Skript zum Kurs
Einführung in das symbolische Rechnen
Wintersemester 2008/09

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

15. Januar 2009

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. Der souveräne Umgang mit solchen Systemen gehört 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 neuen sächsischen Lehrplänen für das Gymnasium ab Klasse 8 geschehen) und sie im universitären Studium als wichtige Hilfsmittel einzusetzen. Eine solche gewisse Vertrautheit im Umgang mit CAS werde ich auch 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 an unserer Universität weit verbreiteten Systeme Maple, MuPAD und Mathematica 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 mir auch deshalb sowohl gerechtfertigt als auch wünschenswert, weil es mittlerweile für jedes der großen Systeme eine Fülle von einführender Literatur höchst unterschiedlicher Qualität und verschiedenen Anspruchsniveaus gibt. Mehr noch, die Entwicklerteams der großen Systeme haben in den letzten Jahren viel Energie darauf verwendet, ihre Produkte auch von der äußeren Gestalt her nach modernen softwaretechnischen und -ergonomischen Gesichts-

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

Für den Nutzer von Computeralgebrasystemen wird es andererseits zunehmend schwieriger, die wirkliche Leistungsfähigkeit der Systeme hinter ihrer gleichermaßen glitzernden Fassade zu erkennen. Auch hierfür soll dieser Kurs ein gewisses Testmaterial zur Verfügung stellen, obwohl bei einem globalen Vergleich der Leistungsfähigkeit der verschiedenen Systeme durchaus Vorsicht geboten ist. Unterschiedliche Herangehensweisen im Design zusammen mit der jeweils spezifischen Entstehungsgeschichte führten dazu, dass die Leistungsfähigkeit unterschiedlicher Systeme in unterschiedlichen Bereichen sehr differiert und, mehr noch, sich verschiedene „mathematische Rigorosa“ wie etwa in einem Teil der Wester-Liste [23, ch. 3] oder in Bernardins Übersicht [23, ch. 7] enthalten, mit sehr unterschiedlichem Aufwand im Nachhinein integrieren lassen. Diese Feinheiten führen an einigen Stellen zu unterschiedlichem Verhalten der verschiedenen Systeme bis hin zu sehr unterschiedlichen Antworten. Wir werden uns dabei auf die Frage beschränken, wie konsistent die einzelnen Systeme ihre eigenen Konzepte verfolgen, da gerade Übersichten 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 [23, 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[2].

Allerdings begegnet man diesem „Zauberlehrlingseffekt“, den Weizenbaum in seinem inzwischen klassischen Werk [21] 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 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.

---

[1] 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 aussieht. 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 (Stichwort: LINUX-Distributionen und freie Software) ebenfalls stattfinden.

[2] 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.“

Die Voraussetzungen

Symbolische Rechnungen sind das wohl grundlegendste methodische Handwerkszeug in der Mathematik und durchdringen alle ihrer 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 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“.

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 nachvollziehbar sind. Dies verlangt, eine wohlüberlegte Auswahl zu treffen.

Im folgenden werden wir uns deshalb zunächst auf die Darstellung wichtiger Designaspekte eines CAS der zweiten Generation konzentrieren, daran anschließend theoretische und praktische Aspekte der Simplifikationsproblematik als besonderem Charakteristikum des symbolischen Rechnens diskutieren und schließlich am Beispiel der symbolischen Behandlung algebraischer Zahlen diese Konzepte in ihrem komplexen Zusammenspiel erleben, Anforderungen und Lösungen gegenüberstellen und dabei einige auf den ersten Blick merkwürdige Ansätze besser verstehen. Im letzten Teil des Kurses – und darin unterscheidet sich die Anordnung des Materials gegenüber den Vorjahren – sollen schließlich übergreifende Aspekte eher philosophischer Natur berührt werden, um die Stellung des symbolischen Rechnens als einer zentralen technologischen Entwicklungslinie im Wissenschaftsgebäude zu verstehen.

Nicht berühren werden wir das Feld der *Differentialgleichungen*, da die Schwierigkeiten der damit verbundenen Mathematik in keinem Verhältnis zu den dabei gewinnbaren neuen Einsichten in Zusammenhänge des symbolischen Rechnens steht. 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.

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 MuPAD 2.5, hätten aber mit jedem der großen CAS in ähnlicher Weise ausgeführt werden können.

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!;
```

93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000

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);
```

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

```
float(%);
```

$$4.499205338$$

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

```
sqrt(2);
```

$$\sqrt{2}$$

```
PI;
```

$$\pi$$

```
sin(PI);
```

$$0$$

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

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

und sogar

```
sin(PI/5);
```

$$\frac{\sqrt{2}\sqrt{-\sqrt{5}+5}}{4}$$

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

```
DIGITS:=20:
p:=float(PI); sin(p);
delete DIGITS:
```

$$3.1415926535897932384$$
$$1.0097419586828951109\,10^{-28}$$

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 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);
```

$$11$$

```
ifactor(12!);
```

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

```
isprime(12!-1);
```

$$TRUE$$

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));
```

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

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

Das ermöglicht die Funktion `normal`. Das Ergebnis mag verblüffen; jedenfalls sieht man das dem ursprünglichen Ausdruck nicht ohne weiteres an.

```
normal(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);
```

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

```
normal(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 Funktion `subs`, die lokal eine Variable durch einen anderen Ausdruck, in diesem Fall eine Zahl, ersetzt.

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

$$1$$

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

$$a + b + c$$

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

$$ab + ac + bc + a^2 + b^2 + c^2$$

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

$$abc + a^3 + b^3 + c^3 + ab^2 + a^2b + ac^2 + a^2c + bc^2 + b^2c$$

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

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, wie erwartet vereinfacht.

```
(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);
```

$$1 + \mathrm{i}$$

Oftmals müssen solche Umformungen durch entsprechende Funktionsaufrufe gezielt angestoßen werden.

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

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

```
expand((sqrt(2)+sqrt(3))^n) $ n=2..5;
```

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

Manchmal ist es nicht einfach, das System zu „überreden“, genau das zu tun, was man will.

So liefern weder `expand` noch `normal` oder `simplify` eine Form von $(\sqrt{2} + \sqrt{3})^{-1}$ mit rationalem Nenner. Erst die offensichtlich aufwändige Funktion (Laufzeit 1.4 s.) liefert das erwartete Ergebnis, das man auch schnell im Kopf ausrechnen kann.

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

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

Der Grund liegt darin, dass es aus Effizienzgründen nicht klug ist, Nenner rational zu machen. Wir kommen auf diese Frage später zurück.

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 betrachteten Systeme (nur Derive macht hier bisher eine Ausnahme und orientiert sich stärker an einer Drag-and-Drop-Philosophie). 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 von MuPAD, wenn er die dem Computer „angemessene“ Frage nach letzten Ziffern der Zahl $2^{10^{10}}$ stellt. Zunächst ist zu beachten, dass MuPAD `2^10^10` als 2^{100} berechnet, der Operator ^ also linksassoziativ statt wie sonst üblich rechtsassoziativ wirkt[1]. Bei der korrekten Eingabe `2^(10^10)` gibt MuPAD nach endlicher Zeit auf mit der Information

```
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 unserem 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[2], können wir nach Regelmäßigkeiten für die Potenzreste für beliebige Moduln fragen.

```
2^k mod 10 $ k=1..15;
```

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

```
2^k mod 3 $ k=1..15;
```

$$2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2$$

```
2^k mod 11 $ k=1..15;
```

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

```
2^k mod 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.

```
2^(10^10) mod 10000;
```

Wir brauchen statt dessen eine spezielle Potenzfunktion, die bereits in den Zwischenrechnungen die Reste reduziert und damit Zwischenergebnisse überschaubarer Größe produziert.

[1] Zum Vergleich: Maple lässt solche Ausdrücke gar nicht zu, Mathematica und Maxima folgen der allgemeinen – und mit Blick auf die Potenzgesetze auch sinnvollen – Konvention, Reduce rechnet wie MuPAD.

[2] Notwendige Nebenbemerkung für den Einsatz in der Schule: Nachdem dieser Gegenstand genügend geübt worden ist.

MuPAD verfolgt einen objektorientierten Ansatz, der funktionale Polymorphie (hier der Potenzfunktion) an Hand der Argumenttypen auflösen kann. Die korrekte Potenzfunktion wird also ausgewählt, wenn bereits die Zahl 2 als Restklasse erzeugt wird.

```
R:=Dom::IntegerMod(10^20);
R(2)^(10^10);
```

$$4637454968178710937 6 \mod 10000000000000000000$$

In Maple erreichen wir denselben Effekt über den *inerten Operator* `&^` (auf solche inerten Funktionen wird später noch einzugehen sein).

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

$$46374549681787109376$$

In Mathematica müssen wir die spezielle Funktion `PowerMod` verwenden[3].

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

$$46374549681787109376$$

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. Jh. vor Christi werden solche Zahlen betrachtet. Euklid kannte eine Formel, nach der man alle gerade perfekte Zahlen findet, die erstmals von Euler exakt bewiesen wurde.

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

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

$$6$$
$$28$$
$$496$$

Betrachten wir die gefundenen Zahlen näher:

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

Alle diese Zahlen haben die Gestalt $2^{k-1}(2^k - 1)$. Wir wollen deshalb versuchen, perfekte Zahlen dieser Gestalt zu finden. Zur Ausgabeformatierung als Tabelle verwenden wir die MuPAD-Funktion `format` aus dem `stringlib`-Paket.

```
Ausgabe:=proc(k,n,Antwort)
begin
  print(Unquoted,stringlib::format("".k,3,Right)
       .stringlib::format("".n,35,Right)
       .stringlib::format(Antwort,10,Right))
end_proc:
```

[3] Eine solche Funktion `powermod` gibt es auch in MuPAD.

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

k	n	perfekt ?
2	6	yes
3	28	yes
4	120	no
5	496	yes
6	2016	no
7	8128	yes
8	32640	no
9	130816	no
10	523776	no
11	2096128	no
12	8386560	no
13	33550336	yes
14	134209536	no
15	536854528	no
16	2147450880	no
17	8589869056	yes
18	34359607296	no
19	137438691328	yes
20	549755289600	no
...		

Die Ausgabe ist als Tabelle zusammengestellt und wegen ihrer Länge gekürzt. Sie 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* $P := 2^k - 1$ *eine Primzahl sein muss.*

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

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 MUPAD:

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

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

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);
```

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

```
u:=int(p,x);
```

$$\ln\left(x^2-x+1\right)-\frac{\ln\left(\left(x+1/2\right)^2+3/4\right)}{2}-\frac{\sqrt{3}\arctan\left(\frac{2\sqrt{3}(x+1/2)}{3}\right)}{3}$$

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

$$\frac{(2x+-1)}{(x^2-x+1)}-\frac{2}{3\left(\frac{4(x+1/2)^2}{3}+1\right)}-\frac{(2x+1)}{2\left(\left(x+1/2\right)^2+3/4\right)}$$

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

$$0$$

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

$$\frac{\ln\left(x-1\right)}{4}-\frac{\arctan\left(x\right)}{2}-\frac{\ln\left(x+1\right)}{4}$$

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 bündelt der `solve`-Operator von MuPAD Wissen über Algorithmen zum Auffinden der Nullstellen univariater Polynome, von linearen und polynomialen Gleichungssystemen und selbst Differentialgleichungen können damit gelöst werden.

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

$$\left\{\left[y=\frac{1}{2},x=\frac{3}{2}\right]\right\}$$

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

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

Selbst für kompliziertere Gleichungen, die trigonometrische Funktionen enthalten, werden Lösungen in mittlerweile schöner 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 nun auch gut 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 antwortet ähnlich, aber bis zur Version 5 gab es keine Möglichkeit, die Lösungsschar in parametrisierter Form auszugeben.

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

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

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

Im Handbuch der Version 3 von Mathematica (S. 794) wurde dies wie folgt begründet:

> *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.

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

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

Die Mathematica-Entwickler stützen sich dabei auf eine andere Phiolosophie 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.

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

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

Aus der gegebenenfalls durch `Solve` die Lösungen in der traditionellen Notation extrahiert werden können.

```
Solve[s,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 des selben 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 $\cos(x) = u \;\Rightarrow\; x = \pm\arccos(u) + 2\,k\,\pi$)

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

MuPADs Antwort lautet

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

$$\left\{ 2\,\arctan\left(\frac{2}{3} - \frac{\sqrt{7}}{3}\right) + 2\,k\,\pi \,|\, k \in \mathbb{Z} \right\} \bigcup \left\{ 2\,\arctan\left(\frac{\sqrt{7}}{3} + \frac{2}{3}\right) + 2\,k\,\pi \,|\, k \in \mathbb{Z} \right\}$$

während Maple folgende Antwort liefert:

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

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

MATHEMATICA schließlich:

```
s=Reduce[Sin[x]+Cos[x]==1/2,{x}]
```

$$c_1 \in \mathbb{Z} \wedge \left(x == 2\pi c_1 + 2\arctan\left(\frac{1}{3}\left(2-\sqrt{7}\right)\right) \vee x == 2\pi c_1 + 2\arctan\left(\frac{1}{3}\left(2+\sqrt{7}\right)\right)\right)$$

> 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:

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

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

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

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

```
s//N (* Mathematica *)
```

$$\{\{x \to 1.99483\}, \{x \to -0.424031\}\}$$

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 das System MuPAD 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 MuPAD lauten (ohne Typprüfungen der Eingabeparameter)

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

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

