 22\,\arctan(1/28) + 2\,\arctan(1/443) - 5\,\arctan(1/1393) - 10\,\arctan(1/11018)\\
u_6 &= 44\,\arctan(1/57) + 7\,\arctan(1/239) - 12\,\arctan(1/682) + 24\,\arctan(1/12943)
\end{aligned}$$

Überlegen und beschreiben Sie ein Verfahren, wie sich überprüfen lässt, ob solche Ausdrücke exakt gleich $\frac{\pi}{4}$ sind und überprüfen Sie jeden der angegebenen Ausdrücke mit Ihrem Verfahren.

38. Eine interessante Formel verbindet die Zahl π mit dem Goldenen Schnitt $\phi = \frac{1+\sqrt{5}}{2}$:

$$\pi = 4\,\left(\arctan\left(\frac{1}{\phi}\right) + \arctan\left(\frac{1}{\phi^3}\right)\right)\,.$$

Beweisen Sie diese Formel.

39. Beweisen Sie CAS-gestützt die Beziehung

$$\tan\left(\frac{3}{11}\,\pi\right) + 4\,\sin\left(\frac{2}{11}\,\pi\right) = \sqrt{11}$$

Finden Sie dazu einen polynomialen Ausdruck in einem einzigen Kern, dessen Verschwinden zur Fragestellung äquivalent ist, und zeigen Sie, dass der Ausdruck für diesen Kern wirklich verschwindet, d.h. für den Kern eine weitere algebraische Beziehung gilt, die bei der Berechnung der polynomialen Normalform nicht berücksichtigt wurde.

Geben Sie ein Regelsystem an, mit dem sich die erforderlichen Transformationen realisieren lassen.