```
p:=proc(d,vars) begin _plus(vars[i]^d $ i=1..nops(vars)) end_proc;
```

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

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

$$xy + xz + yz + x^2 + y^2 + z^2$$

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

$$xyz$$

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

$$xyz + x^3 + y^3 + z^3 + xy^2 + x^2y + xz^2 + x^2z + yz^2 + y^2z$$

1.6 Numerisches versus symbolisches Rechnen

Wir wollen auf der Basis der bisher untersuchten Anwendungen einen ersten Vergleich zwischen numerischen und symbolischen Ansätzen ziehen, denn hier stoßen zwei paradigmatisch unterschiedliche Welten aufeinander: Während die symbolischen Methoden der Computeralgebra – wenigstens im Prinzip – streng mathematisch deduktiver Natur sind, haben numerische Verfahren meist approximativen Charakter und sind damit zunächst auf ein quantitatives Bild der Realität ausgerichtet.

Eine solcher Vergleich ist auch deshalb interessant, weil die Erfahrungen, über welche die meisten Leser dieses Skripts mit automatischen Rechnungen auf dem Computer verfügen, gerade aus dem Gebiet der Numerik kommen. Auch der größte Teil der Rechnungen, die heutzutage auf großen Mainframes und Rechnerverbänden ausgeführt werden, sind numerischer Natur und erfahren bestenfalls noch eine Ergänzung durch Visualisierungstools, die sich auf eben solche Grundlagen stützen.

Den typischen Unterschied zwischen beiden Sichtweisen beschreibt Pavelle in [13] wie folgt:

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

Der Verlust struktureller Information, der im Rahmen numerischer Verfahren durch die Abbildung der Überabzählbarkeit der reellen Welt auf die Endlichkeit des Computers unvermeidlich ist, führt dazu, dass mit diesen Daten streng deduktiv basierte, also dem Rationalitätsanspruch einer „reinen“ Wissenschaft genügende Argumentationen nicht mehr möglich sind. Symbolische Verfahren können dagegen ob der Endlichkeit ihrer Begriffswelt adäquat in die Endlichkeit eines Computers übertragen werden, ohne dadurch Teile ihrer strukturellen Aussagekraft einzubüßen.

Numerische Verfahren kann man einsetzen, um praktisch relevante Zahlenwerte *auszurechnen* (und mit geeigneten Visualisierungswerkzeugen darzustellen), symbolische Verfahren dagegen auch, um strukturelle mathematische Aussagen *zu beweisen*.

Numerische Verfahren gestatten es, eine große Zahl von Daten zu produzieren, in denen sich ein gewisser struktureller Zusammenhang widerspiegelt. Auf die *Allgemeingültigkeit* eines solchen Zusammenhangs (im Sinne wissenschaftlicher Rationalität) kann man aber aus noch so vielen Daten nicht schließen. Solche Datenmengen eignen sich auch nur beschränkt für die (nicht sensorische) Bearbeitung inverser Fragestellungen, die beim Steuern und Regeln von Prozessen auftreten.

Mit diesen Argumenten soll die vielfältig unter Beweis gestellte Bedeutung numerischer Verfahren und Ergebnisse nicht in Abrede gestellt werden. Allerdings ergeben sich im Vergleich von symbolischen und numerischen Methoden eine Reihe von Vorteilen, die eine strukturelle Sicht auf mathematische Fragestellungen gegenüber einer rein quantitativen bietet. Pavelle ([13]) listet die folgenden auf:

- *Erstens* gestattet eine präventive symbolische Analyse es oftmals, einen Ausdruck effektiver numerisch auszuwerten.

  Dieser einmalige vorherige Aufwand spielt besonders für Funktionen, die mit vielen verschiedenen Parameterbelegungen auszuwerten sind, eine entscheidende Rolle. Nach [14] werden moderne CAS zu 90 % gerade zu diesem Zweck, d.h. zur Generierung effizienten Codes, eingesetzt.

- *Zweitens* sind die so gewonnenen Aussagen durch das prinzipielle Vermeiden von numerischen Approximationen exakt in einem streng mathematisch-deduktiven Sinn. Damit werden Fehler- und Rundungsanalysen, die bei der Ergebnisverifikation numerischer Verfahren oftmals den Löwenanteil des Aufwands ausmachen, unnötig.

  Die Frage der Sicherung der Relevanz numerischer Simulationen ist eine zentrale Frage, die bei vielen praktischen Applikationen unbeantwortet im Raum stehen bleibt. Sie erfordern gewöhnlich umfangreiche Stabilitätsuntersuchungen, bei denen ihrerseits symbolische Methoden hilfreich sein können.

- Und *drittens* impliziert ein Ergebnis in einer symbolischen Form ein qualitativ tieferes Problemverständnis als es selbst durch eine große Menge rein numerischer Daten ausgedrückt werden kann.

  Zwar ermöglichen es (z.B. auf numerischen Verfahren basierende) Simulationen, größere Datenmengen zu erzeugen, die man – vielleicht noch unterstützt durch Visualisierungswerkzeuge – nach *Regelmäßigkeiten* absuchen kann. Jedoch erst eine (nur symbolisch mögliche) Analyse in einem streng deduktiven Verständnis von Mathematik vermag diese in *Gesetzmäßigkeiten* zu verwandeln, die unserem *Pool gesicherten Wissens* hinzugefügt werden können und die Basis für die Bearbeitung inverser Fragestellungen des Steuerns und Regelns bilden.

Symbolische Ergebnisse vermitteln also einen wesentlich tieferen strukturellen Einblick in Zusammenhänge als Daten aus (rein numerischen) Simulationen, erfordern aber zugleich einen komplizierteren mathematischen Apparat. Sicher kann man längst nicht alles von praktischer Relevanz bis zu einer solchen strukturellen Sicht theoretisch aufarbeiten und verdichten. Sie ist und bleibt jedoch das eigentliche Ziel mathematischer (und nicht nur mathematischer) Erkenntnis. [13] zitiert in diesem Zusammenhang einen Ausspruch von R.W.Hamming:

The purpose of computing is insight, not numbers.

1.7 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.

Gehen wir 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 [10]. 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.

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 verstärkt die Entwicklungslinien der konstruktiven Mathematik mit ihrer vorerst letzten Blüte in den 1920er Jahren wieder auf, vgl. R.Loos in [10].

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 verwendeten Methoden. Buchberger verwendet deshalb 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.

Auch wenn die Übergänge zu anderen Gebieten der algorithmischen Mathematik fließend sind, werden wir im Folgenden 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

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 zurück. [20] nennt in diesem Zusammenhang Arbeiten aus dem Jahre 1953 von H.G.Kahrimanian [9] und J.Nolan [11] 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 [16] 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. [20] enthält einen detaillierteren Überblick über die Entwicklungen und Systeme jener Zeit. Hulzen nennt insbesondere das System Mathlab-68 von C.Engelman, 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 und 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.

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

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 die Entwicklung 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 inzwischen 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 und Zeitschriften (zuletzt durch Gründung des Verlags Wolfram Media) aufgebaut, die MATHEMATICA über das eigentliche Softwareprodukt hinaus attraktiv machen. Eingeschlossen in dieses Engagement sind Plattformen wie das *Wolfram Information Center* (`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 aktuell gepflegte Datenbestände aus dem Wirtschafts- und Finanzbereich zugegriffen werden kann. Seit November 2008 ist die Version 7 verfügbar.

Maple

Ähnliche Überlegungen des „Downsizing“ der bis dahin nur auf Mainframes laufenden großen Systeme lagen der Entwicklung von MAPLE an der University of Waterloo zugrunde. In nur drei Wochen wurde Ende 1980 eine erste Version (mit beschänkten Fähigkeiten) entwickelt, die auf *B*, einer ressourcen- und laufzeitfreundlichen Sprache aus der BCPL-Familie aufsetzte, aus der sich C als heutiger Standard entwickelt hat. Seit 1983 sind Versionen von MAPLE auch außerhalb der University of Waterloo in Gebrauch. Zur Gewährleistung einer effizienten Portabilität auf immer neue Computergenerationen wurde im Gegensatz zu den großen LISP-Systemen 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) und andererseits in Softwareprodukte anderer Firmen Fähigkeiten zu symbolischen Rechnungen zu integrieren wie etwa in Mathcad (`http://www.mathcad.com`) oder Scientific Workplace, das „Word für Wissenschaftler“ (`http://www.mackichan.com`).

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 (bis hin zu leistungsfähigeren Personalcomputern unter Windows und Linux) 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, [15, 18]. 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“ (`http://www.acdca.ac.at`). In Deutschland wird Computeralgebra in Schulen heute vor allem über CAS-fähige Taschenrechner der Firmen Texas Instruments und Casio eingesetzt, siehe auch die Zusammenstellung der Computeralgebra-Fachgruppe `http://www.fachgruppe-computeralgebra.de`.

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.

Nach der Vorreiterrolle, welche Sachsen vor einigen Jahren deutschlandweit mit der flächendeckenden Einführung des grafikfähigen Taschenrechners (GTR) im Schulunterricht übernommen hatte, wird mit den im Jahr 2004 eingeführten neuen Lehrplänen auch der Einsatz von CAS und DGS (dynamischer Geometrie-Software) im Gymnasium ab Klasse 8 verbindlich geregelt und – mit der stufenweisen Einführung der Lehrpläne – ab 2005 wirksam.

Um auf diesem Markt (dessen wirtschaftliche Potenzen die des gesamten wissenschaftlichen Bereichs um Größenordnungen übersteigen) erfolgreicher agieren zu können, wurde DERIVE im August 1999 von Texas Instruments aufgekauft und bildet die Basis für die Software auf den verschiedenen Handhelds von TI mit CAS-Fähigkeiten. Im Rahmen der CA-Diskussionen im sächsischen Kultusministeriums tauchte ein bis dahin unbekanntes neues CAS *TI Interactive* (`http://education.ti.com`) auf, von dem prompt eine Landeslizenz erworben wurde.

Wie weit solche Produkte mit Laptops, welche die volle Leistungsfähigkeit „großer“ Systeme anbieten, konkurrieren können, wird die (nahe) Zukunft erweisen. Im Sommer 2006 hat TI die Weiterentwicklung von DERIVE als eigenständigem CAS eingestellt.

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-Projekten 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 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 gilt 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 bisher 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 CA-Systemen 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 aufgestellten „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 hat deshalb die eigens dafür gegründete Firma *SciFace* einen Teil dieser Aufgaben insbesondere aus dem software-technischen Bereich übernommen. Der Schwerpunkt der Entwicklungen lag auf dem Grafik-Teil sowie einer besseren Notebook-Oberfläche, wofür zunächst eine von Microsoft lizensierte Technologie zum Einsatz kam.

Damit verbunden war die kommerzielle Vermarktung von MuPAD, die zu der Zeit eine sehr kontroverse Debatte in der deutschen Gemeinde der Computeralgebraiker auslöste. Schließlich waren die entsprechenden 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 der akademische Bereich 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 hat.

Mit der Emeritierung von Prof. Fuchssteiner Ende 2005 wurde auch die universitäre Gruppe aufgelöst, so dass seitdem die Last der weiteren Entwicklung des CAS vollständig auf den Schultern der kleinen Firma *SciFace* liegt. Damit wurden Fragen der Sicherung eines ausreichenden cash flow essenziell für das Überleben des Systems und damit die Fortschreibung der bisher aufgewendeten Forschungs- und Entwicklungsanstrengungen insgesamt. Seit Anfang 2006 steht deshalb keine kostenlose Lightversion mehr zur Verfügung. Im März 2006 wurde mit MuPAD Pro 4.0 eine Entwicklung abgeschlossen, 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].

Trotz aller Anstrengungen – insbesondere auch im Rahmen von Kooperationen mit Firmen wie MacKichan (`http://www.mackichan.com`) sowie einem starken Engagement im Bereich „CA in der Schule“ im Rahmen der NRW-Kandesinitiative MuMM (Mathematikunterricht mit MuPAD) – blieb die finanzielle Situation von Sciface angespannt. Im Herbst 2008 wurde Sciface schließlich von MathWorks (`http://www.mathworks.com`) aufgekauft und MuPAD als symbolische Komponente (Toolbox) in das bis dahin stärker auf numerische Rechnungen spezialisierte Flaggschiff MATLAB integriert. MathWorks wechselt damit zugleich seine Strategie. Während vorher auf eine Kooperation mit Maple und damit eine Outsourcing-Strategie gesetzt wurde, wird 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 ins eigene Haus übernommen. Damit endete zugleich die eigenständige Entwicklung von MuPAD.

Magma ist ein zweites System, dessen Entwicklergruppe nicht in Amerikas residiert und welches in den letzten Jahren an Bedeutung gewann. Es hat seine Wurzeln in der Zahlentheorie und abstrakten Algebra, die bis zum System Cayley von J.Cannon und die 70er Jahre zurückreichen, und wird von einer Gruppe um John Cannon an der School of Mathematics and Statistics an 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

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

refinanzieren und hat ein Lizenzmodell entwickelt, mit welchem die wissenschaftlichen Einrichtungen, die das System nutzen, an dessen Refinanzierung beteiligt werden. Die Rückläufe werden (nach Aussagen von J. Cannon) 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 und Mitarbeiter 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 ist zu verzeichnen, dass im Laufe der 90er Jahre 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 gewonnen hat.

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 Wissenschaftlerkreisen 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.

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 heute unter der GPL als Open-Source-Projekt weiter vorangetrieben wird, siehe `http://maxima.sourceforge.net`. Seit September 2004 steht eine Version für Windows und Linux zur Verfügung.

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

> The current release of REDUCE includes all enhancements and bug fixes through January 1, 2009. ... The complete source code for REDUCE is available. On-line versions of the manual and other support documents and tutorials are also normally included with the distribution.
>
> REDUCE is now available free of charge from SourceForge. Please check this site for further details.
>
> Quelle: `http://www.reduce-algebra.com`

Es existieren einige weitere Open-Source-Projekte zur Computeralgebra wie etwa YACAS (Yet Another Computaer Algebra System, `http://yacas.sourceforge.net`), 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 dehalb nur denkbar

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

Für alle drei Ansätze gibt es Beispiele. Den ersten hatten wir mit MAXIMA bereits belegt. Der zweite wurde und wird mit wechselndem Erfolg und Intensität im Rahmen des OSCAS-Projekts (OSCAS = Open Source Computer Algebra System) immer wieder aufgenommen.

Für den dritten 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 T.J. 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 [20] 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 bewogen haben, 1992 im Zuge einer generellen „Flurbereinigung“ des Firmenprofils die Rechte an AXIOM der Firma NAG Ltd. zu übertragen, einer 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` und `http://savannah.nongnu.org/projects/axiom`. Der strenge Typkonzepte unterstützende Compiler war bereits vorher unter dem Namen ALDOR unter die Fittiche der Nutzergemeinde genommen worden. Die lange und wechselvolle Geschichte ist unter `http://www.aldor.org` dokumentiert.

System	aktuelle Version	Webseite	Preis Einzellizenz	Preis Stud.-version
Axiom	Sept 2008	`www.axiom-developer.org`	Open Source	
GAP	4.4.10	`www.gap-system.org`	kostenfrei	
Magma	2.14-17	`magma.maths.usyd.edu.au`	1200 $	100 $
Maple	12	`www.maplesoft.com`	940 €	94 €
Mathematica	7	`www.wolfram.com`	1345 €	128 € (A)
Maxima	5.16.3	`maxima.sourceforge.net`	Open Source	
MuPAD	4.0 Pro	`www.mupad.de`	von MathWorks aufgekauft und in MATLAB integriert	
MATLAB	7.7	`www.mathworks.com`	?? €	?? €
Reduce	3.8 (B)	`www.reduce-algebra.com`	495 €	99 €
Yacas	1.2.2 (B)	`yacas.sourceforge.net`	Open Source	

Die großen Computeralgebrasysteme allgemeiner Ausrichtung im Überblick
(Stand November 2008)

(A) Semester Edition: 29 €
(B) keine aktive Weiterentwicklung sichtbar

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) entwickelt 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.

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, das später aus markenrechtlichen Gründen in AXIOM umbenannt wurde. 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.

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 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 wird 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 die bis heute rudimentären Versuche mit „Templates“ und Ähnlichem in den „großen Sprachfamilien“ C/C++/C#, PASCAL/MODULA/OBERON oder Java noch nicht zu denken war.

Ein solcher Ansatz wird auch im Design von MAGMA, einem Nachfolger von CAYLEY, verfolgt, das als CAS für komplexe Fragestellungen aus den Bereichen Algebra, Zahlentheorie, Geometrie und Kombinatorik entworfen wurde und Konzepte der universellen Algebra und Kategorientheorie in einem objektorientierten Ansatz direkt umsetzt.

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 auch kausal 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 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, die von MAPLE als *Maple Shared Library* oder von MATHEMATICA als *MathSource* mit den Distributionen mitgeliefert wurden, in die entsprechenden Portale `http://www.maplesoft.com` und `http://library.wolfram.com` integriert, über welche Nutzer relevante Materialien bereitstellen oder suchen und sich herunterladen können. Auch MUPAD hat unter `http://www.sciface.com` mit dem Aufbau eines solchen Portals begonnen.

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 MUPAD zum Beispiel

- Eingaben der Form `?topic` zur Aktivierung des Hilfesystems und

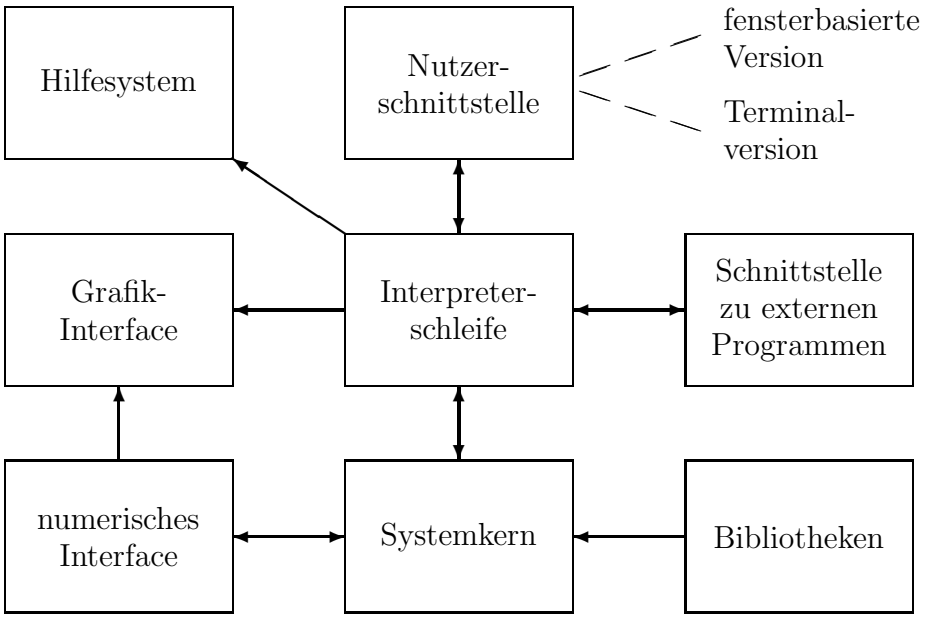

Bild 1: Prinzipieller Aufbau eines Computeralgebra-Systems

- Eingaben der Form `plot(...)`, worauf im Rahmen der X-Windows-Umgebung ein Grafik-Ausgabefenster geöffnet wird.

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

Die Interpreterschleife des CAS bringt also verschiedene Teile des Systems zusammen, wovon nur eines der unmittelbar symbolische Rechnungen ausführende Kern ist, den wir als den *Systemkern* bezeichnen wollen. Neben den beiden Komponenten *Hilfesystem* und *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 besondere Vorreiterrolle spielen auf diesem Gebiet die Systeme MATHEMATICA und MAPLE mit dem Konzept des *Arbeitsblatts*. Hier treffen sich Entwick-

lungen aus dem Bereich des wissenschaftlichen Publizierens (mit Produkten wie *Scientific Word* von MacKichan), der Gestaltung interaktiver Oberflächen und Methoden des symbolischen Rechnens, womit sich zugleich vollkommen neue Horizonte in Richtung der (elektronischen) Publikation interaktiver mathematischer Texte eröffnen.

Hilfesysteme sind bei den einzelnen CAS sehr unterschiedlich entwickelt. Generell ist jedoch auch hier ein Trend hin zur Verwendung gängiger Techniken zum Entwurf von Hilfesystemen in Form hypertextbasierter Dokumente und entsprechender Analysewerkzeuge zu verspüren. Dabei wird zunehmend, 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.

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 Werkzeug, einem persönlichen digitalen Mathematik-Assistenten, in der Hand eines Wissenschaftlers oder Ingenieurs wird, wenn es die volle „compute power“ bereitstellt, die von dieser Klientel im Alltag benötigt wird. Dazu gehören 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 nachfolgender numerischer 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.

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:

- 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.

[1] Nach [14] werden moderne CAS zu 90 % zur Generierung effizienten Codes eingesetzt.

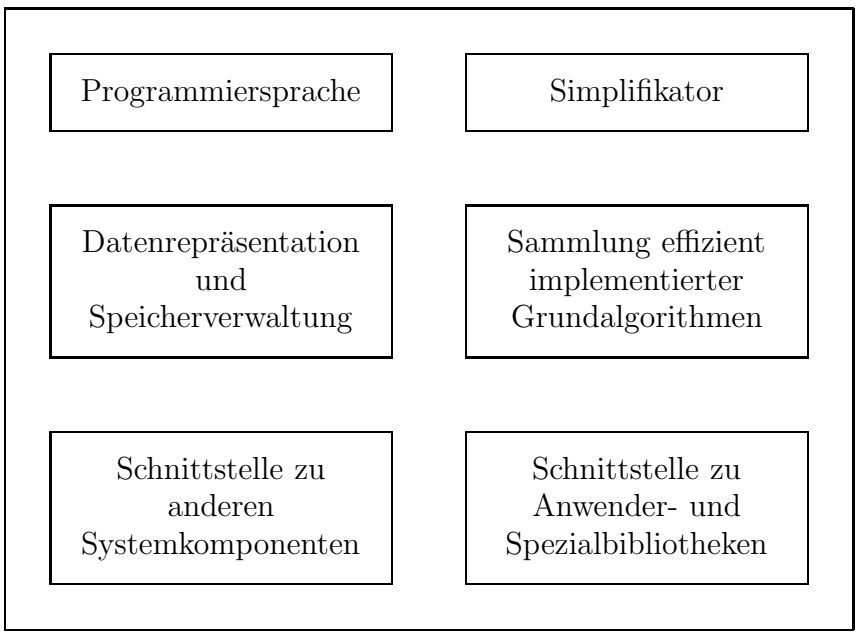

Bild 2: Komponenten des Systemkern-Designs

- 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 [19] 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 ausgeführt werden kann.

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.
symbolisch: Als Form der Darstellung symbolischer Inhalte bleibt ein Parsebaum auch nach der Syntaxanalyse bestehen.
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.
Die Prozesse in einem CAS zur *Laufzeit* haben damit viele Gemeinsamkeiten mit den Analyseprozessen in einem Compiler zur *Übersetzungszeit*.

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 [25] 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* zu-

sammengesetzt 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 [19]) 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 der mit einem Bezeichner zu erwartende „Inhalt“ vorstrukturiert werden kann. 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, werden wir zu einem späteren Zeitpunkt besprechen.

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 Söhne hat noch immer den Nachteil geringer Homogenität, da Knoten mit unterschiedlicher Anzahl von Söhnen 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 und REDUCE intern dargestellt werden. Die folgenden Prozeduren gestatten es jeweils, diese innere Struktur 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[2] dar.

MAXIMA:

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

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]]
```

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

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

Ähnliche Gemeinsamkeiten findet man auch bei der Struktur anderer Ausdrücke. Eine Zusammenstellung verschiedener Beispiele finden Sie in der Tabelle.

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. Ausdücke als 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` 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 den meisten Grundkursen „Algorithmen und Datenstrukturen" (etwa [12] oder [24]). Die dort eingeführte *destruktiv veränderbare Listenstruktur* mit den zentralen Operationen `Einfügen` und `Entfernen` 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[3] erlaubt es, auf aufwändiges Duplizieren von Teilausdrücken zugunsten des Duplizierens von Pointern zu verzichten, indem beim Kopieren einer Liste die Referenzen übernommen, die Links aber kopiert werden. Diese Sprachelemente prädestinieren einen funktionalen und rekursiven Umgang mit Listen, der deshalb in LISP und damit in CAS weit verbreitet ist. Für einen aus einer imperativen Welt kommenden Nutzer ist das gewöhnungsbedürftig.

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 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 MATHEMATICA überdies der Kopfterm einer Sequenz ebenso leicht ausgetauscht werden (zweite Zeile) wie durch `op` in MUPAD (erste Zeile).

```
u:=[1,2,3]: _plus(op(u));
u={1,2,3}; Apply[Plus,u]
```

6

Datendarstellung in MUPAD

In CAS der dritten Generation liegt zwischen der internen Datendarstellung und dem Nutzerinterface noch eine weitere Schicht der Datenorganisation. Im Fall von MUPAD wird ein objektorientierter Ansatz verfolgt, der Ausdrücke verschiedenen Grundbereichen (Domains) zuordnet,

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

deren interner Aufbau jeweils spezifisch organisiert ist. Die Funktion `domtype` ermittelt den Typ des jeweiligen Ausdrucks, wobei klassische Ausdrücke meist vom Typ `DOM_EXPR` sind, die intern ebenso dargestellt werden wie Ausdrücke in anderen CAS (siehe oben). Ausdrücke anderer Typen haben nicht unbedingt ein Feld `op(u,0)`. Mit gewissen Einschränkungen können Sie die folgenden Funktionen zur Strukturbestimmung verwenden:

```
level1:=proc(u)
 begin
  if domtype(u)=DOM_EXPR then [op(u,0..nops(u))] else u end_if;
end_proc;

structure:=proc(u) local i;
begin
  if args(0)>1 then structure(args(i))$i=1..args(0)
  elif type(u) in {DOM_LIST,DOM_SET} then
       [type(u), structure(extop(u,i))$ i = 1 .. extnops(u)]
  elif extop(u,0)=FAIL then u
  else [structure(extop(u,i))$ i = 0 .. extnops(u)]
  end_if
end_proc;
```

Beispiele:

```
level1(a+b*c);
                              [_plus, a, b*c]
structure(a+b*c);
                           [_plus, a, [_mult, b, c]]
structure(sin(x));
                                   [sin, x]
structure(sin(2));
                                   [sin, 2]
M:=Dom::Matrix();
A:=M([[1,2],[3,4]]);
structure(A);

  [Dom::Matrix(), 2, 2, [DOM_LIST,
        poly(3*_X^2 + _X, [_X]), poly(4*_X^2 + 2*_X, [_X])], FLAG]
```

`level1` bestimmt die Struktur der obersten Ebene, wenn es sich um einen Ausdruck vom Typ `DOM_EXPR` handelt. Dies entspricht in etwa der Wirkung der Funktion `prog::exprtree`, die ebenfalls nur die Struktur von Ausdrücken des Typs `DOM_EXPR` expandiert.

`structure` versucht, die interne Struktur von Ausdrücken auch anderer Typen genauer zu bestimmen, wobei berücksichtigt wird, dass in einem Slot auch ganze Ausdruckssequenzen stehen können. `print(NoNL,A)` gibt den Ausdruck A mit `PRETTYPRINT:=FALSE` aus, was ebenfalls einen gewissen Einblick in die innere Struktur geben kann. Für Funktionen steht auch die Ausgabe mit `expose` zur Verfügung.

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

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

x symbolisch gemeint sein, selbst wenn der Bezeichner bereits einen Wert besitzt[4].

Sehen wir uns diese Besonderheit des Variablenkonzepts zunächst in einigen MuPAD-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;
```

$$x := \mathrm{free}$$

Um x als Symbol zu verwenden, müssen wir diese Auswertung verhindern, was in MuPAD durch die Funktion `hold` erreicht werden kann. Leider geht nun gar nichts mehr.

```
x:=hold(x);
p;
```

```
Error: Recursive definition
[See ?MAXLEVEL]
```

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, die hier abgebrochen wird, wenn ein entsprechender Rekursionszähler eine vorgegebene Grenze überschreitet.

Natürlich könnte man für diesen Fall entweder die offensichtlich zu unendlicher Rekursion führende Zuweisung `x:=hold(x)` unterbinden (etwa Reduce) oder einfach mit dem Auswerten aufhören,

[4] Für x im Wertmodus führt der Aufruf in MuPAD zu einem Fehler, d. h. x wird ausgewertet.

wenn sich nichts mehr ändert (MATHEMATICA), aber beide Ansätze lassen sich leicht durch etwas komplexere Zuweisungsnetze aushebeln.

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.

```
delete 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`

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'`).

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. `unames()` listet alle Bezeichner im Symbolmodus (unassigned names) auf, `anames()` alle Bezeichner im Wertmodus. In MUPAD kann man sich mit `anames(All)` einen Überblick über alle Bezeichner im Wertmodus verschaffen.

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.

Mit dem MATHEMATICA-Befehl `Remove` können auch Einträge aus dieser Symboltabelle entfernt werden. Eine solche Möglichkeit ist aber mit Vorsicht zu verwenden, da es sein kann, dass später auf das bereits gelöschte Symbol weitere Referenzen gesetzt werden. Stattdessen sollten Sie nur mit `Clear` bzw. `ClearAll` arbeiten.

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 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. Dies 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 MuPAD-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.

Ähnlich den Bezeichnern für Variablen können auch neue Funktionsbezeichner eingeführt werden, von denen zunächst (fast) nichts[5] bekannt ist und die damit im Symbolmodus als Funktionssymbole ohne Funktionsdefinition behandelt werden.
Mit dem neuen Funktionssymbol f wurden zwei Ausdrücke `f(x)` und `f(2)` konstruiert.

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

$$f(x) + 2\,x + 1$$

```
subs(u,x=2);
```

$$f(2) + 5$$

Transformationen

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

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

[5] Aus der Syntax `f(x)` ist natürlich bekannt, dass f ein Funktionssymbol ist. MuPAD kennt für alle solche Funktionssymbole von sich aus die Kettenregel und führt eine entsprechende *Transformation* aus.

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 MuPAD-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(sum1*sum2)` 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[6].

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 des Ausdrucks zu analysieren sind.

[6] `expose(expand)` gibt nur ein `builtin`-Konstrukt zurück.

[7] In MuPAD heißt sie `sin::expand` und kann mit `expose(sin::expand)` studiert werden.

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

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 Abarbeiten 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[8]. 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;
```

$$(1+x)^3$$

In MUPAD wird derselbe Effekt über den objektorientierten Ansatz erreicht, der aber intern ebenfalls auf symbolische Ausdrücke zurückgreift, an denen der Grundbereich des zu faktorisierenden Polynoms erkannt wird.

```
Z:=Dom::IntegerMod(2);
p:=poly(f,[x],Z);
```

$$\text{poly}\left(x^3+x^2+x+1,[x],\mathbb{Z}_2\right)$$

```
factor(p);
```

$$\text{poly}\left(x-1,[x],\mathbb{Z}_2\right)^3$$

Ähnlich wird das Problem in AXIOM gelöst. `f:A:=B` deklariert den Bezeichner f als Variable aus dem Bereich (Domain) A und weist ihr den (typkorrekten) Wert B zu. `UP` ist die Abkürzung für den Domainkonstruktor `UnivariatePolynomial` und `PF` die Abkürzung für den Domainkonstruktor `PrimeField`. Im Gegensatz zu MUPAD wird die Eingabe x^3+x^2-x+1, die zunächst als `Polynomial Integer` f verstanden wird, in der zweiten Zuweisung als Nebeneffekt zu einem Datum p des Zieltyps umgebaut (type coercion).

```
f:=x^3+x^2-x+1;
p:UP(x,PF(2)):=f
```

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

Type: UnivariatePolynomial(x, PrimeField 2)

```
factor(p)
```

$$(x-1)^3$$

Type: Factored UnivariatePolynomial(x, PrimeField 2)

In beiden Fällen ist die Information über den aufzurufenden Algorithmus symbolisch im *Wert* des Bezeichners p gespeichert, der Transformationsmechanismus also im objektorientierten Ansatz versteckt.

Transformationen sind ihrer Natur nach *rekursiv,* da nach einer Transformation ein Ausdruck entstehen kann, auf den weitere Transformationen angewendet werden können.

[8] Für viele Optionen aus Paketen gilt dies leider nicht.

So führt MuPAD bei dieser Berechnung erst die Transformation $\exp(\ln(u)) = u$ und dann $\cos(\arcsin(x)) = \sqrt{1-x^2}$ aus.

```
cos(exp(ln(arcsin(x))));
```

$$\sqrt{1-x^2}$$

Derselbe Funktionsbezeichner kann in unterschiedlichem Kontext in verschiedenen Rollen auftreten, wie dieses Beispiel für die `exp`-Funktion zeigt. Im ersten Fall erhalten wir einen Funktionsausdruck mit dem Symbol `exp` als Kopf, im zweiten Fall eine Float-Zahl, die mit einem entsprechenden Näherungsverfahren in einem Funktionsaufruf berechnet wurde, im letzten Fall dagegen wurde eine Transformation angewendet, die das Zusammentreffen von `exp` und `ln` erkannt hat.

```
exp(x);
```

$$\exp(x)$$

```
exp(2.0);
```

$$7.389056099$$

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

$$x + y$$

Auch die Ergebnisse von auf den ersten Blick stärker „algorithmischen“ Funktionen wie etwa `diff`, mit der man in MuPAD 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.

```
h:=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.

Ersetzen wir g durch ein „bekanntes“ Funktionssymbol, so erhalten wir die aus der Kettenregel gewohnten Formeln.

```
eval(subs(h,f=sin));
```

$$\cos(g(x))\frac{d}{dx}g(x)$$

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 und MuPAD näher an der üblichen mathematischen Notation als `f'[x]` bzw. `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.

`f'` ist dabei keine neue syntaktische Einheit, sondern das Ergebnis der Anwendung des Postfix-Operators `'` auf f, wie eine Strukturanalyse zeigt.

```
f'(x);
```

$$D(f)(x)$$

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

```
D(sin);
```

$$\cos$$

```
D(exp+ln);
```

$$\exp\ +\ \left(a \mapsto a^{-1}\right) \qquad \text{(Maple)}$$

$$\frac{1}{id} + \exp \qquad \text{(MuPAD)}$$

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.

MATHEMATICA verwendet in seinen Ausgaben die Notation ... [#] & einheitlich auch in einfachen 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'}
```

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

Es ist sogar denkbar, dass nicht nur der Name, sondern auch die genaue Zuordnungsvorschrift nicht bekannt ist, sondern nur eine semantische Spezifikation der Funktion als „Black Box“. Solche Antworten entstehen etwa, wenn Interpolationsfunktionen mit speziellen Eigenschaften zurückgegeben werden.

2.7 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 MUPAD 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. In MUPAD kann man diese Tiefe mit dem Kommando `level` variieren.

```
level(a,i)$i=1..6;
```

$$b+1, c+4, 7, 7, 7, 7$$

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

```
b:=7*d+3: a;
```

$$7\,d+4$$

```
level(a,i)$i=1..3;
```

$$b+1, 7\,d+4, 7\,d+4$$

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

Die bisher verwendeten Bezeichner auf den rechten Seiten waren im Symbolmodus. Werden Bezeichner im Wertmodus verwendet, so müssen wir unterscheiden, ob wir das Symbol oder dessen Wert meinen.

a wurde 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:=hold(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.

Geht ein Bezeichner im Wertmodus in einen Ausdruck ein, so ist 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*.

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

$$[3, 5]$$

```
level(a,i)$i=1..5;
```

$$3, 3, 3, 3, 3$$

```
level(b,i)$i=1..5;
```

$$u, 5, 5, 5, 5$$

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, wenn also von Anfang an genau festgelegt wird, welche Bezeichner im Symbolmodus und welche im Wertmodus verwendet werden sollen. Bezeichnern im Symbolmodus sollte konsequent 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	subst(x=a,f(x))	x:a	–	'x
MAPLE	subs(x=a,f(x))	x:=a	–	'x'
MATHEMATICA	f(x) /. x↦a	x=a	x:=a	Hold[x]
MUPAD	subs(f(x),x=a)	x:=a	–	hold(x)
REDUCE	subst(x=a,f(x))	x:=a	let x=a	–

Tabelle 4: Substitutions- und Zuweisungsoperatoren der verschiedenen CAS

Bei diesem Evaluierungsverfahren 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.

MUPAD bricht nach Erreichen einer durch die Systemvariable `MAXLEVEL` vorgegebenen Tiefe mit einer Fehlermeldung ab. Das Auswerteverhalten kann über eine weitere Variable `LEVEL` gesteuert werden, die angibt, wie viele Auswerterunden im aktuellen Kontext ausgeführt werden. Die Standardwerte sind `LEVEL` = `MAXLEVEL` = 100.

Innerhalb von Funktionsdefinitionen wird aber `LEVEL` = 1 gesetzt 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.

```
uhu:=proc()
  begin print(LEVEL)
end_proc;
uhu();
```

1

Dasselbe Prinzip – Auswertung nur eine Ebene tief – könnte man sich auch als globales Auswertungsschema vorstellen, wenn es gleichzeitig eine Funktion `eval` gibt, mit welcher eine mehrfache Auswertung ausgeführt werden kann. Dieses Verhalten könnte man in MUPAD durch die globale Setzung `LEVEL:=1` herbeiführen. In den CAS MAXIMA und AXIOM wird es standardmäßig verwendet. Das Auswertungsverhalten unterscheidet sich damit von dem der anderen CAS.

Das Auswertungsverhalten der anderen CAS kann in MAXIMA (in erster Näherung) mit der Evaluierungsfunktion `ev` erreicht werden. Allerdings lassen sich mit `ev` in MAXIMA komplexere Effekte erzielen.

Zuweisung	MAXIMA	MUPAD
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 5: Unterschiedliches Auswertungsverhalten von MAXIMA und MUPAD

Datentypen und Polymorphie

Beim Auswerten von Ausdrücken spielt in klassischen Programmiersprachen *Polymorphie* eine wichtige Rolle, d.h. die Möglichkeit, Funktions- und insbesondere Operatorsymbolen unterschiedliche Bedeutung in Abhängigkeit von der Art der Argumente zu geben. Funktionen auf verschiedenen Strukturen mit gemeinsamen Eigenschaften durch dasselbe Symbol zu bezeichnen ist in der mathematischen Notation weit verbreitet und auch notwendig, um geeignete Abstraktionen (Gruppen, Ringe) überhaupt formulieren zu können.

Gleichwohl muss man diese Ambiguitäten in konkreten Implementierungen wieder auflösen, da natürlich z. B. zur Berechnung von $a + b$ für ganze und reelle Zahlen unterschiedliche Verfahren aufzurufen sind. Diese Unterscheidung vermögen Compiler klassischer Sprachen selbstständig zu treffen. Sie sind darüber hinaus auch in der Lage, etwa bei der Addition einer ganzen und einer reellen Zahl selbstständig eine passende Typkonversion auszuführen.

Beschränkt sich Polymorphie in klassischen Sprachen wie C noch auf von den Systemdesignern vorgesehene Typambiguitäten, so hat sie inzwischen mit dem Paradigma der Objektorientierung auch in allgemeinerer Form in die Informatik Einzug gehalten. Grundlage für Polymorphie ist in jedem Fall die Möglichkeit, Referenzen auf denselben Funktionsnamen an Hand der *Typinformationen*, welche die Parameter des jeweiligen Funktionsaufrufes tragen, aufzulösen.

Typsysteme sind damit ein wichtiges Instrument in modernen Programmiersprachen und wären sicher auch für das symbolische Rechnen von Bedeutung. Warum fehlt den großen CAS, die unter dem Begriff „Systeme der 2. Generation“ zusammengefasst werden, trotzdem ein strenges Typkonzept bzw. ist ein solches nur in rudimentären Ansätzen vorhanden?

Untersuchen wir dazu am Beispiel des einfachen Ausdrucks `f=2*x+3`, welche Ansprüche an ein Typsystem im symbolischen Rechnen zu stellen sind. f müsste ein passender Datentyp `Polynomial` zugeordnet werden, der (im Sinne eines des Typkonzepts eines abstrakten Datentyps) nicht nur ein Bezeichner ist, sondern für eine Signatur steht. Da in die Definition der Signaturen der Polynomoperationen die Signaturen der Operationen auf Koeffizienten und auf Termen eingehen, müsste ein qualifiziertes Typsystem wenigstens die unterschiedlichen Koeffizientenbereiche berücksichtigen.

In Java könnte man dazu etwa ein `interface Coeff` mit entsprechender Signatur vereinbaren und dann jeweils Koeffizientenbereiche `CInteger implements Coeff`, `CFloat implements Coeff` etc. verwenden, welche diese Schnittstelle implementieren. Polynome ließen sich dann als Listen von Instanzen der Klasse `class Monom {Coeff c; Term t;}` implementieren. Neben dem Umstand, dass + nicht als Operationssymbol verwendet werden kann, da in Java (im Gegensatz zu C++) Operatoren nicht neu überladen werden können, schlägt negativ zu Buche, dass in einer solchen Liste von Monomen Koeffizienten aus verschiedenen Bereichen bequem nebeneinander existieren können.

Eigentlich wird also ein Datentypkonstruktor `Polynomial(R:Ring)` benötigt, in welchem die Koeffizienten eines Polynoms alle aus *derselben* Ringinstanz R gewählt sind. R ist hier ein formaler Parameter, für den jede Datentypimplementierung, etwa rationale Zahlen, Restklassenbereiche, komplexe Zahlen oder gar Matrizen, eingesetzt werden kann, welche die unter dem Bezeichner `Ring` zusammengefasste Signatur realisiert.

Die Umsetzung dieses Ansatzes als *C++-Template* `Polynomial<R>` verwendet diese Signaturinformation nicht und hat darüber hinaus den Nachteil, dass für jeden konkreten Ring R eine neue Klasse `Polynomial<R>` generiert und compiliert wird, was den Code des Gesamtsystems unnötig aufbläht. Es würde nämlich ausreichen, eine einzige compilierte Instanz von `Polynomial` vorzuhalten, wenn dort die Einsprungpunkte für die Methoden der Schnittstelle R als Erweiterungspunkte verfügbar wären. Dies wird mit Java Generics seit Java 5 zum Teil umgesetzt.

Ist darüber hinaus die Signatur von R zur Compilezeit bekannt, so lassen sich mit einer solchen generischen Implementierung der Polynomoperationen in `Polynomial(R)` relativ zu den Operationen aus R typsicher Rechnungen in Polynomringen über den verschiedensten Grundbereichen ausführen, sobald eine (neue) Implementierung eines solchen Grundbereichs zugeschaltet wird.

Dieser Ansatz erlaubt es dann auch, Polynomringe *dynamisch* zu erzeugen, d. h. als Koeffizienten Bereiche zu verwenden, an die bei der Erstellung des Polynomcodes nicht gedacht worden ist bzw. die vielleicht noch nicht einmal zur Verfügung standen. Solche Konzepte von *Typsprachen* sind aus dem funktionalen Programmieren gut bekannt, ebenso die Tatsache, dass das Wortproblem für einigermaßen aussagekräftige Typsprachen (insbesondere solche mit Selbstreferenzen) algorithmisch nicht entscheidbar ist.

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

Funktionale Programmiersprachen vermeiden (aus gutem Grund) die Einführung von Variablen als „Wertcontainer“, denn insbesondere das *Problem der Inferenz von Datentypen* bringt erhebliche Schwierigkeiten mit sich. Um etwa den Ausdruck `f := 2*x+3` als `Polynomial(Integer)` zu interpretieren, ist in einem ersten Schritt `2*x` zu berechnen, d. h. das Produkt aus `2:Integer` und `x:Symbol` zu bilden. Dazu muss der gemeinsame Oberbereich erst gefunden werden, in den beide Datentypen transformiert werden können, damit die Multiplikation ausgeführt werden kann. Noch komplizierter wird es bei der Berechnung von $f + g$ mit $g = 2/3$. Für g ergibt sich bei entsprechender Analyse der Datentyp `Fraction(Integer)`, für f dagegen `Polynomial(Integer)`. Das Ergebnis könnte entweder die Einbettung `Integer` $\subset$ `Polynomial(Integer)` verwenden und damit zum Typ `Fraction(Polynomial(Integer))` einer rationalen Funktion mit ganzzahligen Koeffizienten führen oder aber die Einbettung `Integer` $\subset$ `Fraction(Integer)` verwenden und damit zum Typ `Polynomial(Fraction(Integer))` eines Polynoms mit rationalen Koeffizienten führen.

Beiden Fällen ist gemein, dass der entsprechende Obertyp, in den eine Typkonversion erfolgen muss, aus einer Obertyprelation des *Parameters* zu inferieren ist. Sind Polynome intern als Listen von Koeffizienten dargestellt, so wären Daten des Typs `Fraction(Polynomial(Integer))` Paare von Listen, Daten des Typs `Polynomial(Fraction(Integer))` dagegen Listen von Paaren. Programmiertechnisch sind mit solchen Typkonversionen also aufwändige Restrukturierungen der inneren Darstellung von Objekten des jeweiligen Typs verbunden.

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

Allerdings gibt es langjährige Untersuchungen zu den genannten Fragen, die mit dem Open-Source-Projekt ALDOR `http://www.aldor.org` bzw. innerhalb von AXIOM bis zu einer praktisch einsetzbaren Programmiersprache mit strengem Typkonzept für symbolische Rechnungen geführt worden sind (und weiter geführt werden). In AXIOM wird die erste Umwandlungsmöglichkeit automatisch erkannt.

```
f:=2*x+1
```

$$2\,x + 1 \qquad \text{Type: Polynomial Integer}$$

```
f+2/3
```

$$2\,x + \frac{5}{3} \qquad \text{Type: Polynomial Fraction Integer}$$

Die zweite kann durch eine explizite Typumwandlung von f erzwungen werden.

```
f::(Fraction Polynomial Integer)+2/3
```

$$\frac{6\,x + 5}{3} \qquad \text{Type: Fraction Polynomial Integer}$$

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 koplexeren Programmkonstrukten zuwenden, die wir weiter vorn im Konzept der *Anweisung* bzw. Steeuerstrukturen 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 wie PASCAL oder C orientieren. Auch Unterprogrammtechniken stehen zur Verfügung, wie wir im Abschnitt „Funktionen“ bereits gesehen hatten. Wir verzichten deshalb hier auf eine detaillierte Darstellung dieser Konzepte und Sprachmittel.

Im letzten Abschnitt kommen wir 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]]`
MUPAD	`op(l,i)` oder `l[i]`	`op(l,[i,j])` oder `l[i][j]`
REDUCE	`part(l,i)`	`part(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. In Maple und MuPAD wird dafür ein eigener Operator verwendet. Reduce hat 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}]`
MuPAD	`[i^2 $ i=1..5]`
Reduce	`for i:=1:5 collect i^2`

Tabelle 7: Liste der ersten 5 Quadratzahlen generieren

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]`
MuPAD	`select(isprime,[$1..50])`
Reduce	`for i:=1:50 join if primep(i) then {i} else {}`

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 MuPAD hier offensichtlich die Transformation

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

angewendet.

```
u:=[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];
```

$$[1, 2, 3]$$

```
sin(u);
map(u,sin);
```

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

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

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 (MuPAD)

  ```
  [f(l1[i],...,ln[i]) $ i=1..nops(l1)]
  ```

  implementiert werden kann, stellen einige CAS aus Geschwindigkeitsgründen eine spezielle Zip-Funktion zur Verfügung.

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

	Mapping	Join	Zip
Axiom	`map(f,l)`		
Maxima	`map(f,l)`	`flatten(l)`	
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`]]`
MuPAD	`map(l,f)`	`map(l,op)`	`zip(`l_1`, `l_2`,f)`
Reduce	`map(f,l)`	`for each x in l join x`	

Tabelle 9: Weitere elementare Listentransformationen

Ein komplexes Beispiel

Zur Illustration wird nun in einem komplexen Beispiel das Zusammenwirken der verschiedenen Listentransformationen 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.

```
Z:=Dom::IntegerMod(31);
wertetafel:=[[x,Z(x^3+x+1)]$x=0..30];
```

```
[[0, 1 mod 31], [1, 3 mod 31], [2, 11 mod 31], [3, 0 mod 31], [4, 7 mod 31],
[5, 7 mod 31], [6, 6 mod 31], [7, 10 mod 31], [8, 25 mod 31], [9, 26 mod 31],
[10, 19 mod 31], [11, 10 mod 31], [12, 5 mod 31], [13, 10 mod 31], [14, 0 mod
31], [15, 12 mod 31], [16, 21 mod 31], [17, 2 mod 31], [18, 23 mod 31], [19, 28
mod 31], [20, 23 mod 31], [21, 14 mod 31], [22, 7 mod 31], [23, 8 mod 31], [24,
23 mod 31], [25, 27 mod 31], [26, 26 mod 31], [27, 26 mod 31], [28, 2 mod 31],
[29, 22 mod 31], [30, 30 mod 31]]
```

Sie 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:=select(wertetafel,
     x -> iszero(op(x,2)));
```

```
[[3, 0 mod 31], [14, 0 mod 31]]
```

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

```
map(u,x -> x[1]);
```

```
[3, 14]
```

Eine kompakte Lösung der Aufgabe lautet also:

```
select([$0..30],x-> iszero(Z(x^3+x+1)));
```

Mit dem folgenden Kommando können Sie sich einen Überblick über die Nullstellen des Polynoms $x^3 + x + 1 \pmod p$ für verschiedene Primzahlen p verschaffen:

```
sol:=proc(p)
begin
  if isprime(p)<>TRUE then hold(sol)(p)
  else select([$0 .. (p-1)], x -> iszero(Dom::IntegerMod(p)(x^3+x+1)))
  end_if
end_proc:
```

Das `if`-Kommando beschränkt den Definitionsbereichs von `sol` auf Primzahlen. Für alle anderen Argumente x bleibt `sol(p)` in seiner symbolischen Form stehen.

Nun wird diese Funktion auf die Primzahlen kleiner als 50 angewendet, die vorher in einer Liste `primelist` aufgesammelt werden.

```
primelist:=select([$1..50],isprime):
map(primelist,p-> [p,sol(p)]);
```

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

Sie 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 nachgeprüft werden.

```
map(primelist,p-> [p,map(factor(poly(x^3+x+1,[x],IntMod(p))),expr)]);
```

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

Einige Erläuterungen: `poly(x^3+x+1,[x],IntMod(p))` konstruiert das Polynom $f = x^3 + x + 1 \in \mathbb{Z}_p[x]$, `factor(f)` zerlegt dieses Polynom in Faktoren, allerdings als Objekt vom Typ `Factored`, der für die Ausgabe nicht so optimal ist. Deshalb werden vom äußeren `map` die einzelnen Faktoren in Ausdrücke vom Typ `DOM_EXPR` zurückverwandelt.

Alternativ kann auch der Bereichskonstruktor `Dom::UnivariatePolynomial` verwendet werden.

```
UP:=p-> Dom::UnivariatePolynomial(x,Dom::IntegerMod(p));
map(primelist,p-> [p,expr(factor(UP(p)(x^3+x+1)))]);
```

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

$$\{x^2 + y = 2,\ y^2 + x = 2\}$$

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})];
```

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

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.

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

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

Beachten Sie, dass `s` eine Liste aus 3 Elemente war, jetzt aber in eine Liste von 4 Lösungen 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 `s` in `sys` substituieren und nachfolgend vereinfachen

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

Allerdings ist es nicht sehr weitsichtig, hier das Kommando `print` zu verwenden, 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)),s);
```

$$[\{2 = 2\},\ \{2 = 2\},\ \{2 = 2\},\ \{2 = 2\}]$$

Die booleschen Ausdrücke $2 = 2$ werden aus noch zu erläuternden Gründen nur sehr zögerlich ausgewertet. Dies können wir hier wie folgt erzwingen:

```
map(x -> evalb(x[1]),probe);
```

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

Ähnlich kann man auch in MUPAD vorgehen, wobei die Gleichungen auch als Liste angegeben werden können.

```
polys:=[x^2+y=2,y^2+x=2];
sol:=solve(polys,{x,y});
```

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

MuPAD produziert die Lösungen bereits in expandierter Form. Der Rückgabetyp ist außerdem bereits eine Menge (von Lösungspaaren) und nicht nur eine Sequenz wie in Maple. Für die Probe verwandeln wir die Lösungs*menge* in eine Liste, um Reihenfolge und Anzahl der Ergebnisse zu erhalten.

```
probe:=map(coerce(sol,DOM_LIST),v -> expand(subs(polys,v)));
```

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

`expand` bewirkt dabei das Auflösen von Klammern. Im Gegensatz zu früheren Versionen wird das Kommando nun automatisch auf die Elemente angewendet, wenn das Argument von `expand` eine Liste ist.

Die Gründe für die zögerliche Auswertung der booleschen Ausdrücke zu $2 = 2$ und nicht zu `true` besprechen wir weiter unten. Eine komplette Auswertung kann mit dem Kommando `is` (oder `bool`) erreicht werden, die einen eindeutigen booleschen Wert zurückgeben. Dazu ist eine etwas kompliziertere Syntax erforderlich, um `is` auf die Ausdrücke $2 = 2$ in der zweiten Ebene des Ergebnisses anzuwenden, da `is` nicht mit Listenbildung vertauscht. Es wäre ja auch unklar, mit welchem booleschen Operator die verschiedenen Ergebnisse zu verknüpfen wären.

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

$$[\,[\mathtt{TRUE},\mathtt{TRUE}],[\mathtt{TRUE},\mathtt{TRUE}],[\mathtt{TRUE},\mathtt{TRUE}],[\mathtt{TRUE},\mathtt{TRUE}]\,]$$

Mit entsprechenden Listenoperationen können wir die beiden Einzelergebnisse der Probe bis zu einer Liste von vier `true`-Werten umformen.

```
map(probe,x -> is(_and(op(x))));
```

$$[\mathtt{TRUE},\mathtt{TRUE},\mathtt{TRUE},\mathtt{TRUE}]$$

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

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

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

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.

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

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

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, MUPAD 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 und MUPAD 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. Beachten Sie die von anderen CAS abweichenden Bezeichner des Zuweisungsoperators.

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

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

```
/* Probe ausführen */
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 Liste */
every(%);
```

```
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 in Steuerstrukturen

Auch Boolesche Funktionen können in der ganzen Vielfalt von Formen auftreten, in welcher Funktionen im symbolischen Rechnen generell vorkommen. Boolesche Funktionsausdrücke bezeichnen wir auch kurz als *Boolesche Ausdrücke*. Es handelt sich dabei um Boolesche Funktionen mit Bezeichnern im Symbolmodus als Argumenten. Dies entspricht in der mathematischen Logik Booleschen Ausdrücken mit freien Variablen. Bekanntlich kann einem solchen Ausdruck erst dann ein Wahrheitswert zugeordnet werden, wenn all freien Variablen an konkrete Werte gebunden sind.

Boolesche Funktionen spielen in gewissen Steuerstrukturen (`while`, `if`) eine wichtige Rolle, wobei zur korrekten Ablaufsteuerung an dieser Stelle Funktions*aufrufe* abgesetzt werden müssen, die keinen Funktions*ausdruck*, sondern (garantiert) einen der beiden Werte `true` oder `false` zurückliefern. Dies gilt vor allem für relationale Operatoren wie `=` (equal), die in vielen anderen Kontexten auch als Operator*symbole* verwendet werden.

So wird Gleichheit in den verschiedenen CAS sowohl bei der Formulierung eines Gleichungssystems als auch dessen Lösung verwendet (exemplarisch in MuPAD).
In beiden Kontexten wurde `=` 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: `bool` oder `is`) erzwungen werden. Allerdings entsprechen die Ergebnisse nicht immer den Erwartungen (MuPAD, ähnlich auch die anderen CAS).

```
1=2;
```

$$1 = 2$$

aber

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

```
     no
```

Sehen wir uns die boolesche Auswertung einzelner Ausdrücke näher an.
Im ersten Fall ist die Antwort für alle Variablenbelegungen $(x, y) = (t, 3 - t)$ falsch, aber danach war hier nicht gefragt. Im zweiten Fall sind linke und rechte Seite syntaktisch (literal) gleich.

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

```
     FALSE
```

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

```
     true
```

Im ersten Beispiel sind die beiden Seiten der Gleichung *nach Auswertung* syntaktisch gleich, im zweiten Fall besteht semantische, nicht aber syntaktische Gleichheit.

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

```
     true
```

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

```
     FALSE
```

Eine ähnliche Wirkung hat die MuPAD-Funktion `is`, allerdings kommt das Ergebnis aus dem dreiwertigen booleschen Bereich {`TRUE`, `FALSE`, `UNKNOWN`}. `is` kann als zweistellige Funktion auch zur Analyse von Eigenschaften verwendet werden, die mit einzelnen Bezeichnern assoziiert sind.

Einfache Größenvergleiche symbolischer Ausdrücke, die zu reellen Zahlen auswerten, lassen sich auf diese Weise oft erfolgreich ausführen.

```
bool(1<sqrt(5));
```

```
     TRUE
```

Allerdings bleibt bei der Bestimmung des booleschen Werts von Zahlenvergleichen das *prinzipielle Problem* bestehen, 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. Das Auseinanderfallen kann erst durch sehr genaue Berechnung mit vielen Nachkommastellen bestätigt werden.

Beispiel: Die Folgenglieder $a_n = \alpha^n + \beta^n$ mit $\alpha = 1+\sqrt{5}$ und $\beta = 1-\sqrt{5}$ sind sämtlich ganzzahlig, da sich in der Expansion nach der binomischen Formel die Summanden, welche $\sqrt{5}$ mit ungeraden Exponenten enthalten, gerade wegheben.
Wegen $|\beta| < 1$ kommt also α^n ganzen Zahlen beliebig nahe. MuPAD versucht eine Entscheidung an Hand numerischer Näherungswerte zu finden. Das Ergebnis ist nur dann korrekt, wenn vorher die Zahl der Rechenziffern ausreichend hoch eingestellt wurde.
Mathematica dagegen findet ohne Änderungen an den Default-Einstellungen entweder die richtige Antwort (für s^n, $n \in \{10, 100\}$) oder gibt den Ausdruck unausgewertet zurück (für $n = 1000$).

```
s:=(1+sqrt(5)):
is(s^10=round(s^10));

     FALSE

is(s^100=round(s^100));

     TRUE

DIGITS:=50;
is(s^100=round(s^100));

     FALSE

is(s^1000=round(s^1000));

     TRUE
```

Andererseits gibt es Wurzelausdrücke, die exakt mit ganzen Zahlen oder einfacheren Wurzelausdrücken übereinstimmen, was aber in keiner Weise offensichtlich ist. So gilt zum Beispiel

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

Derartige Probleme muss ein CAS bei der Auswertung boolescher Ausdrücke korrekt behandeln können, wenn es für solche Relationen mit Wurzelausdrücken `true` oder `false` entscheiden wollte. MuPAD kommt mit beiden Problemstellungen jenseits des genannten Grenzbereichs inzwischen gut zurecht.

Die Beispiele zeigen, 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 Substitutionskomamndos (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. Diese stärkere Auswertung steht in einzelnen Systemen auch als Nutzerfunktion (Maple: `evalb`, MuPAD: `bool`) zur Verfügung.

Trotzdem lassen sich auch in Steuerstrukturen nicht alle Booleschen Konditionen bereits zur Definitionszeit auswerten. Einige CAS (Mathematica, Reduce) lassen deshalb auch Funktionsausdrücke zu, die Bezeichner für Steuerstrukturen enthalten.
Das Beispiel zeigt das entsprechende Verhalten von Mathematica.

```
u=If[x>0,1,2]
```

$$\texttt{If}[x > 0, 1, 2]$$

```
u /. x->1
```

$$1$$

```
u /. x->-1
```

$$2$$

Als Konsequenz treten die entsprechenden Steuerstruktur-Bezeichner selbst als Funktionssymbole auf und müssen als solche einen Rückgabewert haben. Davon macht etwa Reduce Gebrauch, wenn Listengenerierung und einige Listentransformationen über `for` realisiert 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[1] 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.

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

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 $\sin(2\,x)+\sin(3\,x-\frac{\pi}{2}) = 0$ auf diesen 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

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

[1] Zudem auf teilweise speziellen Datenstrukturen.

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);
```

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

Dabei 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.“ Dies *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).

3.1 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. Eine ähnliche Rolle spielt die MuPAD-Funktion `type(x)`.

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, wie MuPAD die Exponentialfunktion definiert:

```
expose(exp)
```

```
proc(x)
  name exp; local y, lny, c; option noDebug;
begin
  if args(0) = 0 then error("expecting one argument")
  else
    if x::dom::exp <> FAIL then return(x::dom::exp(args()))
    else
      if args(0) <> 1 then error("expecting one argument") end_if
    end_if
  end_if;
  case type(x)
    of DOM_SET do
    of "_union" do
      return(map(x, exp))
  end_case;
  if not testtype(x, Type::Arithmetical) then
    error("argument must be of 'Type::Arithmetical'")
  end_if;
  case type(x)
    of DOM_FLOAT do return(exp::float(x))
    of DOM_COMPLEX do ...
    of DOM_INTERVAL do return(exp::hull(x))
    of "_mult" do ...
    of "_plus" do ...
    of "ln" do return(op(x, 1))
    of "lambertW" do ...
  end_case;
  procname(x)
end_proc
```

Zunächst wird überprüft, ob das übergebene Argument x aus einem Grundbereich `x::dom` stammt, für den eine eigene `exp`-Funktion `x::dom::exp` definiert ist, und ggf. der nötige Funktionsaufruf abgesetzt. Danach wird die Korrektheit der Anzahl der Aufrufargumente geprüft.

Die meisten Zeilen des folgenden Codes beginnen mit `type(x)` und analysieren den Kopfterm des aufgerufenen Arguments. Sehen wir uns die einzelnen Zeilen nacheinander an. Die erste `case`-Anweisung untersucht, ob das Argument x eine Menge ist.

```
case type(x)
  of DOM_SET do
  of "_union" do
    return(map(x, exp))
end_case;
```

In diesem Fall wird die `exp`-Funktion rekursiv auf die einzelnen Elemente angewendet. Die `exp`-Funktion wird also automatisch über Mengen distributiert, nicht aber über Listen. Als nächstes wird geprüft, ob es sich beim Argument um einen arithmetischen Ausdruck handelt, ehe dieser nacheinander genauer analysiert wird.

```
exp([a,b,c])

    Error: argument must be of
    'Type::Arithmetical' [exp]

exp({a,b,c})
```

$$\left\{ \mathrm{e}^{a},\, \mathrm{e}^{b},\, \mathrm{e}^{c} \right\}$$

Dabei wird nacheinander untersucht, ob x eine float-Zahl, eine complex-Zahl, ein Intervall, ein Produkt, eine Summe, ein Ausdruck der Form $x = \ln(y)$ oder $x = \texttt{lambertW}(c, y)$ ist. Für jeden einzelnen Fall steht eine spezielle Routine bereit, nach welcher das Argument weiter verarbeitet wird.

Für eine float- oder complex-Zahl x etwa wird die numerische Näherungsfunktion `exp::float` aufgerufen, die ihrerseits mit `expose(exp::float)` studiert werden kann. An der Ausgabe ist zu erkennen, dass die wirkliche Arbeit von einer weiteren Funktion `specfunc::exp` ausgeführt wird, die nun wirklich eine Kernfunktion ist.

Ehe wir uns genauer anschauen, was im Fall eines Produkts bzw. einer Summe ausgeführt wird, werfen wir einen Blick auf die letzte Zeile. Diese wird ausgeführt, wenn keine der Regeln greift. In diesem Fall muss ein Funktionsausdruck mit dem Kopf `exp` zurückgegeben werden.

Genau ein solcher wird durch `procname(x)` aus dem Namen der aufrufenden Funktion 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);
```

$$\mathrm{e}^{\frac{2}{3}}$$

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

$$\mathrm{e}^{9}$$

Im Zweig `of "_mult" do` wird zunächst geprüft, ob x die Gestalt $x = y \cdot \pi\mathrm{i}$ mit $y \in \mathbb{Z}$ hat, da dann $\mathrm{e}^{y\pi\mathrm{i}}$ zu $(-1)^y$ vereinfacht werden kann.
Ist $y \in \mathbb{Q}$, so wird weiter geprüft, ob $0 \le y < 1$ gilt. Anderenfalls wird die Zerlegung $y = k + y'$ mit $0 \le y' < 1$ berechnet und $(-1)^k \exp(y')$ zurückgegeben.
Unklar bleibt, an welcher Stelle Ausdrücke etwa der Form $\exp(\frac{1}{2}\pi\mathrm{i})$ vereinfacht werden. Die genaue Aufrufhierarchie kann mit `prog::trace(exp)` verfolgt werden, aber laut Ausschrift wird die Funktion in diesem Fall gar nicht betreten. Der Grund ist, dass MuPAD einige vordefinierte Funktionswerte der exp-Funktion kennt, die unter `op(exp,[1,5])` als Hash-Tabelle abgespeichert sind. Diese Tabelle wird zuerst durchsucht.
Ähnlich wird ein Argument x analysiert, das aus Summanden besteht. Im letzten Teil wird schließlich geprüft, ob $x = \ln(y)$ ist und in diesem Fall wegen $\exp(x) = \exp(\ln(y)) = y$ das erste Argument `op(x,1)` des Aufrufarguments als Ergebnis zurückgegeben.

```
y := (x/PI)/I;
if testtype(y, DOM_INT)
  then return((-1)^y)
end_if;
if testtype(y, DOM_RAT) then
  if 1 <= y or y < 0 then
    y := 2*frac(y/2);
    if % < 1 then
      return(exp(PI*I*%))
    else
      return(-exp(PI*I*(% - 1)))
    end_if
  else break
  end_if
end_if;
of "_plus" do ...
of "ln" do return(op(x, 1))
```

Schauen wir uns noch einmal genauer die Behandlung von Argumenten des Typs `DOM_COMPLEX` an. Da x vor Betreten des Funktionskörpers ausgewertet wurde und damit etwa das Argument der Form `3*I+2+5*I-7` bereits zu `-5+8*I` zusammengefasst ist, kann x hier nur die Form `a+b*I` oder `b*I` haben mit exakten oder float-Werten a und b.

```
of DOM_COMPLEX do
  if domtype(op(x, 1)) = DOM_FLOAT
    or domtype(op(x, 2)) = DOM_FLOAT
      then return(exp::float(x))
  end_if;
  break
```

Ist einer der beiden Werte ein float-Wert, so wird die numerische Routine aufgerufen. Dies gilt selbst für `2+3.0*I`, dessen erstes Argument vom Typ `DOM_INT` und dessen zweites Argument vom Typ `DOM_COMPLEX` ist, so dass die Definition nicht zu passen scheint.

Die Lösung: Beim Auswerten wurde x in `2.0+3.0*I` verwandelt. Ist dagegen `a+b*I` ein exakter Ausdruck, so wird `break` wirksam und ein `exp`-Funktionsausdruck zurückgegeben.

```
exp(3.0*I)
```

$$-0.9899924966 + 0.1411200081\,\mathrm{i}$$

```
exp(2+3.0*I)
```

$$-7.315110095 + 1.042743656\,\mathrm{i}$$

```
2+3.0*I
```

$$2.0 + 3.0\,\mathrm{i}$$

```
exp(2+3*I)
```

$$\mathrm{e}^{2+3\,\mathrm{i}}$$

Das funktionale Transformationskonzept kann an manchen Stellen (die allerdings von den Systementwicklern explizit vorgesehen sein müssen) relativ einfach erweitert werden. Betrachten wir dazu die MAPLE-Funktion `expand`. `expand(A)` analysiert die Gestalt des Ausdrucks $A = f(x_1, \ldots, x_n)$ und ruft eine entsprechende Funktion

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

auf, welche die eigentliche Transformation vornimmt. Hier wird die Fähigkeit eines CAS zur Stringmanipulation verwendet, indem zur Laufzeit (!), während des Aufrufs von `expand`, aus einem in den Argumenten vorkommenden Funktionssymbol `f` ein neuer Funktionsname `` `expand/f` `` generiert und, sofern eine entsprechende Funktionsdefinition vorhanden ist, diese aufgerufen wird. Ansonsten bleibt $f(x_1, \ldots, x_n)$ unverändert. Dies ist zugleich der Mechanismus, der es einem Nutzer erlaubt, die Fähigkeiten des Systems zum „Expandieren“ für selbstdefinierte Funktionen zu erweitern.

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

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

definieren.

Dies lässt sich ähnlich auch mit MUPAD umsetzen: Hier kann einem (speziell als solches zu vereinbarenden) Funktionssymbol l, auf dem `expand` nichttrivial agieren soll, ein Attribut `expand` im objektorientierten Sinn (Slot) mit einer entsprechenden Funktion als Wert zugeordnet werden. Obigen Effekt für l kann man folgendermaßen erreichen:

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

Die erste Anweisung verwandelt das Symbol l in ein Funktionssymbol (vom Typ `DOM_FUNC_ENV`), das einen Eintrag `l::expand` erlaubt, die zweite Zuweisung fügt diesen Slot zur Funktionsdefinition hinzu.

Dasselbe 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 kurz den Quellcode der speziellen modularen Faktorisierungsroutine `‘mod/Factor‘(f,p)` an, die hier aufgerufen wird.

```
print(‘mod/Factor‘);

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.

In MUPAD wird diese Unterscheidung nach dem objektorientierten Ansatz aus dem `domtype` der Aufrufargumente deduziert:

```
P:=Dom::UnivariatePolynomial(x, Dom::IntegerMod(2));
f:=P(x^3+x^2-x+1);
```

$$(1 \bmod 2)\,x^3 + (1 \bmod 2)\,x^2 + (1 \bmod 2)\,x + (1 \bmod 2)$$

```
factor(f);
```

$$((1 \bmod 2)\,x + (1 \bmod 2))^3$$

```
expr(%);
```

$$(x+1)^3$$

Mit dem funktionalen Transformationskonzept lässt sich also recht konsequent ein objektorientierter Zugang umsetzen. Die (nicht leicht nachvollziehbaren) Details können für dieses Beispiel mit `expose(factor)` eingesehen werden, zeigen aber, dass auch hier nichts anderes ausgeführt wird als in MAPLE – eine detaillierte syntaktische Analyse der Aufrufparameter, um aus ihnen die erforderlichen Informationen über den Grundbereich zu extrahieren und dann die korrekte Faktorisierungsroutine über diesem Grundbereich anzuwerfen.

Die Nachteile dieses Konzepts 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 neu definieren.

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

3.2 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:

```
showrules log;

{log(1) => 0,
 log(e) => 1,
 log(e^~x) => x,
 df(log(~x),~x) => 1/x,
 df(log(~x/~y),~z) => df(log(x),z) - df(log(y),z)}
```

Die ersten beiden 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.

Anders in den beiden letzten Regeln, in denen Bezeichner auch als **formale Parameter** vorkommen, die als Platzhalter für beliebige Teilausdrücke auftreten. So vereinfacht REDUCE etwa `log(exp(A))` zu A, egal wie der Teilausdruck A beschaffen ist. Der Bezeichner `e` steht dagegen für das Symbol e 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 Funktionssymbole `df` und `log`) und Bezeichnern als Wertcontainer (hier: als formale Parameter) zu unterscheiden. Regeln mit formalen Parametern werden auch als *allgemeine Regeln* bezeichnet.

Ein solches Regelsystem kann durchaus einen größeren Umfang erreichen:

```
showrules sin;

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

Manche der angegebenen Regeln sind noch konditional untersetzt, d. h. 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.

Zusammenfassung

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 eines vorab zu berechnenden booleschen Wächterbedinung (guard clause) abhängen.

   Eine Regeldefinition besteht damit aus vier Teilen: `Rule(lhs, rhs, bool)`$(u_1, \ldots, u_n)$

MATHEMATICA	`lhs /; bool` $\rightarrow$ `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.

Diese Regeln werden automatisch angewendet, wenn REDUCE das Funktionssymbol `sin` in einem Ausdruck antrifft. So werden etwa mit den Regel 9 bis 11 Ausdrücke der Form

$$\sin(n \cdot \arctan(x))$$

aufgelöst.

```
sin(5*atan(x));
```

$$\frac{x^5 - 10\,x^3 + 5\,x}{\sqrt{x^2+1}\,(x^4 + 2\,x^2 + 1)}$$

Dasselbe Ergebnis kann man mit MUPAD erreichen, da die Definition von `sin` die Information enthält, dass

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

gilt, wie an diesem Quelltextfragment zu erkennen ist.

```
expose(sin)

...
of "arctan" do
  return(op(x)/sqrt(1 + op(x)^2))
```

Allerdings muss dazu über `expand` erst eine Transformation angestoßen werden, welche die Funktionssymbole `sin` und `arctan` unmittelbar zusammenbringt, indem die Mehrfachwinkelausdrücke aufgelöst werden.

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

$$\sin(5\,\arctan(x))$$

```
expand(%);
```

$$\frac{x^5}{(1+x^2)^{5/2}} - \frac{10\,x^3}{(1+x^2)^{5/2}} + \frac{5\,x}{\left(\sqrt{1+x^2}\right)^{5/2}}$$

```
normal(%);
```

$$\frac{x^5 - 10\,x^3 + 5\,x}{(1+x^2)^{5/2}}$$

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.

Substitution wird als lokale Regelanwendung realisiert (MATHEMATICA)

```
expr /. x -> A
```

Wertzuweisung ohne Auswertung wird als globale Regelanwendung realisiert (REDUCE).

```
let x=A
```

Mischung von Funktionsdefinition und global vereinbarter Regel (MATHEMATICA).
Zunächst wird die Funktion $f : x \to x^2 + 1$ definiert. Diese ist als Regel dem Symbol `f` zugeordnet.

```
f[x_]:=x^2+1;
?f
```

```
     Global‘f
     f[x_] := x^2 + 1
```

```
f[a+b]
```

$$1 + (a + b)^2$$

Nun wird zusätzlich eine Regel für f definiert, die f (mathematisch nicht korrekt) als lineares Funktional ausweist. Diese Regel ist dem Symbol auf dieselbe Weise zugeordnet wie die Funktionsdefinition.
Die Regeln werden intern sortiert, zunächst die Summenregel und erst danach die Funktionsdefintionsregel angewendet. Wäre die zweite Regel zuerst angewendet worden, so lautete das Ergebnis $1 + (a + b)^2$.

```
f[x_+y_]:=f[x]+f[y];
?f
```

```
     Global‘f
     f[(x_) + (y_)] := f[x] + f[y]
     f[x_] := x^2 + 1
```

```
f[a+b]
```

$$2 + a^2 + b^2$$

3.3 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.

`sqrt(36)`	$\Rightarrow$	6
`sin(PI/4)`	$\Rightarrow$	$\frac{1}{2}\sqrt{2}$
`tan(PI/6)`	$\Rightarrow$	$\frac{1}{3}\sqrt{3}$
`arcsin(1)`	$\Rightarrow$	$\frac{1}{2}\pi$

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

`gamma(1/2)`	$\Rightarrow$	$\sqrt{\pi}$
`int(exp(-x^2),x=0..infinity)`	$\Rightarrow$	$\frac{1}{2}\sqrt{\pi}$
`int(exp(-x^2),x=0..y)`	$\Rightarrow$	$\frac{1}{2}\sqrt{\pi}\,\mathrm{erf}\,(y)$
`sum(1/i^2,i=1..infinity)`	$\Rightarrow$	$\pi^2/6$
`sum(1/i^7,i=1..infinity)`	$\Rightarrow$	$\zeta(7)$
`sum(1/i^n,i=1..infinity)`	$\Rightarrow$	$\{\,\zeta(n)$ if $1 < \mathrm{Re}(n)$

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. Das letzte Ergebnis hat die interne Darstellung

```
piecewise([1 < Re(n), zeta(n)])
```

und liefert für den (nicht abgedeckten) Fall $1 \geq \mathrm{Re}(n)$ das (mathematisch korrekte) Ergebnis `undefined`.

Weiterhin wird auch eine Reihe komplizierterer Umformungen von einigen der Systeme automatisch ausgeführt wie z. B. (Maple automatisch, MuPAD erst nach Aufruf von `radsimp`):

`sqrt(24)`	$\Rightarrow$	$2\sqrt{6}$
`sqrt(2*sqrt(3)+4)`	$\Rightarrow$	$\sqrt{3}+1$
`sqrt(11+6*sqrt(2))+sqrt(11-6*sqrt(2))`	$\Rightarrow$	6

Auch werden eindeutige Simplifikationen von Funktionsausdrücken ausgeführt wie etwa

`abs(abs(x))`	$\Rightarrow$	$\lvert x\rvert$
`tan(arctan(x))`	$\Rightarrow$	x
`tan(arcsin(x))`	$\Rightarrow$	$\frac{x}{\sqrt{1-x^2}}$
`abs(-PI*x)`	$\Rightarrow$	$\pi\,\lvert x\rvert$
`cos(-x)`	$\Rightarrow$	$\cos(x)$
`exp(3*ln(x))`	$\Rightarrow$	x^3

Auf den ersten Blick mag es deshalb verwundern, dass folgende Ausdrücke nicht vereinfacht werden:

`sqrt(x^2)`	$\Rightarrow$	$\sqrt{x^2}$
`ln(exp(x))`	$\Rightarrow$	$\ln(\exp(x))$
`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 unmittelbar zeigt:

```
plot(plot::Function2d(arctan(tan(x)),x=-10..10));
```

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

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

wobei $[a]$ 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

$$\texttt{simplify(sqrt(x\^{}2))} \quad \Rightarrow \quad \operatorname{csgn}(x)\, x$$

zurück, obwohl wir in den Beispielen gesehen hatten, dass es 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 vereinfacht deshalb den Ausdruck auch unter `simplify` nicht.

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

Weitere interessante Simplifikationsfälle sind die Ausdrücke

$$\begin{aligned} &\texttt{sqrt(1/x)-1/sqrt(x)} && \Rightarrow && \sqrt{\frac{1}{x}} - \frac{1}{\sqrt{x}} \\ &\texttt{sqrt(x*y)-sqrt(x)*sqrt(y)} && \Rightarrow && \sqrt{x\,y} - \sqrt{x}\,\sqrt{y} \end{aligned}$$

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 haben wir in diesem Fall allerdings nicht nur die Mehrdeutigkeit der Wurzelfunktion im Bereich der komplexen Zahlen zu berücksichtigen, sondern kollidieren mit anderen, wesentlich zentraleren Annahmen, wie etwa der automatischen Ersetzung von $\sqrt{-1}$ durch die imaginäre Einheit i. Setzen wir in obigen Ausdrücken $x = y = -1$ und führen diese Ersetzung aus, so erhalten wir im ersten Fall $\mathrm{i} - 1/\mathrm{i} = 2\,\mathrm{i}$ und im zweiten Fall $\sqrt{1} - \mathrm{i}^2 = 2$. 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:

$$\texttt{ln((-1)*(-1))=ln(-1)+ln(-1)} \quad \Rightarrow \quad 0 = 2\,\mathrm{i}\,\pi$$

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	Axiom	Derive	Maxima	Maple	Mma	MuPAD	Reduce
	2008	6.1	5.13	11	6.0	4.0	3.8
$\lvert -\pi \cdot x\rvert$	$\lvert\pi\, x\rvert$	$\pi\,\lvert x\rvert$	$\pi\,\lvert x\rvert$	$\pi\,\lvert x\rvert$	$\pi\,\lvert x\rvert$	$\pi\,\lvert x\rvert$	$\pi\,\lvert x\rvert$
$\arctan(\tan(x))$	x	(1)	x	$*$	$*$	$*$	$*$
$\arctan(\tan(\frac{25}{7}\pi))$	$\frac{25}{7}\pi$	$-\frac{3}{7}\pi$	$\frac{25}{7}\pi$	$-\frac{3}{7}\pi$	$-\frac{3}{7}\pi$	$-\frac{3}{7}\pi$	(2)
$\sqrt{x^2}$	$*$	$\lvert x\rvert$	$\lvert x\rvert$	$*$	$*$	$*$	$\lvert x\rvert$
$\sqrt{x\,y}-\sqrt{x}\,\sqrt{y}$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
$\sqrt{\frac{1}{z}-\frac{1}{\sqrt{z}}}$	$*$	$\frac{\operatorname{sgn}(z)-1}{\sqrt{z}}$	0	$*$	$*$	$*$	$*$
$\ln(\exp(x))$	x	x	x	$*$	$*$	$*$	x
$\ln(\exp(10\,\mathrm{i}))$	$*$	$2\,\mathrm{i}\,(5-2\pi)$	$10\,\mathrm{i}$	$*$	$10\,\mathrm{i}-4\pi\,\mathrm{i}$	$-4\pi\,\mathrm{i}+10\,\mathrm{i}$	$10\,\mathrm{i}$

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

Tabelle 1: Simplifikationsverhalten der verschiedenen Systeme an ausgewählten Beispielen

Eine Bemerkung zu den beiden Beispielen $\arctan\left(\tan\left(\frac{25}{7}\pi\right)\right)$ und $\ln(\exp(10\,\mathrm{i}))$: Im ersten Fall wird das innere Argument von einigen CAS zu $\tan\left(\frac{4}{7}\pi\right)$ (Reduce) bzw. $-\tan\left(\frac{3}{7}\pi\right)$ (Maple, MuPAD) ausgewertet und damit bereits eine gewisse Simplifikation vorgenommen, im zweiten Fall dagegen nicht.

Das erstaunliche Ergebnis von Axiom, dass $\ln(\exp(10\,\mathrm{i}))$ falsch zu x, $\ln(\exp(10\,\mathrm{i}))$ aber nicht umgeformt wird, hat damit zu tun, dass der erste Ausdruck als vom Typ `Expression Integer` identifiziert wird, der zweite dagegen vom Typ `Expression Complex Integer`.

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.

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.

Symbole, für die Eigenschaften global definiert wurden, werden in Maple mit einer Tilde versehen. 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.

```
assume(y,real); about(y);

     Originally y, renamed y~:
     is assumed to be: real

assume(x>0); about(x);

     Originally x, renamed x~:
     is assumed to be:
     RealRange(Open(0),infinity)
```

Ähnlich ist das Vorgehen in MuPAD oder Maxima. Im Gegensatz zum Maple-Beispiel haben wir hier $x < 0$ angenommen.

MuPAD:

```
assume(y,Type::Real); assume(x<0);
getprop(x); getprop(y);

     Type::Real
```

Maxima:

```
declare(y,real); assume(x<0);
facts(x); facts(y);

     [0 > x]    [KIND(y, REAL)]
```

MuPAD liefert unter diesen Annahmen die nebenstehenden Ergebnisse. Dieses Verhalten ist sinnvoll, da für $x < 0$ das einfache Expandieren der Wurzel nicht korrekt ist.
Maxima liefert die ebenfalls korrekte Vereinfachung $\sqrt{-x}\,\sqrt{-y} - \sqrt{x}\,\sqrt{y}$.

```
abs(-PI*x); sqrt(x^2);
simplify(sqrt(x*y) -sqrt(x)*sqrt(y));
```

$$-x\,\pi$$
$$-x$$
$$\sqrt{x\,y} - \sqrt{x}\,\sqrt{y}$$

`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 wurde die bis dahin nur rudimentär vorhandene Möglichkeit zur Vereinbarung von Annahmen mit der Version 5 als Option `Assumptions` für die Befehle `Simplify`, `FullSimplify`, `Refine` oder `FunctionExpand` eingeführt und zugleich die Liste möglicher Eigenschaften deutlich erweitert.
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 DERIVE können ebenfalls Annahmen (positiv, nichtnegativ, reell, komplex, aus einem reellen Intervall) über die Natur einzelner Variablen getroffen werden.

In MAXIMA und REDUCE lässt sich außerdem die Strenge der mathematischen Umformungen durch verschiedene Schalter verändern. So kann man etwa in REDUCE über den (allerdings nicht dokumentierten) Schalter `reduced` die klassischen Vereinfachungen von Wurzelsymbolen, die für komplexe Argumente zu fehlerhaften Ergebnissen führen können, zulassen. In MAXIMA können die entsprechenden Schaltern 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 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 ([17, 16, 22, 23]), erkennen kann. Allerdings werden selbst innerhalb eines Systems an unterschiedlichen Stellen manchmal unterschiedliche Maßstäbe der mathematischen Strenge angelegt, insbesondere bei der Implementierung komplexerer Verfahren.

3.4 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. Wir wollen im Weiteren die *Kontextfreiheit* dieser Relation voraussetzen, dass also das Ersetzen eines Teilausdrucks durch einen semantisch äquivalenten Teilausdruck zu einem semantisch äquivalenten Ausdruck führt. Genauer fordern wir für alle x-freien Ausdrücke U, V mit $U \sim V$ und alle Ausdrücke A, dass $A(x \vdash U) \sim A(x \vdash V)$ 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[2].

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 $L(u \vdash U) > R(u \vdash U)$.

[2]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.

(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.

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 bezeichnet man auch als *starken Nulltester*.

3.5 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	`normal(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;
```

$$a^3 + 3\,a^2\,b + 3\,a\,b^2 + b^3$$

```
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);
```

$$D3^3 + D4^3 + 3\,D3\,D4^2 + 3\,D3^2\,D4,$$
$$\{D3 = \cos(x), D4 = \sin(x)\}$$

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 vorzunehmende 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);
```

$$/R/ \quad \cos(x)^3 + 3\cos(x)^2\sin(x) + 3\cos(x)\sin(x)^2 + \sin(x)^3$$

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 4.0 fasst allerdings automatisch $\exp(m\,x)\cdot\exp(n\,x)$ für $m, n \in \mathbb{Z}$ zu $\exp((m+n)\,x)$ zusammen.

```
v:=normal(u);
```

$$-\frac{\begin{pmatrix}\cos(x) + \cos(x)\ \sin(x)^2 - \cos(x)^3\ \mathrm{e}^{-2x} + \cos(x)^5\ \mathrm{e}^{-x} + \cos(x)^3 + \cos(x)\ \mathrm{e}^{-x} - \\ \cos(x)\ \mathrm{e}^{2\,x} - \cos(x)\ \mathrm{e}^{-3\,x} - \cos(x)^5\ \mathrm{e}^{x} - \cos(x)\ \sin(x)^2\ \mathrm{e}^{-2\,x} - \cos(x)\ \sin(x)^4\ \mathrm{e}^{x} + \\ \cos(x)\ \sin(x)^4\ \mathrm{e}^{-x} - 2\ \cos(x)^3\ \sin(x)^2\ \mathrm{e}^{x} + 2\ \cos(x)^3\ \sin(x)^2\ \mathrm{e}^{-x}\end{pmatrix}}{x^2\ \mathrm{e}^{x} - 2\,x^2\ \mathrm{e}^{-x} + x^2\ \mathrm{e}^{-3\,x}}$$

Maple belässt dabei den Nenner in faktorisierter Form, was aus algorithmischer Sicht vorteilhafter ist und die Normalformeigenschaft nicht zerstört. `normal(u1,expanded)` expandiert wie `expand` den Nenner, aber auch die verschiedenen exp-Kerne, was die Zahl der Kerne reduziert:

$$\frac{\begin{pmatrix}\cos(x)(\,\mathrm{e}^{x})^3 + \cos(x)(\sin(x))^4(\,\mathrm{e}^{x})^2 + (\cos(x))^5(\,\mathrm{e}^{x})^2 + \\ 2\,(\cos(x))^3(\sin(x))^2(\,\mathrm{e}^{x})^2 - \cos(x)(\sin(x))^2\,\mathrm{e}^{x} - (\cos(x))^3\,\mathrm{e}^{x} - \cos(x)\end{pmatrix}}{-x^2 + x^2(\,\mathrm{e}^{x})^2}$$

Ersetzt man zusätzlich $\sin(x)^2$ durch $1 - \cos(x)^2$, so ergibt sich eine besonders einfache Form des Ausdrucks.

```
simplify(v);
```

$$\frac{e^x\cos(x) + \cos(x)}{x^2}$$

Ein ähnliches Ergebnis liefert MuPAD mit `simplify(u)` auch aus dem Originalausdruck, kann dabei aber die Kerne nicht alle eliminieren.

```
simplify(u);
```

$$\frac{\cos(x)\,\mathrm{e}^{x}\,(\,\mathrm{e}^{-x} - 1)^2\,(\,\mathrm{e}^{-x} + 1)^3}{x^2\,(\,\mathrm{e}^{-2x} - 1)^2}$$

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.

```
rat(u1);
```

$$/R/\quad \frac{\begin{pmatrix}(e^x)^2\ \cos(x)\ \sin^4(x) + \left(2\ (e^x)^2\ \cos^3(x) - e^x\ \cos(x)\right)\ \sin^2(x) + (e^x)^2\ \cos^5(x) - \\ e^x\ \cos^3(x) + \left((e^x)^3 - 1\right)\ \cos(x)\end{pmatrix}}{x^2\ (e^x)^2 - x^2}$$

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]
```

$$\frac{\cos(x)\left(-1+\mathrm{e}^{3x}-\mathrm{e}^{x}\cos(x)^2+\mathrm{e}^{2x}\cos(x)^4-\mathrm{e}^{x}\sin(x)^2+2\,\mathrm{e}^{2x}\cos(x)^2\sin(x)^2+\mathrm{e}^{2x}\sin(x)^4\right)}{(-1+\mathrm{e}^{2x})\,x^2}$$

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.

3.6 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. Die verschiedenen Additionstheoreme zeigen, dass zwischen derartigen Ausdrücken eine große Zahl von Abhängigkeiten besteht. Die trigonometrischen 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.

$$f_1 := \sin(3x)\,\sin(5x),$$
$$f_2 := \cos\left(\frac{x}{2}\right)\,\cos\left(\frac{x}{3}\right),$$
$$f_3 := \sin\left(2x-\frac{\pi}{6}\right)\,\cos\left(3x+\frac{\pi}{4}\right),$$

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 Summe kann man leicht integrieren. Wollen wir aber jetzt prüfen, dass $f_1 = g_1'$ gilt, so sind diese Winkelfunktionen mit verschiedenen Argumenten zu vergleichen, wozu es notwendig ist, die Vielfachen von x als Argument wieder aufzulösen, also die entsprechenden Regeln in der anderen 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 **Produkt-Summe**.

$$\begin{aligned}\sin(x)\sin(y) &\Rightarrow\ 1/2\,(\cos(x-y)-\cos(x+y))\\ \cos(x)\cos(y) &\Rightarrow\ 1/2\,(\cos(x-y)+\cos(x+y))\\ \sin(x)\cos(y) &\Rightarrow\ 1/2\,(\sin(x-y)+\sin(x+y))\end{aligned}$$

Diese Regeln sind invers zu den Regeln **Summe-Produkt**, die man beim Expandieren von Winkelfunktionen, deren Argumente Summen oder Differenzen sind, anwendet.

$$\begin{aligned}\sin(x+y) &\Rightarrow\ \sin(x)\cos(y)+\cos(x)\sin(y)\\ \cos(x+y) &\Rightarrow\ \cos(x)\cos(y)-\sin(x)\sin(y)\end{aligned}$$

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.

$$\begin{aligned}\sin(-x) &\Rightarrow\ -\sin(x)\\ \cos(-x) &\Rightarrow\ \cos(x)\end{aligned}$$

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))};

trigexpand0:={ % Summen-Produkt-Regeln
        sin(~x+~y) => sin x * cos y + cos x * sin y,
        cos(~x+~y) => cos x * cos y - sin x * sin y};
```

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[3] 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;
```

$$\frac{-\cos(6\,x)+\cos(4\,x)-2\sin(x)^2}{4}$$

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 $\sin(x)^{n-2}$

[3] wie man mit `flagp('sin,'odd)` und `flagp('cos,'even)` erkennt

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(2x))/2,
        sin(~x)^2 => (1-cos(2x))/2};
```

Die Anwendung dieses Regelsystems **Produkt-Summe** führt auf eine kanonische Darstellung polynomialer trigonometrischer Ausdrücke, ist also für Simplifikationszwecke hervorragend geeignet. Genauer gilt folgender

Satz 4 *Sei R ein Unterkörper von $\mathbb{C}$. Dann kann jeder polynomiale Ausdruck $P(\sin(x), \cos(x))$ mit Koeffizienten aus R mit obigem Regelsystem* `trigsum` *in einen Ausdruck der Form*

$$\sum_{k>0} (a_k \sin(kx) + b_k \cos(kx)) + c$$

mit $a_k, b_k, c \in R$ verwandelt werden.

Diese Darstellung ist eindeutig. Genauer: Hat R eine kanonische oder Normalform, so kann diese Darstellung zu einer kanonischen bzw. Normalform für die Klasse der betrachteten Ausdrücke verfeinert werden.

Beweis: Da das Regelsystem alle Produkte und Potenzen von trigonometrischen Kernen ersetzt und sich in jedem Transformationsschritt die Anzahl der Multiplikationen verringert, ist nur die Eindeutigkeit der Darstellung zu zeigen. Die Koeffizienten a_k, b_k, c in einer Darstellung

$$f(x) = \sum_{k>0} (a_k \sin(kx) + b_k \cos(kx)) + c$$

kann man aber wie in der Theorie der Fourierreihen gewinnen. Berechnen wir etwa für ein festes ganzzahliges $n > 0$ das Integral

$$\begin{aligned}
2\int_0^{2\pi} f(x)\sin(nx)\,dx &= \sum_{k>0} a_k \int_0^{2\pi} 2\sin(kx)\sin(nx)\,dx \\
&\quad + \sum_{k>0} b_k \int_0^{2\pi} 2\cos(kx)\sin(nx)\,dx + 2c\int_0^{2\pi} \sin(nx)\,dx \\
&= \sum_{k>0} a_k \int_0^{2\pi} (\cos((k-n)x) - \cos((k+n)x))\,dx \\
&\quad + \sum_{k>0} b_k \int_0^{2\pi} (\sin((n-k)x) + \sin((n+k)x))\,dx \\
&= 2\pi\, a_n,
\end{aligned}$$

so sehen wir, 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. $\int_0^{2\pi} f(x)\,dx$. □

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.

```
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 schlielich das Endergebnis

$$\frac{35\,\sin(x)}{64}-\frac{21\,\sin(3\,x)}{64}+\frac{7\,\sin(5\,x)}{64}-\frac{\sin(7\,x)}{64}$$

feststeht.

Wir benötigen also 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:

```
trigsum={ (* Verwandelt Produkte in Summen von Mehrfachwinkeln *)
        Cos[x_]*Cos[y_] -> 1/2*(Cos[x+y]+Cos[x-y]),
        Sin[x_]*Sin[y_] -> 1/2*(-Cos[x+y]+Cos[x-y]),
        Sin[x_]*Cos[y_] -> 1/2*(Sin[x+y]+Sin[x-y]),
        Sin[x_]^(n_Integer)/; (n>1) -> (1-Cos[2*x])/2*Sin[x]^(n-2) ,
        Cos[x_]^(n_Integer)/; (n>1) -> (1+Cos[2*x])/2*Cos[x]^(n-2) ,
        (a_+b_)*c_ -> a*c+b*c};
```

Nun zeigt die Simplifikation das erwünschte Verhalten:

```
Sin[x]^7//.trigsum
```

$$\frac{35\,\sin(x)}{64} - \frac{21\,\sin(3\,x)}{64} + \frac{7\,\sin(5\,x)}{64} - \frac{\sin(7\,x)}{64}$$

Neben der Umwandlung von Potenzen der Winkelfunktionen in Summen mit Mehrfachwinkel-Argumenten ist an anderen Stellen, etwa beim Lösen goniometrischer Gleichungen, auch die umgekehrte Transformation von Interesse, da sie die Zahl der verschiedenen Funktionsausdrücke, die als Kerne in diesen Ausdruck eingehen, verringert. Auf diese Weise liefert die anschließende Berechnung der rationalen Normalform oft ein besseres Ergebnis.

Mathematisch kann man die entsprechenden Regeln aus der *Moivreschen Formel* für komplexe Zahlen und den Potenzgesetzen herleiten: Aus

$$\cos(nx) + \mathrm{i}\,\sin(nx) = e^{\,\mathrm{i}nx} = (e^{\,\mathrm{i}x})^n = (\cos(x) + \mathrm{i}\,\sin(x))^n$$

und den binomischen Formeln ergibt sich durch Vergleich der Real- und Imaginärteile unmittelbar

$$\cos(nx) = \sum_{2k\le n} (-1)^k \binom{n}{2k} \sin(x)^{2k}\cos(x)^{n-2k}$$

und

$$\sin(nx) = \sum_{2k<n} (-1)^k \binom{n}{2k+1} \sin(x)^{2k+1}\cos(x)^{n-2k-1}$$

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 MuPAD etwa könnte man für die zweite Formel eine Funktion `Msin` als

```
Msin:=proc(n,x) local k;
begin
  _plus((-1)^k*binomial(n,2k+1)*sin(x)^(2k+1)*cos(x)^(n-2k-1) $k=0..n/2)
end_proc;
```

vereinbaren und in der Regel

```
r:=Rule(sin(n*x), hold(Msin)(n,x), (n,x) -> hastype(n,Type::Integer));
Rule::apply(r,sin(7*y));
```

$$7\,\cos(y)^6\,\sin(y) - 35\,\cos(y)^4\,\sin(y)^3 + 21\,\cos(y)^2\,\sin(y)^5 - \sin(y)^7$$

einsetzen. Beachten Sie, dass auch hier `Msin` in der Regeldefinition durch `hold` vor Auswerten geschützt sein muss, denn die Funktion soll ja erst zur Regelanwendung ausgeführt werden.

Einfacher ist es allerdings auch in diesem Fall, die Regeln aus dem Ansatz

$$\sin(kx) = \sin\left((k-1)x + x\right) = \sin\left((k-1)x\right)\cos(x) + \cos\left((k-1)x\right)\sin(x)$$

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, komplizierte trigonometrische Ausdrücke, in deren Argumenten Summen und Mehrfachwinkel auftreten, durch trigonometrische Ausdrücke mit nur noch wenigen verschiedenen und einfachen Argumenten zu ersetzen. Hängen die Argumente nur ganzzahlig von einer Variablen x ab, so können wir das System wiederum zu einer kanonischen oder Normalform erweitern. Genauer gesagt gilt der folgende Satz:

Satz 5 *Sei R ein Unterring von $\mathbb{C}$. Dann kann man jeden polynomialen Ausdruck in $\sin(k\,x)$ und $\cos(k\,x)$, $k \in \mathbb{Z}$, mit Koeffizienten aus R durch obiges Regelsystem* `trigexpand` *und die zusätzliche Regel $\{\sin(x)^2 \Rightarrow 1 - \cos(x)^2\}$ in einen Ausdruck der Form*

$$P(\cos(x)) + \sin(x)\,Q(\cos(x))$$

verwandeln, wobei $P(z)$ und $Q(z)$ Polynome mit Koeffizienten aus R sind.

Diese Darstellung ist eindeutig. Genauer: Hat R eine kanonische oder Normalform, so kann diese Darstellung zu einer kanonischen bzw. Normalform für die Klasse der betrachteten Ausdrücke verfeinert werden.

Beweis: Es ist wiederum nur die Eindeutigkeit zu zeigen.

$$P_1(\cos(x)) + \sin(x)\,Q_1(\cos(x)) = P_2(\cos(x)) + \sin(x)\,Q_2(\cos(x))$$

gilt aber genau dann, wenn $P(\cos(x)) + \sin(x)\,Q(\cos(x))$ mit den Polynomen $P = P_1 - P_2$ und $Q = Q_1 - Q_2$ identisch verschwindet. Wir haben also nur zu zeigen, dass aus der Tatsache, dass $P(\cos(x)) + \sin(x)\,Q(\cos(x))$ identisch verschwindet, bereits folgt, dass $P(z)$ und $Q(z)$ beides Nullpolynome sind. Aus $P(\cos(x)) + \sin(x)\,Q(\cos(x)) = 0$ folgt aber nach der Substitution $x = -x$ auch $P(\cos(x)) - \sin(x)\,Q(\cos(x)) = 0$ und schließlich $P(\cos(x)) = Q(\cos(x)) = 0$. Da ein nichttriviales Polynom aber nur endlich viele Nullstellen besitzt, folgt die Behauptung. □

Mit diesen beiden Strategien lassen sich die drei Integrationsaufgaben, mit denen wir diesen Abschnitt begonnen hatten, zufriedenstellend lösen. 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`.

Auf trigonometrische Ausdrücke angewendet bewirken `combine` (Produkte in Winkelsummen) und `expand` (Winkelsummen in Produkte) die obigen Umformungen, während man mit `rewrite` (u.a.) trigonometrische Funktionen und deren Umkehrfunktionen in Ausdrücke mit exp und log von komplexen Argumenten umformen kann. Dies entspricht dem REDUCE-Regelsatz

```
trigexp:={
        tan(~x) => sin(x)/cos(x),
        sin(~x) => (exp(i*x)-exp(-i*x))/(2*i),
        cos(~x) => (exp(i*x)+exp(-i*x))/2};
```

Diese Umformungen sind allerdings nur für solche Systeme sinnvoll, die mit `exp`-Ausdrücken gut rechnen können, etwa wenn sie die Regel

```
exp(n~*x~) => exp(x)^n
```

für $n \in \mathbb{Z}$ anwenden, um die Zahl unterschiedlicher `exp`-Kerne überschaubar zu halten. Dies kann am Ausdruck

$$\frac{\exp(5\,x) + \exp(3\,x)}{\exp(5\,x) - \exp(3\,x)}$$

getestet werden, der dann bei der Berechnung der rationalen Normalform zum Ausdruck

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

vereinfacht wird.

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`	`combine`	`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. Ähnlich kann man in MuPAD die Funktionsaufrufe `combine` und `expand` durch die Definition entsprechender Attribute auf andere Funktionssymbole ausdehnen.

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.

Kehren wir zur Berechnung der Beispielintegrale zurück. Für Integrale von rationalen trigonometrischen Ausdrücken liefert MuPAD 4.0, Reduce 3.7, Mathematica 6 und Maxima 5.14 die

folgenden Resultate:

$$\begin{aligned}
\int \frac{1}{\cos(x)^4}\,dx &= \frac{\sin(x)\,\left(2\,\cos(x)^2+1\right)}{3\,\cos(x)^3} && \text{(MuPAD)}\\
&= \frac{\sin(x)\,\left(2\sin(x)^2-3\right)}{3\cos(x)\,(\sin(x)^2-1)} && \text{(Reduce)}\\
&= \frac{1}{3}\,\tan(x)\,\sec^2(x)+\frac{2}{3}\,\tan(x) && \text{(Mathematica)}\\
&= \frac{1}{3}\,\tan(x)^3+\tan(x) && \text{(Maxima)}\\
\int \frac{1}{\sin(x)^2\,(1-\cos(x))}\,dx &= -\frac{1}{12}\,\cot\left(\frac{x}{2}\right)^3-\frac{1}{2}\,\cot\left(\frac{x}{2}\right)+\frac{1}{4}\,\tan\left(\frac{x}{2}\right) && \text{(MuPAD)}\\
&= \frac{2\sin(x)^2+2\cos(x)-1}{3\,\sin(x)\,(\cos(x)-1)} && \text{(Reduce)}\\
&= \frac{\sin(x)-(\cos(x)-2)\,\cot(x)}{3\,(\cos(x)-1)} && \text{(Mathematica)}\\
&= \frac{2\,\cos^3(x)-3\,\cos(x)-1}{3\,\sin^3(x)} && \text{(Maxima)}\\
\int \frac{1}{\sin(2x)\,(3\tan(x)+5)}\,dx &= \frac{1}{10}\,\ln\left(-\frac{3}{5}\,\tan(x)\right)-\frac{1}{10}\,\ln\left(-\frac{3}{5}\,\tan(x)+1\right) && \text{(MuPAD)}\\
&= \frac{-\ln(3\tan(x)+5)+\ln(\tan(x))}{10} && \text{(Reduce)}\\
&= -\frac{1}{5}\,\operatorname{arctanh}\left(\frac{6}{5}\,\tan(x)+1\right) && \text{(Mathematica)}\\
&= \texttt{error} && \text{(Maxima)}
\end{aligned}$$

Es ist bereits eine kleine Herausforderung, die semantische Gleichwertigkeit dieser syntaktisch verschiedenen Ergebnisse festzustellen. Die verschiedenen Berechnungsverfahren gründen meist auf der Möglichkeit, die Kerne $\sin(x)$, $\cos(x)$ 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` und `cos` ausgedrückt sind:

```
trigtan:={
        sin(~x) => (2*tan(x/2))/(1+tan(x/2)^2),
        cos(~x) => (1-tan(x/2)^2)/(1+tan(x/2)^2)};
```

Die dazu inverse Regel

```
invtrigtan:={ tan(~x) => sin(2x)/(1+cos(2x) };
```

erlaubt es, Halbwinkel im Ergebnis wieder so weit als möglich zu eliminieren. Grundlage dieses Vorgehens ist die Fähigkeit der Systeme, rationale Funktionen in der Integrationsvariablen zu integrieren sowie der

Satz 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. □

3.7 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

$$\frac{\cos(x)\,(\mathrm{e}^x+1)}{x^2}\,.$$

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 dieses Ausdrucks v zu erkennen, der ja hinter vielen verschiedenen `exp`-Kernen verborgen ist. In den Versionen 4 – 6 wurde das korrekte Ergebnis e^x gefunden. In den Versionen 7 – 9 hatte MAPLE mit diesem Ausdruck die alten Schwierigkeiten. Die Zusammenfassung der `exp`-Kerne ist nur mit `normal(.., expanded)` möglich.

```
v:=(exp(x)-exp(-x))/(1-exp(-2*x));
simplify(v);
```

$$-\frac{\mathrm{e}^x - \mathrm{e}^{-x}}{-1+\mathrm{e}^{-2\,x}}$$

Ähnliches gilt für MUPAD 4.0. Zwar wird die Vereinfachung von v korrekt vorgenommen, jedoch hat `simplify` Probleme, das noch komplizierte Ergebnis weiter zu vereinfachen, weil die verschiedenen `exp`-Terme als unterschiedliche Kerne aufgefasst werden.

```
simplify(u1);
```

$$\frac{\cos(x)\;\mathrm{e}^x\;\left(\mathrm{e}^{-x}-1\right)^2\;\left(\mathrm{e}^{-x}+1\right)^3}{x^2\;\left(\mathrm{e}^{-2x}-1\right)^2}$$

```
simplify(%,exp): simplify(%);
```

$$\frac{\cos(x)\;\left(\mathrm{e}^x+1\right)}{x^2}$$

REDUCE kommt mit `trigsimp`, DERIVE und MATHEMATICA mit `Simplify` bzw. `Togather` zu dem erwarteten Ergebnis.

MAXIMA kann den Ausdruck mit einer speziellen Simplifikationsroutine für trigonometrische Funktionen ebenfalls zufriedenstellend vereinfachen.

```
trigsimp(u1);
```

$$\frac{\cos(x)\ (\mathrm{e}^x+1)}{x^2}$$

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.

Das kann man etwa durch Umformung in Exponentialausdrücke erreichen. REDUCE erkennt damit bereits, dass dieser Ausdruck null-äquivalent ist.

```
trigsimp(u2,expon);
```

$$0$$

Dasselbe Ergebnis erhielt man in MAPLE 3 über die zwei Zwischenschritte u_{2a} und u_{2b}. Um den Ausdruck auch in neueren Versionen zu Null zu vereinfachen, ist noch eine zusätzliche Anwendung der Potenzgesetze notwendig.

```
u2a:=convert(u2, exp);
u2b:=expand(u2a);
combine(u2b, power);
```

Ähnlich kann man in MAXIMA und MUPAD vorgehen, während in MATHEMATICA wiederum ein Aufruf der Funktion `Simplify` genügt. DERIVE genügt ein Aufruf von Simplify, wenn man vorher `Trigonometry:=Expand` gesetzt hat.

System	Beispiel `u1`	Beispiel `u2`
MAXIMA	`trigsimp(u1))`	`expand(exponentialize(u2))`
MATHEMATICA	`u1 // Simplify`	`u2 // Simplify`
MUPAD	siehe oben	`combine(expand(rewrite(u2,exp)),exp)`
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 wird in DERIVE beim Vereinfachen automatisch vorgenommen, ist in MAXIMA, MAPLE und MUPAD durch eingebaute Funktionen (`convert(u3,ln)` oder `rewrite(u3,ln)`) 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 nur MATHEMATICA mit einem einigermaßen passablen 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): $\log(\sec(\frac{x}{2})+\tan(\frac{x}{2}))$ hat als Definitionsbereich $D=\{x : \cos(\frac{x}{2})>0\}$. 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 des ersten Logarithmanden eine imaginäre Restgröße stehen bliebe. 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.

Wir wandeln dazu wieder zunächst die Hyperbelfunktionen in Logarithmen um, fassen diese zusammen und extrahieren das gemeinsame Argument. In MAPLE erhalten wir unter der Annahme $-1<x<1$ einen Ausdruck in trigonometrischen Funktionen, mit dem das CAS aber nicht Vernünftiges anfangen kann.

```
assume(-1<x,x<1);
A1:=convert(u3,ln);
combine(%,ln);
A2:=exp(%);
```

$$\frac{\tan\left(\frac{x}{2}\right)+\sec\left(\frac{x}{2}\right)}{\frac{\sin(x)}{1+\cos(x)}+\sqrt{\frac{\sin(x)^2}{(1+\cos(x))^2}+1}}$$

Eine zielgerichtete Transformation hilft witer: Vor dem eigentlichen `simplify` werden alle Bestandteile zunächst in Winkelfunktionen von $y=\frac{x}{2}$ als Kernen umgerechnet.

```
A3:=expand(subs(x=2*y, A2));
simplify(A3);
```

$$\frac{\sin(y)+1}{\sin(y)+\operatorname{csgn}(\cos(y))}$$

Allerdings wurde dabei die bestehende Einschränkung $-1<x<1$ nicht auf y übertragen. Dies müssen wir per Hand nachholen.

```
about(x); about(y);
assume(-1/2<y,y<1/2);
A3;
```

$$1$$

Ähnlich können wir in MUPAD 4.0 vorgehen. Die Rechnung führt auf einen Ausdruck ähnliche Qualität wie oben. Allerdings ist MUPAD nun mit seinem Latein weitgehend am Ende.

```
assume(-1<x): assume(x<1, _and):
A1:=rewrite(u3,ln):
A2:=combine(A1,ln):
A3:=exp(A2);
```

DERIVE vereinfacht den Ausdruck mit `Simplify` zu

$$\log\left(\frac{2\cos\left(\frac{x}{2}\right)+\sin(x)}{\sqrt{2}\sqrt{1+\cos(x)}+\sin(x)}\right)$$

Setzt man danach `Trigonometry:=Collect`, so wird dieser Ausdruck weiter zu

$$\log\left(\frac{2\cos\left(\frac{x}{2}\right)+\sin(x)}{2\left|\cos\left(\frac{x}{2}\right)\right|+\sin(x)}\right)$$

vereinfacht. Setzen wir noch $x \in$ `Real`$(-1,1)$ (Author $\rightarrow$ Variable Domain), so wird dieser Ausdruck schließlich zu 0 vereinfacht.

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 von Hauptwert nur um ein ganzzahliges Vielfaches von $2\pi\,\mathrm{i}$ unterscheiden. Die weitere Vereinfachung ergibt

```
trigsimp(A2); A3:=subs(y=2x,ws);
```

$$\frac{\left|1-\sin(y)^2\right|\left(1+\sin(y)\right)}{\left|1-\sin(y)^2\right|\sin(y)+\sqrt{1-\sin(y)^2}\cos(y)}$$

```
A3 where sin(y)^2=>1-cos(y)^2;
```

liefert dann

$$\frac{\cos(y)\left(\sin(y)+1\right)}{\left|\cos(y)\right|+\cos(y)\,\sin(y)}.$$

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 (der hier nicht geführt wird) wird zurückgeführt auf die negative Antwort zum 10. Hilbertschen Problem, die Matiyasewitsch und Roberts Ende der 60er Jahre gefunden haben.

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] 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.

[11] J. Nolan. Analytic differentiation on a digital computer. Master's thesis, Math. Dept., MIT, Cambridge, Mass., 1953.

[12] T. Ottmann and P. Widmeyer. *Algorithmen und Datenstrukturen*. BI Wissenschaftsverlag, 2 edition, 1993.

[13] R. Pavelle, M. Rothstein, and J.P. Fitch. Computer algebra. *Scientific American*, 245(6):102 – 113, dec 1981.

[14] 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.

[15] 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.

[16] U. Schwardmann. *Computeralgebrasysteme*. Addison-Wesley, 1995.

[17] B. Simon. Comparative CAS review. *Notices AMS*, 39:700 – 710, sept 1992.

[18] D.R. Stoutemyer. PICOMATH-80, an even smaller computer algebra package. *SIGSAM Bull.*, 14.3:5–7, 1980.

[19] B. Stroustrup. *Die C++-Programmiersprache*. Addison-Wesley, 2 edition, 1992.

[20] 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.

[21] J. Weizenbaum. *Die Macht der Computer und die Ohnmacht der Vernunft*, volume 274 of *Taschenbuch Wissenschaft*. Suhrkamp, 9 edition, 1994.

[22] M. Wester. A review of CAS mathematical capabilities. *CAN Nieuwsbrief*, 13:41–48, dec 1994.

[23] M. Wester, editor. *Computer Algebra Systems: A Practical Guide*. Wiley, Chichester, 1999.

[24] N. Wirth. *Algorithmen und Datenstrukturen*. B.G. Teubner Verlag Stuttgart, 4 edition, 1986.

[25] S. Wolfram. *The Mathematica Book*. Wolfram Media and Cambridge University Press, 4 edition, 1999.